HTCondor Manual

Release 23.0.8

HTCondor Team

Apr 11, 2024

QUICK START GUIDES

The HTCondor Software Suite (HTCSS) is a software system that creates a High-Throughput Computing (HTC) environment. This environment might be a single cluster, a set of related clusters on a campus, cloud resources, or national or international federations of computers.

If you are a user of HTCondor, and have been given a login or credentials to use a batch scheduler on an Access Point (sometimes called a scheduler or login node), you may want to read our Quick Start guide here: *Users' Quick Start Guide*

If you are beginning administrator of HTCondor, or want to install it for the first time, please look at our installation guide here: *Downloading and Installing*

Otherwise, for users of HTCondor who want more information, a complete user's reference manual is here: *Users' Manual*, and a similar complete reference for administrators of HTCondor can be found here: *Administrators' Manual*

HTCondor contains many command line tools, each with a traditional Unix "man-page". These may be found here: *Command Reference Manual (man pages)*

Finally, for users writing Python interfaces to HTCondor, our Python API documentation is here: *Python Bindings*

A complete table of contents follows.

Manual built on April 11, 2024

CHAPTER

ONE

USERS' QUICK START GUIDE

HTCondor is a system for dynamically sharing computational resources between competing computational tasks. As an HTCondor user, you will describe your computational tasks as a series of independent, asynchronous "jobs." You access computational resources managed by HTCondor by submitting (or "placing") job descriptions at an HTCondor "access point" (AP), also known as a "submit node." HTCondor locates an appropriate machine for each job, packages up the job and ships it off to that machine for execution. Machines providing resources to HTCondor are therefore known as execution points (EP).

This guide covers submitting and observing the successful completion of a first, example job. It then suggests extensions that you can apply to your own jobs.

This guide presumes that

- HTCondor is running
- You have access to a machine within the pool that may submit jobs, termed an Access Point (AP).
- You are logged in to and working on the AP. (If you just finished *getting HTCondor*, the one machine you just installed is this AP.)
- Your program executable, your submit description file, and any needed input files are all on the file system of the AP.
- Your job (the program executable) is able to run without any interactive input. Standard input (from the keyboard), standard output (seen on the display), and standard error (seen on the display) may still be used, but their contents will be redirected from/to files.

1.1 What is a Job?

"Job" is a very specific term in HTCondor. A job is the atomic unit of work. A job may use multiple cores on one machine, but one job may not (in general) run across more than one machine. To effectively use HTCondor, you will need to divide your total work (often called a workflow) into a number of jobs. These atomic units of work run asynchronously with respect to each other, but may be connected by input and output files. Each job is described by a Job ClassAd, which is usually created by the system from a submit description file. HTCondor is a High Throughput system, which means it has been designed to effectively manage hundreds of thousands of jobs. Attributes of jobs that must be defined include the executable or script to run, the amount of memory, CPU and other machine resources it needs, and descriptions of the file inputs it need. The set of files used by a job is called the "sandbox". There is an input sandbox, the input files that exist before a job starts; the output sandbox, the set of files created by the job; and a scratch sandbox, the set of files made as the job runs.

1.2 A First HTCondor Job

For HTCondor to run a job, it must be given details such as the names and location of the executable and all needed input files. These details are specified in a submit description file.

The executable

Before presenting the details of the submit description file, consider this first HTCondor job. It is a sleep job that waits for 6 seconds and then exits. While most aspects of HTCondor are identical on Linux (or Mac) and Windows machines, awareness of the AP's operating system will lead to a better understanding of jobs and job submission.

This first executable program is a shell script (Linux or Mac) or batch file (Windows). The file that represents this differs based on operating system; the Linux (or Mac) version is shown first, and the Windows version is shown second. To try this example, log in to the AP, and use an editor to type in or copy and paste the file contents. Name the resulting file `sleep.sh` if the AP is Linux (or Mac) operating system, and name the resulting file `sleep.bat` if the AP is running Windows. Note that you will need to know whether the operating system on your AP is a Linux (or Mac) operating system or Windows.

Listing 1: Linux (or Mac) executable, a shell script

```
#!/bin/bash
# file name: sleep.sh

TIMETOWAIT="6"
echo "sleeping for $TIMETOWAIT seconds"
/bin/sleep $TIMETOWAIT
```

Listing 2: Windows executable, a batch file

```
:: file name: sleep.bat
@echo off

set TIMETOWAIT=6
echo sleeping for %TIMETOWAIT% seconds
choice /D Y /T %TIMETOWAIT% > NUL
```

For a Linux (or Mac) AP only, change the `sleep.sh` file to be executable by running the following command:

```
chmod u+x sleep.sh
```

The contents of the submit description file

The submit description file describes the job. To submit this sample job, again use an editor to create the file `sleep.sub`. The submit description file contents for this job differs on Linux (or Mac) and Windows machines only in the name of the script or batch file:

Listing 3: Linux (and Mac) submit description file

```
# sleep.sub -- simple sleep job

executable              = sleep.sh
```

(continues on next page)

(continued from previous page)

```
log                     = sleep.log
output                  = sleep.out
error                   = sleep.err

should_transfer_files   = Yes
when_to_transfer_output = ON_EXIT

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

queue
```

Listing 4: Windows submit description file

```
# sleep.sub -- simple sleep job

executable              = sleep.bat

log                     = sleep.log
output                  = sleep.out
error                   = sleep.err

should_transfer_files   = Yes
when_to_transfer_output = ON_EXIT

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

queue
```

The first line of this submit description file is a comment. Comments begin with the # character. Comments do not span lines.

Each line of the submit description file has the form

```
command_name = value
```

The command name is case insensitive and precedes an equals sign. Values to right of the equals sign are likely to be case sensitive, especially in the case that they specify paths and file names.

Next in this file is a specification of the `executable` to run. It specifies the program that becomes the HTCondor job. For this example, it is the file name of the Linux (or Mac) script or Windows batch file. A full path and executable name, or a path and executable relative to the current working directory may be specified.

The `log` command causes a job event log file named `sleep.log` to be created on the AP once the job is submitted. A log is not necessary, but it can be incredibly useful in figuring out what happened or is happening with a job.

HTCondor must be told how many resources your job needs on an Execution Point in order to run. This allows HTCondor to run as many jobs as possible on each EP without overloading them. Jobs must declare the number of CPUs, the amount of memory and disk they need. Special jobs may need to request other resources, such as GPUs or licenses. Ask your administrator if your jobs requires such things. The amount of cpus is unit less, but memory and disk requires can have a “M” for megabyte, “G” for Gigabyte suffix for legibility. Without the suffix, memory units are megabytes and disk kilobytes.

```
request_cpus            = 1
request_memory          = 512M
request_disk            = 1G
```

If this script/batch file were to be invoked from the command line, and outside of HTCondor, its single line of output

```
sleeping for 6 seconds
```

would be sent to standard output (the display). When submitted as an HTCondor job, standard output of the job is on that EP, and thus unavailable. HTCondor captures standard output in a file due to the `output` command in the submit description file. This example names the redirected standard output file `sleep.out`, and this file is returned to the AP when the job completes. The same structure is specified for standard error, as specified with the `error` command.

The commands

```
should_transfer_files   = Yes
when_to_transfer_output = ON_EXIT
```

direct HTCondor to explicitly send the needed files, including the executable, to the machine where the job executes. These commands will likely not be necessary for jobs in which the AP and the EP (the Execution Point, or worker node) access a shared file system. However, including these commands will allow this first sample job to work under a large variety of pool configurations.

The `queue` command tells HTCondor to run one instance of this job.

Submitting the job

With this submit description file, all that remains is to hand off the job to HTCondor. Note that the `queue` command should be the last command in the file. Commands after the `queue` are ignored. Otherwise, the order of commands with the file does not matter. Assuming the current working directory contains the `sleep.sub` submit description file and the executable (`sleep.sh` or `sleep.bat`), the command line

```
condor_submit sleep.sub
```

submits the job to the AP. If the submission is successful, the terminal will display a response that identifies the job, of the form

```
Submitting job(s).
1 job(s) submitted to cluster 6.
```

Monitoring the job

Once the job has been submitted, command line tools may help you follow along with the progress of the job. The `condor_q` command prints a listing of all your jobs currently in the queue. For example, a short time after Kris submits the sleep job from a Linux (or Mac) AP on a pool that has no other queued jobs, the output may appear as

```
$ condor_q
-- Submitter: example.wisc.edu : <128.105.14.44:56550> : example.wisc.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
    6.0   kris            2/13 10:49   0+00:00:03 R  0   97.7 sleep.sh

1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended
```

The first column of output from `condor_q` identifies the job; the identifier is composed of two integers separated by a period. The first integer is known as a cluster number, and it will be the same for each of the potentially many jobs submitted by a single invocation of `condor_submit`. The second integer in the identifier is known as a process ID, and it distinguishes between distinct job instances that have the same cluster number. These values start at 0.

Of interest in this output, the job is running, and it has used 3 seconds of time so far.

At job completion, the log file contains

```
000 (006.000.000) 02/13 10:49:04 Job submitted from host: <128.105.14.44:46062>
...
001 (006.000.000) 02/13 10:49:24 Job executing on host: <128.105.15.5:43051?PrivNet=cs.
↪wisc.edu>
...
006 (006.000.000) 02/13 10:49:30 Image size of job updated: 100000
        0  -  MemoryUsage of job (MB)
        0  -  ResidentSetSize of job (KB)
...
005 (006.000.000) 02/13 10:49:31 Job terminated.
        (1) Normal termination (return value 0)
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
        23  -  Run Bytes Sent By Job
        113  -  Run Bytes Received By Job
        23  -  Total Bytes Sent By Job
        113  -  Total Bytes Received By Job
        Partitionable Resources :    Usage  Request Allocated
           Cpus                 :                1         1
           Disk (KB)            :   100000   100000   2033496
           Memory (MB)          :        0       98      2001
...
```

Each event in the job event log file is separated by a line containing three periods. For each event, the first 3-digit value is an event number.

Removing a job

Successfully submitted jobs will occasionally need to be removed from the queue. The `condor_rm` command with the job identifier as a command line argument removes jobs. Kris' job may be removed from the queue with

```
condor_rm 6.0
```

Specification of the cluster number only as with the command

```
condor_rm 6
```

will cause all jobs within that cluster to be removed.

1.3 The science Job Example

A second example job illustrates aspects of file specification for the job. Assume that the program executable is called `science.exe`. This program does not use standard input or output; instead, the command line to invoke this program specifies two input files and one output file. For this example, the command line to invoke `science.exe` (not as an HTCondor job) will be

```
science.exe infile-A.txt infile-B.txt outfile.txt
```

While the name of the executable is specified in the submit description file with the `executable` command, the remainder of the command line will be specified with the `arguments` command.

Here is the submit description file for this job:

```
# science1.sub -- run one instance of science.exe
executable              = science.exe
arguments               = "infile-A.txt infile-B.txt outfile.txt"

transfer_input_files    = infile-A.txt,infile-B.txt
should_transfer_files   = IF_NEEDED
when_to_transfer_output = ON_EXIT

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

num_retries             = 2
log                     = science1.log
queue
```

The input files `infile-A.txt` and `infile-B.txt` will need to be available on the Execution Point within the pool where the job runs. HTCondor cannot interpret command line arguments, so it cannot know that these command line arguments for this job specify input and output files. The submit command `transfer_input_files` instructs HTCondor to transfer these input files from the machine where the job is submitted to the machine chosen to execute the job. The default operation of HTCondor is to transfer all files created by the job on the EP back to the AP. Therefore, there is no specification of the `outfile.txt` output file.

This example submit description file modifies the commands that direct the transfer of files from AP to EP and back again.

```
should_transfer_files   = IF_NEEDED
when_to_transfer_output = ON_EXIT
```

These values are the HTCondor defaults, so are not needed in this example. They are included to direct attention to the capabilities of HTCondor. The `should_transfer_files` command specifies whether HTCondor should assume the existence of a file system shared by the AP and the EP. Where there is a shared file system, a correctly configured pool of machines will not need to transfer the files from one machine to the other, as both can access the shared file system. Where there is not a shared file system, HTCondor must transfer the files from one machine to the other. The specification `IF_NEEDED` asks HTCondor to use a shared file system when one is detected, but to transfer the files when no shared file system is detected. When files are to be transferred, HTCondor automatically sends the executable as well as a file representing standard input; this file would be specified by the `input` submit command, and it is not relevant to this example. Other files are specified in a comma separated list with `transfer_input_files`, as they are in this example.

When the job completes, all files created by the executable as it ran are transferred back to the AP.

HTCondor assumes that if the job exits of its own accord, with an exit code of zero, that indicates success, and any non-zero exit code is a failure. By default, when the job exits, it will leave the queue. If you would like a job that exits with a non-zero exit code to be restarted some number of times until it does, set num_retries in the submit file like so:

```
num_retries = 2
```

1.4 Expanding the science Job and the Organization of Files

A further example promotes understanding of how HTCondor makes the submission of lots of jobs easy. Assume that the `science.exe` job is to be run 40 times. If the input and output files were exactly the same for each run, then only the last line of the given submit description file changes: from

```
queue
```

to

```
queue 40
```

It is likely that this does not produce the desired outcome, as the output file created, `outfile.txt`, has the same name for each queued instance of the job, and thus this file of results for each run conflicts. Chances are that the input files also must be distinct for each of the 40 separate instances of the job. HTCondor offers the use of a macro that can uniquely name each run's input and output file names. The `$(Process)` macro causes substitution by the process ID from the job identifier. The submit description file for this proposed solution uniquely names the files:

```
# science2.sub -- run 40 instances of science.exe
executable              = science.exe
arguments               =␣
↪"infile-$(Process)A.txt infile-$(Process)B.txt outfile$(Process).txt"

transfer_input_files    = infile-$(Process)A.txt,infile-$(Process)B.txt
should_transfer_files   = IF_NEEDED
when_to_transfer_output = ON_EXIT

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

num_retries             = 2
log                     = science2.log
queue 40
```

The 40 instances of this job will have process ID values that run from 0 to 39. The two input files for process ID 0 are `infile-0A.txt` and `infile-0B.txt`, the ones for process ID 1 will be `infile-1A.txt` and `infile-1B.txt`, and so on, all the way to process ID 39, which will be files `infile-39A.txt` and `infile-39B.txt`. Using this macro for the output file naming of each of the 40 jobs creates `outfile0.txt` for process ID 0; `outfile1.txt` for process ID 1; and so on, to `outfile39.txt` for process ID 39.

This example does not scale well as the number of jobs increases, because the number of files in the same directory becomes unwieldy. Assume now that there will be 100 instances of the `science.exe` job, and each instance has distinct input files, and produces a distinct output file. A recommended organization introduces a unique directory for each job instance. The following submit description file facilitates this organization by specifying the directory with the `initialdir` command. The directories for this example are named `run0`, `run1`, etc. all the way to `run99` for the 100 instances of the following example submit file:

```
# science3.sub -- run 100 instances of science.exe, with
#  unique directories named by the $(Process) macro

executable              = science.exe
arguments               = "infile-A.txt infile-B.txt outfile.txt"

should_transfer_files   = IF_NEEDED
when_to_transfer_output = ON_EXIT

initialdir              = run$(Process)
transfer_input_files    = infile-A.txt,infile-B.txt

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

num_retries             = 2
log                     = science3.log
queue 100
```

The input and output files for each job instance can again be the initial simple names that do not incorporate the `$(Process)` macro. These files are distinct for each run due to their placement within a uniquely named directory. This organization also works well for executables that do not facilitate command line naming of input or output files.

Here is a listing of the files and directories on the AP within this suggested directory structure. The files created due to submitting and running the jobs are shown preceded by an asterisk (*). Only a subset of the 100 directories are shown. Directories are identified using the Linux (and Mac) convention of appending the directory name with a slash character (/).

```
science.exe
science3.sub
run0/
    infile-A.txt
    infile-B.txt
    * outfile.txt
    * science3.log
run1/
    infile-A.txt
    infile-B.txt
    * outfile.txt
    * science3.log
run2/
    infile-A.txt
    infile-B.txt
    * outfile.txt
    * science3.log
```

1.5 Where to Go from Here

- Consider watching our video tutorial for new users.
- Additional tutorials about other aspects of using HTCondor are available in our YouTube channel.
- Slides from past HTCondor Weeks – our annual conference – include the tutorials given there.
- The *Users' Manual* is a good reference.
- If you like what you've seen but want to run more jobs simultaneously, the *administrator's quick start guide* will help you make more of your machines available to run jobs.

CHAPTER

TWO

DOWNLOADING AND INSTALLING

2.1 Windows (as Administrator)

Installation of HTCondor must be done by a user with administrator privileges. We have provided quickstart instructions below to walk you through a single-node HTCondor installation using the HTCondor Windows installer GUI.

For more information about the installation options, or how to use the installer in unattended batch mode, see the complete *Windows Installer* guide.

It is possible to manually install HTCondor on Windows, without the provided MSI program, but we strongly discourage this unless you have a specific need for this approach and have extensive HTCondor experience.

2.1.1 Quickstart Installation Instructions

To download the latest HTCondor Windows Installer:

1. Go to the current channel download site.
2. Click on the second-latest version. (The latest version should always be the under-development version and will only have `daily` builds.)
3. Click on the `release` folder.
4. Click on the file ending in `.msi` (usually the first one).

Start the installer by double clicking on the MSI file once it's downloaded. Then follow the directions below for each option.

If HTCondor is already installed.
: If HTCondor has been previously installed, a dialog box will appear before the installation of HTCondor proceeds. The question asks if you wish to preserve your current HTCondor configuration files. Answer yes or no, as appropriate.

  If you answer yes, your configuration files will not be changed, and you will proceed to the point where the new binaries will be installed.

  If you answer no, then there will be a second question that asks if you want to use answers given during the previous installation as default answers.

STEP 1: License Agreement.
: Agree to the HTCondor license agreement.

STEP 2: HTCondor Pool Configuration.
: Choose the option to create a new pool and enter a name.

STEP 3: This Machine's Roles.
: Check the "submit jobs" box. From the list of execution options, choose "always run jobs".

STEP 4: The Account Domain.
Skip this entry.

STEP 5: E-mail Settings.
Specify the desired email address(es), if any.

STEP 6: Java Settings.
If this entry is already set, accept it. Otherwise, skip it.

STEP 7: Access Permission Settings.
Accept the default values. You can change these later by modifying the configuration files.

STEP 8: VM Universe Setting.
Disable the **vm** universe.

STEP 9: Choose Destination Folder

Accept the default settings.

This should complete the installation process. The installer will have automatically started HTCondor in the background and you do **not** need to restart Windows for HTCondor to work.

Open a command prompt to follow the next set of instructions.

Verifying a Single-Machine Installation

You can easily check to see if the installation procedure succeeded. The following commands should complete without errors, producing output that looks like the corresponding example.

```
condor_status
```

```
Name                OpSys      Arch   State     Activity     LoadAv Mem   Actv

slot1@azaphrael.org  LINUX      X86_64 Unclaimed Benchmarking  0.000 2011  0+00
slot2@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot3@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot4@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00

           Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

X86_64/LINUX    4     0       0         4       0          0        0      0
       Total    4     0       0         4       0          0        0      0
```

```
condor_q
```

```
-- Schedd: azaphrael.org : <184.60.25.78:34585?... @ 11/11/20 14:44:06
OWNER BATCH_NAME      SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS

Total for query: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0 suspended
Total for all users: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0␣
↪suspended
```

If both commands worked, the installation likely succeeded.

Where to Go from Here

- For a brief introduction to running jobs with HTCondor, see the *Users' Quick Start Guide*.
- If you're looking to set up a multi-machine pool, go to the *Administrative Quick Start Guide*.

2.1.2 Setting Up a Whole Pool with Windows

Follow the instructions above through Step 1. Then, customize the installation as follows:

STEP 2: HTCondor Pool Configuration.
Create a new pool only on the machine you've chosen as their central manager. See the *Administrative Quick Start Guide*. Otherwise, choose the option to join an existing pool and enter the name or IP address of the central manager.

STEP 3: This Machine's Roles.
Check the "submit jobs" box to select the submit role, or choose "always run jobs" to select the execute role.

STEP 4: The Account Domain.
Enter the same name on all submit-role machines. This helps ensure that a user can't get more resources by logging in to more than one machine.

STEP 5: E-mail Settings.
Specify the desired email address(es), if any.

STEP 6: Java Settings.
If this entry is already set, accept it. Otherwise, skip it.

Experienced users who know they want to use the **java** universe should instead enter the path to the Java executable on the machine, if it isn't already set, or they want to use a different one.

To disable use of the **java** universe, leave the field blank.

STEP 7: Access Permission Settings.
Machines within the HTCondor pool will need various types of access permission. The three categories of permission that can be set here are read, write, and administrator. The values can be usernames, hostnames or IP address ranges, Wild cards and macros are permitted. It is recommended that you accept the defaults here and change the values later as needed by modifying the HTCondor configuration files.

Read
Read access allows a machine to obtain information about HTCondor such as the status of machines in the pool and the job queues. If all of your HTCondor machines and users are in a single DNS domain or IP Address range, setting this to *.domain an IP address range with wildcards is a good choice. See `ALLOW_READ`

Write
Write access is for submitting jobs to the Schedd. Setting this to * will allow any user that can login to the machine submit jobs. See `ALLOW_WRITE`

Administrator
Administrator access is for starting and stopping the daemons and sending administrative commands such as reconfig and drain. By default the installer will give this permission to the Windows user that runs the installer and to the Windows `Adminstrator` account. See `ALLOW_ADMINISTRATOR`

For more details on these access permissions, and others that can be manually changed in your configuration file, please see the section titled Setting Up Security in HTCondor in the *Authorization* section.

STEP 8: VM Universe Setting.
Disable the **vm** universe.

Experienced users with VMWare and Perl already installed may enable the **vm** universe.

STEP 9: Choose Destination Folder

Experienced users may change the default installation path (`c:\Condor`), but we don't recommend doing so. Certain jobs may not run if the installation path has a space in it.

2.2 Linux (as root)

For ease of installation on Linux, we provide a script that will automatically download, install and start HTCondor.

2.2.1 Quickstart Installation Instructions

Warning:

- RedHat systems must be attached to a subscription.
- Debian and Ubuntu containers don't come with `curl` installed, so run the following first.

```
apt-get update && apt-get install -y curl
```

The command below shows how to download the script and run it immediately; if you would like to inspect it first, see *Inspecting the Script*. The default behavior will create a complete HTCondor pool with its multiple roles on one computer, referred to in this manual as a "minicondor." Experienced users who are making an HTCondor pool out of multiple machines should add a flag to select the desired role; see the *Administrative Quick Start Guide* for more details.

```
curl -fsSL https://get.htcondor.org | sudo /bin/bash -s -- --no-dry-run
```

If you see an error like `bash: sudo: command not found`, try re-running the command above without the `sudo`.

Inspecting the Script

If you would like to inspect the script before you running it on your system as root, you can:

- read the script;
- compare the script to the versions in our GitHub repository;
- or run the script as user `nobody`, dropping the `--no-dry-run` flag. This will cause the script to print out what it would do if run for real. You can then inspect the output and copy-and-paste it to perform the installation.

Verifying a Single-Machine Installation

You can easily check to see if the installation procedure succeeded. The following commands should complete without errors, producing output that looks like the corresponding example.

```
condor_status
```

```
Name                OpSys      Arch   State     Activity     LoadAv Mem   Actv

slot1@azaphrael.org  LINUX      X86_64 Unclaimed Benchmarking  0.000 2011  0+00
slot2@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot3@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot4@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00

           Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

X86_64/LINUX    4     0       0         4       0          0        0      0
       Total    4     0       0         4       0          0        0      0
```

```
condor_q
```

```
-- Schedd: azaphrael.org : <184.60.25.78:34585?... @ 11/11/20 14:44:06
OWNER BATCH_NAME      SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS

Total for query: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0 suspended
Total for all users: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0␣
↪suspended
```

If both commands worked, the installation likely succeeded.

Where to Go from Here

- For a brief introduction to running jobs with HTCondor, see the *Users' Quick Start Guide*.
- If you're looking to set up a multi-machine pool, go to the *Administrative Quick Start Guide*.

2.2.2 Setting Up a Whole Pool

The details of using this installation procedure to create a multi-machine HTCondor pool are described in the admin quick-start guide: *Administrative Quick Start Guide*.

2.3 Linux (from our repositories)

If you're not already familiar with HTCondor, we recommend you follow our *instructions* for your first installation.

If you're looking to automate the installation of HTCondor using your existing toolchain, the latest information is embedded in the output of the script run as part of the *instructions*. This script can be run as a normal user (or `nobody`), so we recommend this approach.

Otherwise, this page contains information about the RPM and deb repositories we offer. These repositories will almost always have more recent releases than the distributions.

2.3.1 RPM-based Distributions

We support several RPM-based platforms: Enterprise Linux 7, including Red Hat, CentOS, and Scientific Linux; Enterprise Linux 8, including Red Hat, CentOS Stream, Alma Linux, and Rocky Linux; Enterprise Linux 9, including Red Hat, CentOS Stream, Alma Linux, and Rocky Linux; openSUSE LEAP 15 including SUSE Linux Enterprise Server (SLES) 15. Binaries are available for x86_64 for all these platforms. For Enterprise Linux 8, HTCondor also supports ARM ("aarch64") and Power ("ppc64le"). For Enterprise Linux 9, HTCondor also supports ARM ("aarch64").

Repository packages are available for each platform:

- Amazon Linux 2023
- Enterprise Linux 7
- Enterprise Linux 8
- Enterprise Linux 9
- openSUSE LEAP 15

Except for Amazon Linux, the HTCondor packages on these platforms depend on the corresponding version of EPEL.

Additionally, the following repositories are required for specific platforms:

- On RedHat 7, `rhel-*-optional-rpms`, `rhel-*-extras-rpms`, and `rhel-ha-for-rhel-*-server-rpms`.
- On RedHat 8, `codeready-builder-for-rhel-8-${ARCH}-rpms`.
- On CentOS 8, `powertools` (or `PowerTools`).
- On RedHat 9, `crb`.

2.3.2 deb-based Distributions

We support four deb-based platforms: Debian 11 (Bullseye) and Debian 12 (Bookworm); and Ubuntu 20.04 (Focal Fossa) and 22.04 (Jammy Jellyfish). Binaries are available for x86_64 for all these platforms. For Unbuntu 20.04 (Focal Fossa) HTCondor also supports Power PC (ppc64el). These repositories also include the source packages.

Debian 11, and 12

Add our Debian signing key with `apt-key add` before adding the repositories below.

- Debian 11: `deb https://research.cs.wisc.edu/htcondor/repo/debian/23.0 bullseye main`
- Debian 12: `deb https://research.cs.wisc.edu/htcondor/repo/debian/23.0 bookworm main`

Ubuntu 20.04, and 22.04

Add our Ubuntu signing key with `apt-key add` before adding the repositories below.

- Ubuntu 20.04: `deb https://research.cs.wisc.edu/htcondor/repo/ubuntu/23.0 focal main`
- Ubuntu 22.04: `deb https://research.cs.wisc.edu/htcondor/repo/ubuntu/23.0 jammy main`

2.4 Linux or macOS (as user)

Installing HTCondor on Linux or macOS as a normal user is a multi-step process. Note that a user-install of HTCondor is always self-contained on a single machine; if you want to create a multi-machine HTCondor pool, you will need to have administrative privileges on the relevant machines and follow the instructions here: *Administrative Quick Start Guide*.

2.4.1 Download

The first step is to download HTCondor for your platform. If you know which platform you're using, that HTCondor supports it, and which version you want, you can download the corresponding file from our website; otherwise, we recommend using our download script, as follows.

```
cd
curl -fsSL https://get.htcondor.org | /bin/bash -s -- --download
```

On macOS, If you use a web browser to download a tarball from our web site, then the OS will mark the file as quarantined. All binaries extracted from the tarball will be similarly marked. The OS will refuse to run any binaries that are quarantined. You can remove the quarantine marking from the tarball before extracting, like so:

```
xattr -d com.apple.quarantine condor-10.7.1-x86_64_macOS13-stripped.tar.gz
```

2.4.2 Install

Unpack the tarball and rename the resulting directory:

```
tar -x -f condor.tar.gz
mv condor-*stripped condor
```

You won't need `condor.tar.gz` again, so you can remove it now if you wish.

2.4.3 Configure

```
cd condor
./bin/make-personal-from-tarball
```

2.4.4 Using HTCondor

You'll need to run the following command now, and every time you log in:

```
. ~/condor/condor.sh
```

Then to start HTCondor (if the machine has rebooted since you last logged in):

```
condor_master
```

It will finish silently after starting up, if everything went well.

Verifying a Single-Machine Installation

You can easily check to see if the installation procedure succeeded. The following commands should complete without errors, producing output that looks like the corresponding example.

```
condor_status
```

```
Name                OpSys      Arch   State     Activity     LoadAv Mem   Actv

slot1@azaphrael.org  LINUX      X86_64 Unclaimed Benchmarking  0.000 2011  0+00
slot2@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot3@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot4@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00

           Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

X86_64/LINUX    4     0       0         4       0          0        0      0
       Total    4     0       0         4       0          0        0      0
```

```
condor_q
```

```
-- Schedd: azaphrael.org : <184.60.25.78:34585?... @ 11/11/20 14:44:06
OWNER BATCH_NAME      SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS

Total for query: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0 suspended
Total for all users: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0␣
↪suspended
```

If both commands worked, the installation likely succeeded.

Where to Go from Here

- For a brief introduction to running jobs with HTCondor, see the *Users' Quick Start Guide*.
- If you're looking to set up a multi-machine pool, go to the *Administrative Quick Start Guide*.

2.5 macOS (as root)

Installing HTCondor on macOS as root user is a multi-step process. For a multi-machine HTCondor pool, information about the roles each machine will play can be found here: *Administrative Quick Start Guide*. Note that the `get_htcondor` tool cannot perform the installation steps on macOS at present. You must follow the instructions below.

Note that all of the following commands must be run as root, except for downloading and extracting the tarball.

2.5.1 The condor Service Account

The first step is to create a service account under which the HTCondor daemons will run. The commands that specify a `PrimaryGroupID` or `UniqueID` may fail with an error that includes `eDSRecordAlreadyExists`. If that occurs, you will have to retry the command with a different id number (other than `300`).

```
dscl . -create /Groups/condor
dscl . -create /Groups/condor PrimaryGroupID 300
dscl . -create /Groups/condor RealName 'Condor Group'
dscl . -create /Groups/condor passwd '*'
dscl . -create /Users/condor
dscl . -create /Users/condor UniqueID 300
dscl . -create /Users/condor passwd '*'
dscl . -create /Users/condor PrimaryGroupID 300
dscl . -create /Users/condor UserShell /usr/bin/false
dscl . -create /Users/condor RealName 'Condor User'
dscl . -create /Users/condor NFSHomeDirectory /var/empty
```

2.5.2 Download

The next step is to download HTCondor. If you want to select a specific version of HTCondor, you can download the corresponding file from our website. Otherwise, we recommend using our download script, as follows.

```
cd
curl -fsSL https://get.htcondor.org | /bin/bash -s -- --download
```

If you use a web browser to download a tarball from our web site, then the OS will mark the file as quarantined. All binaries extracted from the tarball will be similarly marked. The OS will refuse to run any binaries that are quarantined. You can remove the quarantine marking from the tarball before extracting it, like so:

```
xattr -d com.apple.quarantine condor-10.7.1-x86_64_macOS13-stripped.tar.gz
```

2.5.3 Install

Unpack the tarball.

```
mkdir /usr/local/condor
tar -x -C /usr/local/condor --strip-components 1 -f condor.tar.gz
```

You won't need `condor.tar.gz` again, so you can remove it now if you wish.

Set up the log directory and default configuration files.

```
cd /usr/local/condor
mkdir -p local/log
mkdir -p local/config.d
cp etc/examples/condor_config etc/condor_config
cp etc/examples/00-htcondor-9.0.config local/config.d
```

If you are setting up a single-machine pool, then run the following command to finish the configuration.

```
cp etc/examples/00-minicondor local/config.d
```

If you are setting up part of a multi-machine pool, then you'll have to make some other configuration changes, which we don't cover here.

Next, fix up the permissions of the the installed files.

```
chown -R root:wheel /usr/local/condor
chown -R condor:condor /usr/local/condor/local/log
```

Finally, make the configuration file available at one of the well-known locations for the tools to find.

```
mkdir -p /etc/condor
ln -s /usr/local/condor/etc/condor_config /etc/condor
```

2.5.4 Start the Daemons

Now, register HTCondor has a service managed by launchd and start up the daemons.

```
cp /usr/local/condor/etc/examples/condor.plist /Library/LaunchDaemons
launchctl load /Library/LaunchDaemons/condor.plist
launchctl start condor
```

2.5.5 Using HTCondor

You'll want to add the HTCondor `bin` and `sbin` directories to your `PATH` environment variable.

```
export PATH=$PATH:/usr/local/condor/bin:/usr/local/condor/sbin
```

If you want to use the Python bindings for HTCondor, you'll want to add them to your PYTHONPATH.

```
export PYTHONPATH="/usr/local/condor/lib/python3${PYTHONPATH+":"}${PYTHONPATH-}"
```

Verifying a Single-Machine Installation

You can easily check to see if the installation procedure succeeded. The following commands should complete without errors, producing output that looks like the corresponding example.

```
condor_status
```

```
Name                OpSys      Arch   State     Activity     LoadAv Mem   Actv

slot1@azaphrael.org  LINUX      X86_64 Unclaimed Benchmarking  0.000 2011  0+00
slot2@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot3@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00
slot4@azaphrael.org  LINUX      X86_64 Unclaimed Idle          0.000 2011  0+00

           Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

X86_64/LINUX     4     0       0         4       0          0        0      0
       Total     4     0       0         4       0          0        0      0
```

```
condor_q
```

```
-- Schedd: azaphrael.org : <184.60.25.78:34585?... @ 11/11/20 14:44:06
OWNER BATCH_NAME      SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS

Total for query: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0 suspended
Total for all users: 0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0␣
↪suspended
```

If both commands worked, the installation likely succeeded.

Where to Go from Here

- For a brief introduction to running jobs with HTCondor, see the *Users' Quick Start Guide*.
- If you're looking to set up a multi-machine pool, go to the *Administrative Quick Start Guide*.

2.6 Docker Images

HTCondor provides images on Docker Hub.

2.6.1 Quickstart Instructions

If you're just getting started with HTCondor, use `htcondor/mini`, a stand-alone HTCondor configuration. The following command will work on most systems with Docker installed:

```
docker run -it htcondor/mini
```

From here, you can proceed to the *Users' Quick Start Guide*.

2.6.2 Setting Up a Whole Pool with Docker

If you're looking to set up a whole pool, the following images correspond to the three required roles. See the *Administrative Quick Start Guide* for more information about the roles and how to configure these images to work together.

- `htcondor/cm`, an image configured as a central manager
- `htcondor/execute`, an image configured as an execute node
- `htcondor/submit`, an image configured as a submit node

All images include the latest version of HTCondor. If you want to use the latest LTS version, use the docker tag `lts`.

2.7 Administrative Quick Start Guide

This guide does not contain step-by-step instructions for *getting HTCondor*. Rather, it is a guide to joining multiple machines into a single pool of computational resources for use by HTCondor jobs.

This guide begins by briefly describing the three roles required by every HTCondor pool, as well as the resources and networking required by each of those roles. This information will enable you to choose which machine(s) will perform which role(s). This guide also includes instructions on how to use the `get_htcondor` tool to install and configure Linux (or Mac) machines to perform each of the roles.

If you're curious, using Windows machines, or you want to automate the configuration of their pool using a tool like Puppet, the *last section* of this guide briefly describes what the `get_htcondor` tool does and provides a link to the rest of the details.

Single-machine Installations

If you just finished installing a single-machine ("mini") HTCondor using `get_htcondor`, you can just run `get_htcondor` again (and follow its instructions) to reconfigure the machine to be one of these three roles; this may destroy any other configuration changes you've made.

We don't recommend trying to add a machine configured as a "mini" HTCondor to the pool, or trying to add execute machines to an existing "mini" HTCondor pool. We also don't recommend creating an entire pool out of unprivileged installations.

2.7.1 The Three Roles

Even a single-machine installation of HTCondor performs all three roles.

The Execute Role

The most common reason for adding a machine to an HTCondor pool is to make another machine execute HTCondor jobs; the first major role, therefore, is the execute role. This role is responsible for the technical aspects of actually running, monitoring, and managing the job's executable; transferring the job's input and output; and advertising, monitoring, and managing the resources of the execute machine. HTCondor can manage pools containing tens of thousands of execute machines, so this is by far the most common role.

The execute role itself uses very few resources, so almost any machine can contribute to a pool. The execute role can run on a machine with only outbound network connectivity, but being able to accept inbound connections from the machine(s) performing the submit role will simplify setup and reduce overhead. The execute machine does not need to allow user access, or even share user IDs with other machines in the pool (although this may be very convenient, especially on Windows).

The Submit Role

We'll discuss what "advertising" a machine's resources means in the next section, but the execute role leaves an obvious question unanswered: where do the jobs come from? The answer is the submit role. This role is responsible for accepting, monitoring, managing, and scheduling jobs on its assigned resources; transferring the input and output of jobs; and requesting and accepting resource assignments. (A "resource" is some reserved fraction of an execute machine.) HTCondor allows arbitrarily many submit roles in a pool, but for administrative convenience, most pools only have one, or a small number, of machines acting in the submit role.

A submit-role machine requires a bit under a megabyte of RAM for each running job, and its ability to transfer data to and from the execute-role machines may become a performance bottleneck. We typically recommend adding another access point for every twenty thousand simultaneously running jobs. A access point must have outbound network connectivity, but a submit machine without inbound network connectivity can't use execute-role machines without inbound network connectivity. As execute machines are more numerous, access points typically allow inbound connections. Although you may allow users to submit jobs over the network, we recommend allowing users SSH access to the access point.

The Central Manager Role

Only one machine in each HTCondor pool can perform this role (barring certain high-availability configurations, where only one machine can perform this role at a time). A central manager matches resource requests – generated by the submit role based on its jobs – with the resources described by the execute machines. We refer to sending these (automatically-generated) descriptions to the central manager as "advertising" because it's the primary way execute machines get jobs to run.

A central manager must accept connections from each execute machine and each access point in a pool. However, users should never need access to the central manager. Every machine in the pool updates the central manager every few minutes, and it answers both system and user queries about the status of the pool's resources, so a fast network is important. For very large pools, memory may become a limiting factor.

2.7.2 Assigning Roles to Machines

The easiest way to assign a role to a machine is when you initially *get HTCondor*. You'll need to supply the same password for each machine in the same pool; sharing that secret is how the machines recognize each other as members of the same pool, and connections between machines are encrypted with it. (HTCondor uses port 9618 to communicate, so make sure that the machines in your pool accept TCP connections on that port from each other.) In the command lines below, replace `$htcondor_password` with the password you want to use. In addition to the password, you must specify the name of the central manager, which may be a host name (which must resolve on all machines in the pool) or an IP address. In the command lines below, replace `$central_manager_name` with the host name or IP address you want to use.

When you *get HTCondor*, start with the central manager, then add the access point(s), and then add the execute machine(s). You may not have `sudo` installed; you may omit it from the command lines below if you run them as root.

Central Manager

```
curl -fsSL https://get.htcondor.org | sudo GET_HTCONDOR_PASSWORD="$htcondor_password" /
↪bin/bash -s -- --no-dry-run --central-manager $central_manager_name
```

Submit

```
curl -fsSL https://get.htcondor.org | sudo GET_HTCONDOR_PASSWORD="$htcondor_password" /
↪bin/bash -s -- --no-dry-run --submit $central_manager_name
```

Execute

```
curl -fsSL https://get.htcondor.org | sudo GET_HTCONDOR_PASSWORD="$htcondor_password" /
↪bin/bash -s -- --no-dry-run --execute $central_manager_name
```

At this point, users logged in on the access point should be able to see execute machines in the pool (using `condor_status`), submit jobs (using `condor_submit`), and see them run (using `condor_q`).

Creating a Multi-Machine Pool using Windows or Containers

If you are creating a multi-machine HTCondor pool on Windows computers or using containerization, please see the "Setting Up a Whole Pool" section of the relevant installation guide:

- *Setting Up a Whole Pool with Windows*
- *Setting Up a Whole Pool with Docker*

2.7.3 Where to Go from Here

There are two major directions you can go from here, but before we discuss them, a warning.

Making Configuration Changes

HTCondor configuration files should generally be owned by root (or Administrator, on Windows), but readable by all users. We recommend that you don't make changes to the configuration files established by the installation procedure; this avoids conflicts between your changes and any changes we may have to make to the base configuration in future updates. Instead, you should add (or edit) files in the configuration directory; its location can be determined on a given machine by running `condor_config_val LOCAL_CONFIG_DIR` there. HTCondor will process files in this directory in lexicographic order, so we recommend naming files `##-name.config` so that, for example, a setting in `00-base.config` will be overridden by a setting in `99-specific.config`.

Enabling Features

Some features of HTCondor, for one reason or another, aren't (or can't be) enabled by default. Areas of potentially general interest include:

- *Setting Up for Special Environments* (particularly *Enabling the Fetching and Use of OAuth2 Credentials* and *Cgroup-Based Process Tracking*),
- *Setting Up the Docker Universe*
- *Apptainer/Singularity Support*

Implementing Policies

Although your HTCondor pool should be fully functional at this point, it may not be behaving precisely as you wish, particularly with respect to resource allocation. You can tune how HTCondor allocates resources to users, or groups of users, using the user priority and group quota systems, described in *User Priorities and Negotiation*. You can enforce machine-specific policies – for instance, preferring GPU jobs on machines with GPUs – using the options described in *Policy Configuration for Execution Points and for Access Points*.

Further Reading

- It may be helpful to at least skim the *Users' Manual* to get an idea of what your users might want or expect, particularly the sections on *DAGMan Introduction*, *Choosing an HTCondor Universe*, and *Self-Checkpointing Applications*.
- Understanding *HTCondor's ClassAd Mechanism* is essential for many administrative tasks.
- The rest of the *Administrators' Manual*, particularly the section on *Monitoring*.
- Slides from past HTCondor Weeks – our annual conference – include a number of tutorials and talks on administrative topics, including monitoring and examples of policies and their implementations.

2.7.4 What `get_htcondor` Does to Configure a Role

The configuration files generated by `get_htcondor` are very similar, and only two lines long:

- set the HTCondor configuration variable `CONDOR_HOST` to the name (or IP address) of your central manager;
- add the appropriate metaknob: `use role : get_htcondor_central_manager`, `use role : get_htcondor_submit`, or `use role : get_htcondor_execute`.

Putting all of the pool-independent configuration into the metaknobs allows us to change the metaknobs to fix problems or work with later versions of HTCondor as you upgrade.

The `get_htcondor` *documentation* describes what the configuration script does and how to determine the exact details.

These instructions show how to create a complete HTCondor installation with all of its components on a single computer, so that you can test HTCondor and explore its features. We recommend that new users start with the *first set of instructions* here and then continue with the *Users' Quick Start Guide*; that link will appear again at the end of these instructions.

If you know how to use Docker, you may find it easier to start with the `htcondor/mini` image; see the *Docker Images* entry. If you're familiar with cloud computing, you may also get HTCondor *in the cloud*.

Installing HTCondor on a Cluster

Experienced users who want to make an HTCondor pool out of multiple machines should follow the *Administrative Quick Start Guide*. If you're new to HTCondor administration, you may want to read the *Administrators' Manual*.

Installing HTCondor on a Single Machine with Administrative Privileges

If you have administrative privileges on your machine, choose the instructions corresponding to your operating system:

- *Windows*.
- *Linux*. HTCondor supports Amazon Linux 2023; Enterprise Linux 7 including Red Hat, CentOS, and Scientific Linux 7; Enterprise Linux 8 including Red Hat, CentOS Stream, Alma Linux, and Rocky Linux; Enterprise Linux 9 including Red Hat, CentOS Stream, Alma Linux, and Rocky Linux; openSUSE LEAP 15 including SUSE Linux Enterprise Server 15; Debian 11 and 12; and Ubuntu 20.04 and 22.04.
- *macOS*. HTCondor supports macOS 10.15 and later.

Hand-Installation of HTCondor on a Single Machine with User Privileges

If you don't have administrative privileges on your machine, you can still install HTCondor. An unprivileged installation isn't able to effectively limit the resource usage of the jobs it runs, but since it only works for the user who installed it, at least you know who to blame for misbehaving jobs.

- *Linux*. HTCondor supports Amazon Linux 2023; Enterprise Linux 7 including Red Hat, CentOS, and Scientific Linux 7; Enterprise Linux 8 including Red Hat, CentOS Stream, Alma Linux, and Rocky Linux; Enterprise Linux 9 including Red Hat, CentOS Stream, Alma Linux, and Rocky Linux; openSUSE LEAP 15 including SUSE Linux Enterprise Server 15; Debian 11 and 12; and Ubuntu 20.04 and 22.04.
- *macOS*. HTCondor supports macOS 10.15 and later.

Docker Images

HTCondor is also available on Docker Hub.

If you're new to HTCondor, the `htcondor/mini` image is equivalent to following any of the instructions above, and once you've started the container, you can proceed directly to the *Users' Quick Start Guide* and learn how to run jobs.

For other options, see our *docker image list*.

Kubernetes

You can deploy a complete HTCondor pool with the following command:

```
kubectl apply -f https://github.com/htcondor/htcondor/blob/latest/build/docker/k8s/pool.
↪yaml
```

If you're new to HTCondor, you can proceed directly to the *Users' Quick Start Guide* after logging in to the `submit` pod.

In the Cloud

Although you can use our Docker images (or Kubernetes support) in the cloud, HTCondor also supports cloud-native distribution.

- For Amazon Web Services, we offer a minicondor image preconfigured for use with *condor_annex*, which allows to easily add cloud resources to your pool.
- The *Google Cloud Marketplace Entry* lets you construct an entire HTCondor pool that scales automatically to run submitted jobs. If you're new to HTCondor, you can proceed to the *Users' Quick Start Guide* immediately after following those instructions.
- We also have documentation on creating a *HTCondor in the Cloud* by hand.

CHAPTER

THREE

OVERVIEW

3.1 High-Throughput Computing (HTC) and its Requirements

The quality of many projects is dependent upon the quantity of computing cycles available. Many problems require years of computation to solve. These problems demand a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High-Throughput Computing (HTC) environment. In contrast, High Performance Computing (HPC) environments deliver a tremendous amount of compute power over a short period of time. HPC environments are often measured in terms of Floating point Operations Per Second (FLOPS). A growing community is not concerned about operations per second, but operations per month or per year (FLOPY). They are more interested in how many jobs they can complete over a long period of time instead of how fast an individual job can finish.

The key to HTC is to efficiently harness the use of all available resources. Years ago, the engineering and scientific community relied on a large, centralized mainframe or a supercomputer to do computational work. A large number of individuals and groups needed to pool their financial resources to afford such a machine. Users had to wait for their turn on the mainframe, and they had a limited amount of time allocated. While this environment was inconvenient for users, the utilization of the mainframe was high; it was busy nearly all the time.

As computers became smaller, faster, and cheaper, users moved away from centralized mainframes. Today, most organizations own or lease many different kinds of computing resources in many places. Racks of departmental servers, desktop machines, leased resources from the Cloud, allocations from national supercomputer centers are all examples of these resources. This is an environment of distributed ownership, where individuals throughout an organization own their own resources. The total computational power of the institution as a whole may be enormous, but because of distributed ownership, groups have not been able to capitalize on the aggregate institutional computing power. And, while distributed ownership is more convenient for the users, the utilization of the computing power is lower. Many machines sit idle for very long periods of time while their owners have no work for the machines to do.

3.2 HTCondor's Power

HTCondor is a software system that creates a High-Throughput Computing (HTC) environment. It effectively uses the computing power of machines connected over a network, be they a single cluster, a set of clusters on a campus, cloud resources either stand alone or temporarily joined to a local cluster, or international grids. Power comes from the ability to effectively harness shared resources with distributed ownership.

A user submits jobs to HTCondor. HTCondor finds available machines and begins running the jobs there. HTCondor has the capability to detect that a machine running a job is no longer available (perhaps the machine crashed, or maybe it prefers to run another job). HTCondor will automatically restart the job on another machine without intervention from the user.

HTCondor is useful when a job must be run many (thousands of) times, perhaps with hundreds of different data sets. With one command, all of the jobs are submitted to HTCondor. Depending upon the number of machines in the HTCondor pool, hundreds of otherwise idle machines can be running the jobs at any given moment.

HTCondor does not require an account (login) on machines where it runs a job. HTCondor can do this because of its file transfer and split execution mechanisms.

HTCondor provides powerful resource management by match-making resource owners with resource consumers. This is the cornerstone of a successful HTC environment. Other compute cluster resource management systems attach properties to the job queues themselves, resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands. HTCondor implements ClassAds, a clean design that simplifies the user's submission of jobs.

ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the HTCondor pool advertise their resource properties, both static and dynamic, such as available RAM memory, CPU type, CPU speed, virtual memory size, physical location, and current load average, in a resource offer ad. A user specifies a resource request ad when submitting a job. The request defines both the required and a desired set of properties of the resource to run the job. HTCondor acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, HTCondor also considers several layers of priority values: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and the desire of machines in the pool to accept certain types of ads over others.

3.3 Exceptional Features

Reliability
: An HTCondor job "is like money in the bank". After successful submission, HTCondor owns the job, and will run it to completion, even if the submit machine or execute machine crash, and require HTCondor to restart the job elsewhere.

Scalability
: An HTCondor pool is horizontally scalable to hundreds of thousands of execute cores running a similar number of running jobs, and an even larger number of idle jobs. HTCondor is also scalable down to run an entire pool on a single machine, and many scales between these two extremes.

Security
: HTCondor, by default, uses strong authentication and encryption on the wire. The HTCondor worker node scratch directories can be encrypted, so that if a node is stolen or broken into, scratch files are unreadable.

Parallelization without Reimplementation or Redesign
: HTCondor is able to run most programs which researchers can run on their laptop or their desktop, in any programming language, such as C, Fortran, Python, Julia, Matlab, R or others, without changing the code. HTCondor will do the work of running your code as parallel jobs, so it is not necessary to implement parallelism in your code.

Portability and Heterogeneity
: HTCondor runs on most Linux distributions and on Windows. A single HTCondor pool can support machines of different OSes. Worker nodes need not be identically provisioned – HTCondor detects the memory, CPU cores, GPUs and other machine resources available on a machine, and only runs jobs that match their needs to the machine's capabilities.

Pools of Machines can be Joined Together
: Flocking allows jobs submitted from one pool of HTCondor machines to execute on another authorized pool.

Jobs Can Be Ordered
: A set of jobs where the output of one or more jobs becomes the input of one or more other jobs, can

be defined, such that HTCondor will run the jobs in the proper order, and organize the inputs and outputs properly. This is accomplished with a directed acyclic graph, where each job is a node in the graph.

HTCondor Can Use Remote Resources, from a Cloud, a Supercomputer Allocation, or a Grid
Glidein allows jobs submitted to HTCondor to be executed on machines in remote pools in various locations worldwide. These remote pools can be in one or more clouds, in an allocation on a HPC site, in a different HTCondor pool or on a compute grid.

Sensitive to the Desires of Machine Owners
The owner of a machine has complete priority over the use of the machine. HTCondor lets the machine's owner decide if and how HTCondor uses the machine. When HTCondor relinquishes the machine, it cleans up any files created by the jobs that ran on the system.

Flexible Policy Mechanisms
HTCondor allows users to specify very flexible policies for how they want jobs to be run. Conversely, it independently allows the owners of machines to specify very flexible policies about what jobs (if any) should be run on their machines. Together, HTCondor merges and adjudicates these policy requests into one coherent system.

The ClassAd mechanism in HTCondor provides an expressive framework for matchmaking resource requests with resource offers. Users can easily request both job requirements and job desires. For example, a user can require that their job must be started on a machine with a certain amount of memory, but should there be multiple available machines that meet that criteria, to select the one with the most memory.

3.4 Availability

HTCondor is available for download from the URL http://htcondor.org/downloads/.

For more platform-specific information about HTCondor's support for various operating systems, see the *Platform-Specific Information* chapter.

3.5 Contributions and Acknowledgments

The quality of the HTCondor project is enhanced by the contributions of external organizations. We gratefully acknowledge the following contributions.

- The GOZAL Project from the Computer Science Department of the Technion Israel Institute of Technology (http://www.technion.ac.il/), for their enhancements for HTCondor's High Availability. The *condor_had* daemon allows one of multiple machines to function as the central manager for a HTCondor pool. Therefore, if an acting central manager fails, another can take its place.
- Micron Corporation (http://www.micron.com/) for the MSI-based installer for HTCondor on Windows.
- Paradyn Project (http://www.paradyn.org/) and the Universitat Autònoma de Barcelona (http://www.caos.uab.es/) for work on the Tool Daemon Protocol (TDP).

The HTCondor project wishes to acknowledge the following:

- This material is based upon work supported by the National Science Foundation under Grant Numbers MCS-8105904, OCI-0437810, and OCI-0850745. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

3.6 Support, Downloads and Bug Reporting

The latest software releases, publications/papers regarding HTCondor and other High-Throughput Computing research can be found at the official web site for HTCondor at http://htcondor.org/.

3.6.1 Downloads

A list of recent HTCondor software releases is available on our downloads page: https://htcondor.org/downloads.

Selecting a release channel will lead you to the *Downloading and Installing* section of the HTCondor Manual, which describes how to download and install HTCondor.

3.6.2 Support

Mailing Lists

Our users support each other on a community unmoderated mailing list (htcondor-users@cs.wisc.edu) targeted at solving problems with HTCondor. HTCondor team members attempt to monitor traffic to htcondor-users, responding as they can. Follow the instructions at http://htcondor.org/mail-lists. If you have a question or potential bug report for HTCondor that can be asked on a public mailing list, this is the first place to go.

In addition, there is a very low-volume e-mail list at htcondor-world@cs.wisc.edu. We use this e-mail list to announce new releases of HTCondor and other major HTCondor-related news items. To subscribe or unsubscribe from the list, follow the instructions at http://htcondor.org/mail-lists. The HTCondor World e-mail list group is moderated, and only major announcements of wide interest are distributed.

Email Support

You can reach the HTCondor Team directly. The HTCondor Team is composed of the developers and administrators of HTCondor at the University of Wisconsin-Madison. HTCondor questions, bug reports, comments, pleas for help, and requests for commercial contract consultation or support are all welcome; send e-mail to htcondor-admin@cs.wisc.edu. Please include your name, organization, and email in your message. If you are having trouble with HTCondor, please help us troubleshoot by including as much pertinent information as you can, including snippets of HTCondor log files, and the version of HTCondor you are running.

Finally, we have several options for users who require additional support for HTCondor beyond the free support listed above. All details are available on our website: https://htcondor.org/htcondor-support/

3.6.3 Reporting Bugs

We recommend you use the mailing lists or email support listed above to report bugs. Please provide as much information as possible: detailed information about the problem, relevant log files, and steps on how to reproduce it. If it's a new issue that our team was not aware of, we'll create a new ticket in our system.

Ticketing System

Experienced HTCondor users can also request a user account that will allow them to create tickets directly in our system:

https://htcondor-wiki.cs.wisc.edu/index.cgi/rptview?rn=4

To get an account, send an email to htcondor-admin@cs.wisc.edu explaining why you want it and how you intend to use it. These are typically reserved for known collaborators with direct contact to the HTCondor team.

CHAPTER

FOUR

USERS' MANUAL

4.1 Welcome and Introduction to HTCondor

The HTCondor software system is developed by the Center for High Throughput Computing at the University of Wisconsin-Madison (UW-Madison), and was first installed as a production system in the UW-Madison Computer Sciences department in the 1990s. HTCondor pools have since served as a major source of computing cycles to thousands of campuses, labs, organizations and commercial entities. For many, it has revolutionized the role computing plays in their research. Increasing computing throughput by several orders of magnitude may not merely deliver the same results faster, but may enable qualitatively different avenues of research.

HTCondor is a specialized batch system for managing compute-intensive jobs. HTCondor provides a queuing mechanism, scheduling policy, priority scheme, and resource classifications. Users submit their compute jobs to HTCondor, HTCondor puts the jobs in a queue, runs them, and then informs the user as to the result.

Batch systems normally operate only with dedicated machines. Often termed worker nodes, these dedicated machines are typically owned by one group and dedicated to the sole purpose of running compute jobs. HTCondor can schedule jobs on dedicated machines. But unlike traditional batch systems, HTCondor is also designed to run jobs on machines shared and used by other systems or people. By running on these shared resources, HTCondor can effectively harness all machines throughout a campus. This is important because often an organization has more latent, idle computers than any single department or group otherwise has access to.

4.2 Running a Job: the Steps To Take

Here are the basic steps to run a job with HTCondor.

Work Decomposition

Typically, users want High Throughput computing systems when they have more work than can reasonably run on a single machine. Therefore, the computation must run concurrently on multiple machines. HTCondor itself does not help with breaking up a large amount of work to run independently on many machines. In many cases, such as Monte Carlo simulations, this may be trivial to do. In other situations, the code must be refactored or code loops may need to be broken into separate work steps in order to be suitable for High Throughput computing. Work must be broken down into a set of *jobs* whose runtime is neither too short nor too long. HTCondor is most efficient when running jobs whose runtime is measured in minutes or hours. There is overhead in scheduling each job, which is why very short jobs (measured in seconds) do not work well. On the other hand, if a job takes many days to run, there is the threat of losing work in progress should the job or the server it runs on crashes.

Prepare the job for batch execution.
To run under HTCondor a job must be able to run as a background batch job. HTCondor runs the program unattended and in the background. A program that runs in the background will not be able to do interactive input and output. Create any needed input files for the program. Make certain the program will run correctly with these files.

Create a description file.
A submit description file controls the all details of a job submission. This text file tells HTCondor everything it needs to know to run the job on a remote machine, e.g. how much memory and how many cpu cores are needed, what input files the job needs, and other aspects of machine the job might need.

Write a submit description file to go with the job, using the examples provided in the *Submitting a Job* section for guidance. There are many possible options that can be set in a submit file, but most submit files only use a few. The complete list of submit file options is in *condor_submit*.

Submit the Job.
Submit the program to HTCondor with the *condor_submit* command. HTCondor will assign the job a unique Cluster and Proc identifier as integers separated by a dot. You use this Cluster and Proc id to manage the job later.

Manage the Job.
After submission, HTCondor manages the job during its lifetime. You can monitor the job's progress with the *condor_q*. On some platforms, you can ssh to a running job with the *condor_ssh_to_job* command, and inspect the job as it runs.

HTCondor can write into a log file describing changes to the state of your job – when it starts executing, when it uses more resources, when it completes, or when it is preempted from a machine. You can remove a running or idle job from the queue with *condor_rm*.

Examine the results of a finished job.
When your program completes, HTCondor will tell you (by e-mail, if preferred) the exit status of your program and various statistics about its performances, including time used and I/O performed. If you are using a log file for the job, the exit status will be recorded in there. Output files will be transferred back to the submitting machine, if a shared filesystem is not used. After the job completes, it will not be visible to the *condor_q* command , but is queryable with the *condor_history* command.

4.3 Submitting a Job

The *condor_submit* command takes a job description file as input and submits the job to HTCondor. In the submit description file, HTCondor finds everything it needs to know about the job. Items such as the name of the executable to run, the initial working directory, and command-line arguments to the program all go into the submit description file. *condor_submit* creates a job ClassAd based upon the information, and HTCondor works toward running the job.

It is easy to submit multiple runs of a program to HTCondor with a single submit description file. To run the same program many times with different input data sets, arrange the data files accordingly so that each run reads its own input, and each run writes its own output. Each individual run may have its own initial working directory, files mapped for `stdin`, `stdout`, `stderr`, command-line arguments, and shell environment.

The *condor_submit* manual page contains a complete and full description of how to use *condor_submit*. It also includes descriptions of all of the many commands that may be placed into a submit description file. In addition, the index lists entries for each command under the heading of Submit Commands.

4.3.1 Sample submit description files

In addition to the examples of submit description files given here, there are more in the *condor_submit* manual page.

Example 1

Example 1 is one of the simplest submit description files possible. It queues the program *myexe* for execution somewhere in the pool. As this submit description file does not request a specific operating system to run on, HTCondor will use the default, which is to run the job on a machine which has the same architecture and operating system it was submitted from.

Before submitting a job to HTCondor, it is a good idea to test it first locally, by running it from a command shell. This example job might look like this when run from the shell prompt.

```
$ ./myexe SomeArgument
```

The corresponding submit description file might look like the following

```
# Example 1
# Simple HTCondor submit description file
# Everything with a leading # is a comment

executable   = myexe
arguments    = SomeArgument

output       = outputfile
error        = errorfile
log          = myexe.log

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

should_transfer_files = yes

queue
```

The standard output for this job will go to the file `outputfile`, as specified by the **output** command. Likewise, the standard error output will go to `errorfile`.

HTCondor will append events about the job to a log file with the requested name `myexe.log`. When the job finishes, its exit conditions and resource usage will also be noted in the log file. This file's contents are an excellent way to figure out what happened to jobs.

HTCondor needs to know how many machine resources to allocate to this job. The `request_` lines describe that this job should be allocated 1 cpu core, 1024 megabytes of memory and 10240 kilobytes of scratch disk space.

Finally, the queue statement tells HTCondor that you are done describing the job, and to send it to the queue for processing.

Example 2

The submit description file for Example 2 queues 150 runs of program *foo*. This job requires machines which have at least 4 GiB of physical memory, one cpu core and 16 Gb of scratch disk. Each of the 150 runs of the program is given its own HTCondor process number, starting with 0. $(Process) is expanded by HTCondor to the actual number used by each instance of the job. So, `stdout`, and `stderr` will refer to `out.0`, and `err.0` for the first run of the program, `out.1`, and `err.1` for the second run of the program, and so forth. A log file containing entries about when and where HTCondor runs, transfer files, and terminates for all the 150 queued programs will be written into the single

file `foo.log`. If there are 150 or more available slots in your pool, all 150 instances might be run at the same time, otherwise, HTCondor will run as many as it can concurrently.

Each instance of this program works on one input file. The name of this input file is passed to the program as the only argument. We prepare 150 copies of this input file in the current directory, and name them input_file.0, input_file.1 ... up to input_file.149. Using transfer_input_files, we tell HTCondor which input file to send to each instance of the program.

```
# Example 2: Show off some fancy features,
# including the use of pre-defined macros.

executable     = foo
arguments      = input_file.$(Process)

request_cpus   = 1
request_memory = 4096M
request_disk   = 16383K

error   = err.$(Process)
output  = out.$(Process)
log     = foo.log

should_transfer_files = yes
transfer_input_files = input_file.$(Process)

# submit 150 instances of this job
queue 150
```

4.3.2 Submitting many similar jobs with one queue command

A wide variety of job submissions can be specified with extra information to the **queue** submit command. This flexibility eliminates the need for a job wrapper or Perl script for many submissions.

The form of the **queue** command defines variables and expands values, identifying a set of jobs. Square brackets identify an optional item.

queue [**<int expr>**]

queue [**<int expr>**] [**<varname>**] **in** [**slice**] **<list of items>**

queue [**<int expr>**] [**<varname>**] **matching** [**files | dirs**] [**slice**] **<list of items with file globbing>**

queue [**<int expr>**] [**<list of varnames>**] **from** [**slice**] **<file name> | <list of items>**

All optional items have defaults:

- If <`int expr`> is not specified, it defaults to the value 1.
- If <`varname`> or <`list of varnames`> is not specified, it defaults to the single variable called `ITEM`.
- If `slice` is not specified, it defaults to all elements within the list. This is the Python slice `[::]`, with a step value of 1.
- If neither `files` nor `dirs` is specified in a specification using the **from** key word, then both files and directories are considered when globbing.

The list of items uses syntax in one of two forms. One form is a comma and/or space separated list; the items are placed on the same line as the **queue** command. The second form separates items by placing each list item on its own line, and delimits the list with parentheses. The opening parenthesis goes on the same line as the **queue** command. The closing parenthesis goes on its own line. The **queue** command specified with the key word **from** will always use the second form of this syntax. Example 3 below uses this second form of syntax. Finally, the key word **from** accepts a shell command in place of file name, followed by a pipe | (example 4).

The optional `slice` specifies a subset of the list of items using the Python syntax for a slice. Negative step values are not permitted.

Here are a set of examples.

Example 1

```
transfer_input_files = $(filename)
arguments            = -infile $(filename)
queue filename matching files *.dat
```

The use of file globbing expands the list of items to be all files in the current directory that end in `.dat`. Only files, and not directories are considered due to the specification of `files`. One job is queued for each file in the list of items. For this example, assume that the three files `initial.dat`, `middle.dat`, and `ending.dat` form the list of items after expansion; macro `filename` is assigned the value of one of these file names for each job queued. That macro value is then substituted into the **arguments** and **transfer_input_files** commands. The **queue** command expands to

```
transfer_input_files = initial.dat
arguments            = -infile initial.dat
queue
transfer_input_files = middle.dat
arguments            = -infile middle.dat
queue
transfer_input_files = ending.dat
arguments            = -infile ending.dat
queue
```

Example 2

```
queue 1 input in A, B, C
```

Variable `input` is set to each of the 3 items in the list, and one job is queued for each. For this example the **queue** command expands to

```
input = A
queue
input = B
queue
input = C
queue
```

Example 3

```
queue input, arguments from (
  file1, -a -b 26
  file2, -c -d 92
)
```

Using the `from` form of the options, each of the two variables specified is given a value from the list of items. For this example the **queue** command expands to

```
input = file1
arguments = -a -b 26
queue
input = file2
arguments = -c -d 92
queue
```

Example 4

```
queue from seq 7 9 |
```

feeds the list of items to queue with the output of `seq 7 9`:

```
item = 7
queue
item = 8
queue
item = 9
queue
```

4.3.3 Variables in the Submit Description File

There are automatic variables for use within the submit description file.

$(Cluster) or $(ClusterId)
: Each set of queued jobs from a specific user, submitted from a single submit host, sharing an executable have the same value of `$(Cluster)` or `$(ClusterId)`. The first cluster of jobs are assigned to cluster 0, and the value is incremented by one for each new cluster of jobs. `$(Cluster)` or `$(ClusterId)` will have the same value as the job ClassAd attribute `ClusterId`.

$(Process) or $(ProcId)
: Within a cluster of jobs, each takes on its own unique `$(Process)` or `$(ProcId)` value. The first job has value 0. `$(Process)` or `$(ProcId)` will have the same value as the job ClassAd attribute `ProcId`.

$$(a_machine_classad_attribute)
: When the machine is matched to this job for it to run on, any dollar-dollar expressions are looked up from the machine ad, and then expanded. This lets you put the value of some machine ad attribute into your job. For example, if you to pass the actual amount of memory a slot has provisioned as an argument to the job, you could add `arguments = --mem $$(Memory)`

```
arguments = --mem $$(Memory)
```

or, if you wanted to put the name of the machine the job ran on into the output file name, you could add

```
output = output_file.$$(Name)
```

$$([an_evaluated_classad_expression])
: This dollar-dollar-bracket syntax is useful when you need to perform some math on a value before passing it to your job. For example, if want to pass 90% of the allocated memory as an argument to your job, the submit file can have

```
arguments = --mem $$([ Memory * 0.9 ])
```

and when the job is matched to a machine, condor will evaluate this expression in the context of both the job and machine ad

$(ARCH)
The Architecture that HTCondor is running on, or the ARCH variable in the config file. Example might be X86_64.

$(OPSYS) $(OPSYSVER) $(OPSYSANDVER) $(OPSYSMAJORVER)
These submit file macros are availle at submit time, and mimic the classad attributes of the same names.

$(SUBMIT_FILE)
The name of the submit_file as passed to the `condor_submit` command.

$(SUBMIT_TIME)
The Unix epoch time submit was run. Note, this may be useful for naming output files.

$(Year) $(Month) $(Day)
These integer values are derived from the *$(SUBMIT_TIME)* macro above.

$(Item)
The default name of the variable when no <varname> is provided in a **queue** command.

$(ItemIndex)
Represents an index within a list of items. When no slice is specified, the first `$(ItemIndex)` is 0. When a slice is specified, `$(ItemIndex)` is the index of the item within the original list.

$(Step)
For the <int expr> specified, `$(Step)` counts, starting at 0.

$(Row)
When a list of items is specified by placing each item on its own line in the submit description file, `$(Row)` identifies which line the item is on. The first item (first line of the list) is `$(Row)` 0. The second item (second line of the list) is `$(Row)` 1. When a list of items are specified with all items on the same line, `$(Row)` is the same as `$(ItemIndex)`.

Here is an example of a **queue** command for which the values of these automatic variables are identified.

Example 1

This example queues six jobs.

```
queue 3 in (A, B)
```

- `$(Process)` takes on the six values 0, 1, 2, 3, 4, and 5.
- Because there is no specification for the <varname> within this **queue** command, variable `$(Item)` is defined. It has the value `A` for the first three jobs queued, and it has the value `B` for the second three jobs queued.
- `$(Step)` takes on the three values 0, 1, and 2 for the three jobs with `$(Item)=A`, and it takes on the same three values 0, 1, and 2 for the three jobs with `$(Item)=B`.
- `$(ItemIndex)` is 0 for all three jobs with `$(Item)=A`, and it is 1 for all three jobs with `$(Item)=B`.
- `$(Row)` has the same value as `$(ItemIndex)` for this example.

4.3.4 Including Submit Commands Defined Elsewhere

Externally defined submit commands can be incorporated into the submit description file using the syntax

```
include : <what-to-include>
```

The <what-to-include> specification may specify a single file, where the contents of the file will be incorporated into the submit description file at the point within the file where the **include** is. Or, <what-to-include> may cause a program to be executed, where the output of the program is incorporated into the submit description file. The specification of <what-to-include> has the bar character (|) following the name of the program to be executed.

The **include** key word is case insensitive. There are no requirements for white space characters surrounding the colon character.

Included submit commands may contain further nested **include** specifications, which are also parsed, evaluated, and incorporated. Levels of nesting on included files are limited, such that infinite nesting is discovered and thwarted, while still permitting nesting.

Consider the example

```
include : ./list-infiles.sh |
```

In this example, the bar character at the end of the line causes the script `list-infiles.sh` to be invoked, and the output of the script is parsed and incorporated into the submit description file. If this bash script is in the PATH when submit is run, and contains

```
#!/bin/sh

echo "transfer_input_files = `ls -m infiles/*.dat`"
exit 0
```

then the output of this script has specified the set of input files to transfer to the execute host. For example, if directory `infiles` contains the three files `A.dat`, `B.dat`, and `C.dat`, then the submit command

```
transfer_input_files = infiles/A.dat, infiles/B.dat, infiles/C.dat
```

is incorporated into the submit description file.

4.3.5 Using Conditionals in the Submit Description File

Conditional if/else semantics are available in a limited form. The syntax:

```
if <simple condition>
   <statement>
   . . .
   <statement>
else
   <statement>
   . . .
   <statement>
endif
```

An else key word and statements are not required, such that simple if semantics are implemented. The <simple condition> does not permit compound conditions. It optionally contains the exclamation point character (!) to represent the not operation, followed by

- the defined keyword followed by the name of a variable. If the variable is defined, the statement(s) are incorporated into the expanded input. If the variable is not defined, the statement(s) are not incorporated into the expanded input. As an example,

```
if defined MY_UNDEFINED_VARIABLE
   X = 12
else
   X = -1
endif
```

  results in `X = -1`, when `MY_UNDEFINED_VARIABLE` is not yet defined.

- the version keyword, representing the version number of of the daemon or tool currently reading this conditional. This keyword is followed by an HTCondor version number. That version number can be of the form x.y.z or x.y. The version of the daemon or tool is compared to the specified version number. The comparison operators are
  - == for equality. Current version 8.2.3 is equal to 8.2.
  - >= to see if the current version number is greater than or equal to. Current version 8.2.3 is greater than 8.2.2, and current version 8.2.3 is greater than or equal to 8.2.
  - <= to see if the current version number is less than or equal to. Current version 8.2.0 is less than 8.2.2, and current version 8.2.3 is less than or equal to 8.2.

  As an example,

```
if version >= 8.1.6
   DO_X = True
else
   DO_Y = True
endif
```

  results in defining `DO_X` as `True` if the current version of the daemon or tool reading this if statement is 8.1.6 or a more recent version.

- True or yes or the value 1. The statement(s) are incorporated.
- False or no or the value 0 The statement(s) are not incorporated.
- $(<variable>) may be used where the immediately evaluated value is a simple boolean value. A value that evaluates to the empty string is considered False, otherwise a value that does not evaluate to a simple boolean value is a syntax error.

The syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
elif <simple condition>
   <statement>
   . . .
   <statement>
endif
```

is the same as syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
else
   if <simple condition>
      <statement>
      . . .
      <statement>
   endif
endif
```

Here is an example use of a conditional in the submit description file. A portion of the `sample.sub` submit description file uses the if/else syntax to define command line arguments in one of two ways:

```
if defined X
  arguments = -n $(X)
else
  arguments = -n 1 -debug
endif
```

Submit variable `X` is defined on the *condor_submit* command line with

```
$ condor_submit  X=3  sample.sub
```

This command line incorporates the submit command `X = 3` into the submission before parsing the submit description file. For this submission, the command line arguments of the submitted job become

```
arguments = -n 3
```

If the job were instead submitted with the command line

```
$ condor_submit  sample.sub
```

then the command line arguments of the submitted job become

```
arguments = -n 1 -debug
```

4.3.6 Function Macros in the Submit Description File

A set of predefined functions increase flexibility. Both submit description files and configuration files are read using the same parser, so these functions may be used in both submit description files and configuration files.

Case is significant in the function's name, so use the same letter case as given in these definitions.

$CHOICE(index, listname) or $CHOICE(index, item1, item2, ...)
: An item within the list is returned. The list is represented by a parameter name, or the list items are the parameters. The `index` parameter determines which item. The first item in the list is at index 0. If the index is out of bounds for the list contents, an error occurs.

$ENV(environment-variable-name[:default-value])
: Evaluates to the value of environment variable `environment-variable-name`. If there is no environment

variable with that name, Evaluates to UNDEFINED unless the optional :default-value is used; in which case it evaluates to default-value. For example,

```
A = $ENV(HOME)
```

binds `A` to the value of the `HOME` environment variable.

`$F[fpduwnxbqa](filename)`
One or more of the lower case letters may be combined to form the function name and thus, its functionality. Each letter operates on the `filename` in its own way.

- `f` convert relative path to full path by prefixing the current working directory to it. This option works only in *condor_submit* files.
- `p` refers to the entire directory portion of `filename`, with a trailing slash or backslash character. Whether a slash or backslash is used depends on the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified.
- `d` refers to the last portion of the directory within the path, if specified. It will have a trailing slash or backslash, as appropriate to the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified unless u or w is used. if b is used the trailing slash or backslash will be omitted.
- `u` convert path separators to Unix style slash characters
- `w` convert path separators to Windows style backslash characters
- `n` refers to the file name at the end of any path, but without any file name extension. As an example, the return value from `$Fn(/tmp/simulate.exe)` will be `simulate` (without the `.exe` extension).
- `x` refers to a file name extension, with the associated period (`.`). As an example, the return value from `$Fn(/tmp/simulate.exe)` will be `.exe`.
- `b` when combined with the d option, causes the trailing slash or backslash to be omitted. When combined with the x option, causes the leading period (`.`) to be omitted.
- `q` causes the return value to be enclosed within quotes. Double quote marks are used unless a is also specified.
- `a` When combined with the q option, causes the return value to be enclosed within single quotes.

`$DIRNAME(filename)` is the same as `$Fp(filename)`

`$BASENAME(filename)` is the same as `$Fnx(filename)`

`$INT(item-to-convert) or $INT(item-to-convert, format-specifier)`
Expands, evaluates, and returns a string version of `item-to-convert`. The `format-specifier` has the same syntax as a C language or Perl format specifier. If no `format-specifier` is specified, "%d" is used as the format specifier.

`$RANDOM_CHOICE(choice1, choice2, choice3, ...)`
A random choice of one of the parameters in the list of parameters is made. For example, if one of the integers 0-8 (inclusive) should be randomly chosen:

```
$RANDOM_CHOICE(0,1,2,3,4,5,6,7,8)
```

`$RANDOM_INTEGER(min, max [, step])`
A random integer within the range min and max, inclusive, is selected. The optional step parameter controls the stride within the range, and it defaults to the value 1. For example, to randomly chose an even integer in the range 0-8 (inclusive):

```
$RANDOM_INTEGER(0, 8, 2)
```

$REAL(item-to-convert) or $REAL(item-to-convert, format-specifier)
Expands, evaluates, and returns a string version of `item-to-convert` for a floating point type. The `format-specifier` is a C language or Perl format specifier. If no `format-specifier` is specified, "%16G" is used as a format specifier.

$SUBSTR(name, start-index) or $SUBSTR(name, start-index, length)
Expands name and returns a substring of it. The first character of the string is at index 0. The first character of the substring is at index start-index. If the optional length is not specified, then the substring includes characters up to the end of the string. A negative value of start-index works back from the end of the string. A negative value of length eliminates use of characters from the end of the string. Here are some examples that all assume

```
Name = abcdef
```

- `$SUBSTR(Name, 2)` is `cdef`.
- `$SUBSTR(Name, 0, -2)` is `abcd`.
- `$SUBSTR(Name, 1, 3)` is `bcd`.
- `$SUBSTR(Name, -1)` is `f`.
- `$SUBSTR(Name, 4, -3)` is the empty string, as there are no characters in the substring for this request.

Here are example uses of the function macros in a submit description file. Note that these are not complete submit description files, but only the portions that promote understanding of use cases of the function macros.

Example 1

Generate a range of numerical values for a set of jobs, where values other than those given by $(Process) are desired.

```
MyIndex     = $(Process) + 1
initial_dir = run-$INT(MyIndex,%04d)
```

Assuming that there are three jobs queued, such that $(Process) becomes 0, 1, and 2, `initial_dir` will evaluate to the directories `run-0001`, `run-0002`, and `run-0003`.

Example 2

This variation on Example 1 generates a file name extension which is a 3-digit integer value.

```
Values     = $(Process) * 10
Extension  = $INT(Values,%03d)
input      = X.$(Extension)
```

Assuming that there are four jobs queued, such that $(Process) becomes 0, 1, 2, and 3, `Extension` will evaluate to 000, 010, 020, and 030, leading to files defined for **input** of `X.000`, `X.010`, `X.020`, and `X.030`.

Example 3

This example uses both the file globbing of the **queue** command and a macro function to specify a job input file that is within a subdirectory on the submit host, but will be placed into a single, flat directory on the execute host.

```
arguments            = $Fnx(FILE)
transfer_input_files = $(FILE)
queue FILE matching (
    samplerun/*.dat
)
```

Assume that two files that end in `.dat`, `A.dat` and `B.dat`, are within the directory `samplerun`. Macro `FILE` expands to `samplerun/A.dat` and `samplerun/B.dat` for the two jobs queued. The input files transferred are `samplerun/A.dat` and `samplerun/B.dat` on the submit host. The `$Fnx()` function macro expands to the complete file name with any leading directory specification stripped, such that the command line argument for one of the jobs will be `A.dat` and the command line argument for the other job will be `B.dat`.

4.3.7 About Requirements and Rank

The `requirements` and `rank` commands in the submit description file are powerful and flexible. Using them effectively requires care, and this section presents those details.

Both `requirements` and `rank` need to be specified as valid HTCondor ClassAd expressions, however, default values are set by the *condor_submit* program if these are not defined in the submit description file. From the *condor_submit* manual page and the above examples, you see that writing ClassAd expressions is intuitive, especially if you are familiar with the programming language C. There are some pretty nifty expressions you can write with ClassAds. A complete description of ClassAds and their expressions can be found in the *HTCondor's ClassAd Mechanism* section.

All of the commands in the submit description file are case insensitive, except for the ClassAd attribute string values. ClassAd attribute names are case insensitive, but ClassAd string values are case preserving.

Note that the comparison operators (<, >, <=, >=, and ==) compare strings case insensitively. The special comparison operators =?= and =!= compare strings case sensitively.

A **requirements** or **rank** command in the submit description file may utilize attributes that appear in a machine or a job ClassAd. Within the submit description file (for a job) the prefix MY. (on a ClassAd attribute name) causes a reference to the job ClassAd attribute, and the prefix TARGET. causes a reference to a potential machine or matched machine ClassAd attribute.

The *condor_status* command displays statistics about machines within the pool. The **-l** option displays the machine ClassAd attributes for all machines in the HTCondor pool. The job ClassAds, if there are jobs in the queue, can be seen with the *condor_q -l* command. This shows all the defined attributes for current jobs in the queue.

A list of defined ClassAd attributes for job ClassAds is given in the Appendix on the *Job ClassAd Attributes* page. A list of defined ClassAd attributes for machine ClassAds is given in the Appendix on the *Machine ClassAd Attributes* page.

Rank Expression Examples

When considering the match between a job and a machine, rank is used to choose a match from among all machines that satisfy the job's requirements and are available to the user, after accounting for the user's priority and the machine's rank of the job. The rank expressions, simple or complex, define a numerical value that expresses preferences.

The job's `Rank` expression evaluates to one of three values. It can be UNDEFINED, ERROR, or a floating point value. If `Rank` evaluates to a floating point value, the best match will be the one with the largest, positive value. If no `Rank` is given in the submit description file, then HTCondor substitutes a default value of 0.0 when considering machines to match. If the job's `Rank` of a given machine evaluates to UNDEFINED or ERROR, this same value of 0.0 is used. Therefore, the machine is still considered for a match, but has no ranking above any other.

A boolean expression evaluates to the numerical value of 1.0 if true, and 0.0 if false.

The following `Rank` expressions provide examples to follow.

For a job that desires the machine with the most available memory:

```
Rank = memory
```

For a job that prefers to run on a friend's machine on Saturdays and Sundays:

```
Rank = ( (clockday == 0) || (clockday == 6) )
       && (machine == "friend.cs.wisc.edu")
```

For a job that prefers to run on one of three specific machines:

```
Rank = (machine == "friend1.cs.wisc.edu") ||
       (machine == "friend2.cs.wisc.edu") ||
       (machine == "friend3.cs.wisc.edu")
```

For a job that wants the machine with the best floating point performance (on Linpack benchmarks):

```
Rank = kflops
```

This particular example highlights a difficulty with `Rank` expression evaluation as currently defined. While all machines have floating point processing ability, not all machines will have the `kflops` attribute defined. For machines where this attribute is not defined, `Rank` will evaluate to the value UNDEFINED, and HTCondor will use a default rank of the machine of 0.0. The `Rank` attribute will only rank machines where the attribute is defined. Therefore, the machine with the highest floating point performance may not be the one given the highest rank.

So, it is wise when writing a `Rank` expression to check if the expression's evaluation will lead to the expected resulting ranking of machines. This can be accomplished using the *condor_status* command with the *-constraint* argument. This allows the user to see a list of machines that fit a constraint. To see which machines in the pool have `kflops` defined, use

```
$ condor_status -constraint kflops
```

Alternatively, to see a list of machines where `kflops` is not defined, use

```
$ condor_status -constraint "kflops=?=undefined"
```

For a job that prefers specific machines in a specific order:

```
Rank = ((machine == "friend1.cs.wisc.edu")*3) +
       ((machine == "friend2.cs.wisc.edu")*2) +
        (machine == "friend3.cs.wisc.edu")
```

If the machine being ranked is `friend1.cs.wisc.edu`, then the expression

```
(machine == "friend1.cs.wisc.edu")
```

is true, and gives the value 1.0. The expressions

```
(machine == "friend2.cs.wisc.edu")
```

and

```
(machine == "friend3.cs.wisc.edu")
```

are false, and give the value 0.0. Therefore, `Rank` evaluates to the value 3.0. In this way, machine `friend1.cs.wisc.edu` is ranked higher than machine `friend2.cs.wisc.edu`, machine `friend2.cs.wisc.edu` is ranked higher than machine `friend3.cs.wisc.edu`, and all three of these machines are ranked higher than others.

4.3.8 Submitting Jobs Using a Shared File System

If vanilla, java, or parallel universe jobs are submitted without using the File Transfer mechanism, HTCondor must use a shared file system to access input and output files. In this case, the job must be able to access the data files from any machine on which it could potentially run.

As an example, suppose a job is submitted from blackbird.cs.wisc.edu, and the job requires a particular data file called `/u/p/s/psilord/data.txt`. If the job were to run on cardinal.cs.wisc.edu, the file `/u/p/s/psilord/data.txt` must be available through either NFS or AFS for the job to run correctly.

HTCondor allows users to ensure their jobs have access to the right shared files by using the `FileSystemDomain` and `UidDomain` machine ClassAd attributes. These attributes specify which machines have access to the same shared file systems. All machines that mount the same shared directories in the same locations are considered to belong to the same file system domain. Similarly, all machines that share the same user information (in particular, the same UID, which is important for file systems like NFS) are considered part of the same UID domain.

The default configuration for HTCondor places each machine in its own UID domain and file system domain, using the full host name of the machine as the name of the domains. So, if a pool does have access to a shared file system, the pool administrator must correctly configure HTCondor such that all the machines mounting the same files have the same `FileSystemDomain` configuration. Similarly, all machines that share common user information must be configured to have the same `UidDomain` configuration.

When a job relies on a shared file system, HTCondor uses the `requirements` expression to ensure that the job runs on a machine in the correct `UidDomain` and `FileSystemDomain`. In this case, the default `requirements` expression specifies that the job must run on a machine with the same `UidDomain` and `FileSystemDomain` as the machine from which the job is submitted. This default is almost always correct. However, in a pool spanning multiple `UidDomain`s and/or `FileSystemDomain`s, the user may need to specify a different `requirements` expression to have the job run on the correct machines.

For example, imagine a pool made up of both desktop workstations and a dedicated compute cluster. Most of the pool, including the compute cluster, has access to a shared file system, but some of the desktop machines do not. In this case, the administrators would probably define the `FileSystemDomain` to be `cs.wisc.edu` for all the machines that mounted the shared files, and to the full host name for each machine that did not. An example is `jimi.cs.wisc.edu`.

In this example, a user wants to submit vanilla universe jobs from her own desktop machine (jimi.cs.wisc.edu) which does not mount the shared file system (and is therefore in its own file system domain, in its own world). But, she wants the jobs to be able to run on more than just her own machine (in particular, the compute cluster), so she puts the program and input files onto the shared file system. When she submits the jobs, she needs to tell HTCondor to send them to machines that have access to that shared data, so she specifies a different `requirements` expression than the default:

```
Requirements = TARGET.UidDomain == "cs.wisc.edu" && \
               TARGET.FileSystemDomain == "cs.wisc.edu"
```

WARNING: If there is no shared file system, or the HTCondor pool administrator does not configure the `FileSystemDomain` setting correctly (the default is that each machine in a pool is in its own file system and UID domain), a user submits a job that cannot use remote system calls (for example, a vanilla universe job), and the user does not enable HTCondor's File Transfer mechanism, the job will only run on the machine from which it was submitted.

4.3.9 Jobs That Require Credentials

If the HTCondor pool administrator has configured the access point with one or more credential monitors, jobs submitted on that machine may automatically be provided with credentials and/or it may be possible for users to request and obtain credentials for their jobs.

Suppose the administrator has configured the access point such that users may obtain credentials from a storage service called "CloudBoxDrive." A job that needs credentials from CloudBoxDrive should contain the submit command

```
use_oauth_services = cloudboxdrive
```

Upon submitting this job for the first time, the user will be directed to a webpage hosted on the access point which will guide the user through the process of obtaining a CloudBoxDrive credential. The credential is then stored securely on the access point. (**Note: depending on which credential monitor is used, the original job may have to be re-submitted at this point.**) (Also note that at no point is the user's *password* stored on the access point.) Once a credential is stored on the access point, as long as it remains valid, it is transferred securely to all subsequently submitted jobs that contain `use_oauth_services = cloudboxdrive`.

When a job that contains credentials runs on an execute machine, the job's executable will have the environment variable `_CONDOR_CREDS` set, which points to the location of all of the credentials inside the job's sandbox. For credentials obtained via the `use_oauth_services` submit file command, the "access token" is stored under `$_CONDOR_CREDS` in a JSON-encoded file named with the name of the service provider and with the extension `.use`. For the "CloudBoxDrive" example, the access token would be located in `$_CONDOR_CREDS/cloudboxdrive.use`.

The HTCondor file transfer mechanism has built-in plugins for using user-obtained credentials to transfer files from some specific storage providers, see *File Transfer Using a URL*.

Some credential providers may require the user to provide a description of the permissions (often called "scopes") a user needs for a specific credential. Credential permission scoping is possible using the `<service name>_oauth_permissions` submit file command. For example, suppose our CloudBoxDrive service has a `/public` directory, and the documentation for the service said that users must specify a `read:<directory>` scope in order to be able to read data out of `<directory>`. The submit file would need to contain

```
use_oauth_services = cloudboxdrive
cloudboxdrive_oauth_permissions = read:/public
```

Some credential providers may also require the user to provide the name of the resource (or "audience") that a credential should allow access to. Resource naming is done using the `<service name>_oauth_resource` submit file command. For example, if our CloudBoxDrive service has servers located at some universities and the documentation says that we should pick one near us and specify it as the audience, the submit file might look like

```
use_oauth_services = cloudboxdrive
cloudboxdrive_oauth_permissions = read:/public
cloudboxdrive_oauth_resource = https://cloudboxdrive.myuni.edu
```

It is possible for a single job to request and/or use credentials from multiple services by listing each service in the `use_oauth_services` command. Suppose the nearby university has a SciTokens service that provides credentials to access the `localstorage.myuni.edu` machine, and the HTCondor pool administrator has configured the access point to allow users to obtain credentials from this service, and that a user has write access to the */foo* directory on the storage machine. A submit file that would result in a job that contains credentials that can read from CloudBoxDrive and write to the local university storage might look like

```
use_oauth_services = cloudboxdrive, myuni
```

(continues on next page)

(continued from previous page)

```
cloudboxdrive_oauth_permissions = read:/public
cloudboxdrive_oauth_resource = https://cloudboxdrive.myuni.edu

myuni_oauth_permissions = write:/foo
myuni_oauth_resource = https://localstorage.myuni.edu
```

A single job can also request multiple credentials from the same service provider by affixing handles to the <service>_oauth_permissions and (if necessary) <service>_oauth_resource commands. For example, if a user wants separate read and write credentials for CloudBoxDrive

```
use_oauth_services = cloudboxdrive
cloudboxdrive_oauth_permissions_readpublic = read:/public
cloudboxdrive_oauth_permissions_writeprivate = write:/private

cloudboxdrive_oauth_resource_readpublic = https://cloudboxdrive.myuni.edu
cloudboxdrive_oauth_resource_writeprivate = https://cloudboxdrive.myuni.edu
```

Submitting the above would result in a job with respective access tokens located in $_CONDOR_CREDS/cloudboxdrive_readpublic.use and $_CONDOR_CREDS/cloudboxdrive_writeprivate.use.

Note that the permissions and resource settings for each handle (and for no handle) are stored separately from the job so multiple jobs from the same user running at the same time or for a period of time consecutively may not use a different set of permissions and resource settings for the same service and handle. If that is attempted, a new job submission will fail with instructions on how to resolve the conflict, but the safest thing is to choose a unique handle.

If a service provider does not require permissions or resources to be specified, a user can still request multiple credentials by affixing handles to <service>_oauth_permissions commands with empty values

```
use_oauth_services = cloudboxdrive
cloudboxdrive_oauth_permissions_personal =
cloudboxdrive_oauth_permissions_public =
```

When the Vault credential monitor is configured, the service name may optionally be split into two parts with an underscore between them, where the first part is the issuer and the second part is the role. In this example the issuer is "dune" and the role is "production", both as configured by the administrator of the Vault server:

```
use_oauth_services = dune_production
```

Vault does not require permissions or resources to be set, but they may be set to reduce the default permissions or restrict the resources that may use the credential. The full service name including an underscore may be used in an oauth_permissions or oauth_resource. Avoid using handles that might be confused as role names. For example, the following will result in a conflict between two credentials called dune_production.use:

```
use_oauth_services = dune, dune_production
dune_oauth_permissions_production =
dune_production_oauth_permissions =
```

4.3.10 Jobs That Require GPUs

HTCondor has built-in support for detecting machines with GPUs, and matching jobs that need GPUs to machines that have them. If your job needs a GPU, you'll first need to tell HTCondor how many GPUs each job needs with the submit command:

```
request_GPUs = <n>
```

where <n> is replaced by the integer quantity of GPUs required for the job. For example, a job that needs 1 GPU uses

```
request_GPUs = 1
```

Because there are different capabilities among GPUs, your job might need to further qualify which GPU is required. The submit command *require_gpus* does this. For example, to request a CUDA GPU whose CUDA Capability is at least 8, add the following to your submit file:

```
request_GPUs = 1
require_gpus = Capability >= 8.0
```

To see which CUDA capabilities are available in your HTCondor pool, you can run the command

```
$ condor_status -af Name GPUS_Capability
```

To see which GPU devices HTCondor has detected on your pool, you can run the command

```
$ condor_status -af Name GPUS_DeviceName
```

Access to GPU resources by an HTCondor job needs special configuration of the machines that offer GPUs. Details of how to set up the configuration are in the *Policy Configuration for Execution Points and for Access Points* section.

4.3.11 Interactive Jobs

An interactive job is a Condor job that is provisioned and scheduled like any other vanilla universe Condor job onto an execute machine within the pool. The result of a running interactive job is a shell prompt issued on the execute machine where the job runs. The user that submitted the interactive job may then use the shell as desired, perhaps to interactively run an instance of what is to become a Condor job. This might aid in checking that the set up and execution environment are correct, or it might provide information on the RAM or disk space needed. This job (shell) continues until the user logs out or any other policy implementation causes the job to stop running. A useful feature of the interactive job is that the users and jobs are accounted for within Condor's scheduling and priority system.

Neither the submit nor the execute host for interactive jobs may be on Windows platforms.

The current working directory of the shell will be the initial working directory of the running job. The shell type will be the default for the user that submits the job. At the shell prompt, X11 forwarding is enabled.

Each interactive job will have a job ClassAd attribute of

```
InteractiveJob = True
```

Submission of an interactive job specifies the option **-interactive** on the *condor_submit* command line.

A submit description file may be specified for this interactive job. Within this submit description file, a specification of these 5 commands will be either ignored or altered:

1. **executable**
2. **transfer_executable**
3. **arguments**
4. **universe** . The interactive job is a vanilla universe job.
5. **queue <n>**. In this case the value of **<n>** is ignored; exactly one interactive job is queued.

The submit description file may specify anything else needed for the interactive job, such as files to transfer.

If no submit description file is specified for the job, a default one is utilized as identified by the value of the configuration variable `INTERACTIVE_SUBMIT_FILE` .

Here are examples of situations where interactive jobs may be of benefit.

- An application that cannot be batch processed might be run as an interactive job. Where input or output cannot be captured in a file and the executable may not be modified, the interactive nature of the job may still be run on a pool machine, and within the purview of Condor.
- A pool machine with specialized hardware that requires interactive handling can be scheduled with an interactive job that utilizes the hardware.
- The debugging and set up of complex jobs or environments may benefit from an interactive session. This interactive session provides the opportunity to run scripts or applications, and as errors are identified, they can be corrected on the spot.
- Development may have an interactive nature, and proceed more quickly when done on a pool machine. It may also be that the development platforms required reside within Condor's purview as execute hosts.

4.3.12 Submitting Lots of Jobs

When submitting a lot of jobs with a single submit file, you can dramatically speed up submission and reduce the load on the *condor_schedd* by submitting the jobs as a late materialization job factory.

A submission of this form sends a single ClassAd, called the Cluster ad, to the *condor_schedd*, as well as instructions to create the individual jobs as variations on that Cluster ad. These instructions are sent as a *submit digest* and optional *itemdata*. The *submit digest* is the submit file stripped down to just the statements that vary between jobs. The *itemdata* is the arguments to the `Queue` statement when the arguments are more than just a count of jobs.

The *condor_schedd* will use the *submit digest* and the *itemdata* to create the individual job ClassAds when they are needed. Materialization is controlled by two values stored in the Cluster classad, and by optional limits configured in the *condor_schedd*.

The `max_idle` limit specifies the maximum number of non-running jobs that should be materialized in the *condor_schedd* at any one time. One or more jobs will materialize whenever a job enters the Run state and the number of non-running jobs that are still in the *condor_schedd* is less than this limit. This limit is stored in the Cluster ad in the *JobMaterializeMaxIdle* attribute.

The `max_materialize` limit specifies an overall limit on the number of jobs that can be materialized in the *condor_schedd* at any one time. One or more jobs will materialize when a job leaves the *condor_schedd* and the number of materialized jobs remaining is less than this limit. This limit is stored in the Cluster ad in the *JobMaterializeLimit* attribute.

Late materialization can be used as a way for a user to submit millions of jobs without hitting the or limits in the *condor_schedd*, since the *condor_schedd* will enforce these limits by applying them to the `max_materialize` and `max_idle` values specified in the Cluster ad.

To give an example, the following submit file:

```
executable     = foo
arguments      = input_file.$(Process)

request_cpus   = 1
request_memory = 4096M
request_disk   = 16383K

error   = err.$(Process)
output  = out.$(Process)
log     = foo.log

should_transfer_files = yes
transfer_input_files = input_file.$(Process)

# submit as a factory with an idle jobs limit
max_idle = 100

# submit 15,000 instances of this job
queue 15*1000
```

When submitted as a late materialization factory, the *submit digest* for this factory will contain only the submit statements that vary between jobs, and the collapsed queue statement like this:

```
arguments = input_file.$(Process)
error = err.$(Process)
output = out.$(Process)
transfer_input_files = input_file.$(Process)

queue 15000
```

Materialization log events

When a Late Materialization job factory is submitted to the *condor_schedd*, a `Cluster submitted` event will be written to the UserLog of the Cluster ad. This will be the same log file used by the first job materialized by the factory. To avoid confusion, it is recommended that you use the same log file for all jobs in the factory.

When the Late Materialization job factory is removed from the *condor_schedd*, a `Cluster removed` event will be written to the UserLog of the Cluster ad. This event will indicate how many jobs were materialized before the factory was removed.

If Late Materialization of jobs is paused due to an error in materialization or because condor_hold was used to hold the cluster id, a `Job Materialization Paused` event will be written to the UserLog of the Cluster ad. This event will indicate the reason for the pause.

When `condor_release` is used to release the the cluster id of a Late Materialization job factory, and materialization was paused because of a previous use of *condor_hold*, a `Job Materialization Resumed` event will be written to the UserLog of the Cluster ad.

Limitations

Currently, not all features of *condor_submit* will work with late materialization. The following limitations apply:

- Only a single `Queue` statement is allowed, lines from the submit file after the first `Queue` statement will be ignored.
- the `$RANDOM_INTEGER` and `$RANDOM_CHOICE` macro functions will expand at submit time to produce the Cluster ad, but these macro functions will not be included in the *submit digest* and so will have the same value for all jobs.
- Spooling of input files does not work with late materialization.

Displaying the Factory

condor_q can be use to show late materialization job factories in the *condor_schedd* by using the `-factory` option.

```
> condor_q -factory
-- Schedd: submit.example.org : <192.168.101.101:9618?... @ 12/01/20 13:35:00
ID     OWNER          SUBMITTED  LIMIT PRESNT   RUN    IDLE   HOLD NEXTID MODE DIGEST
77.    bob         12/01  13:30  15000    130     30     80     20   1230      /var/lib/
↪condor/spool/77/condor_submit.77.digest
```

The factory above shows that 30 jobs are currently running, 80 are idle, 20 are held and that the next job to materialize will be job `77.1230`. The total of Idle + Held jobs is 100, which is equal to the `max_idle` value specified in the submit file.

The path to the *submit digest* file is shown. This file is used to reload the factory when the *condor_schedd* is restarted. If the factory is unable to materialize jobs because of an error, the `MODE` field will show `Held` or `Errs` to indicate there is a problem. `Errs` indicates a problem reloading the factory, `Held` indicates a problem materializing jobs.

In case of a factory problem, use `condor_q -factory -long` to see the the factory information and the `JobMaterializePauseReason` attribute.

Removing a Factory

The Late materialization job factory will be remove from the schedd automatically once all of the jobs have materialized and completed. To remove the factory without first completing all of the jobs use *condor_rm* with the ClusterId of the factory as the argument.

Editing a Factory

The *submit digest* for a Late Materialization job factory cannot be changed after submission, but the Cluster ad for the factory can be edited using *condor_qedit*. Any *condor_qedit* command that has the ClusterId as a edit target will edit all currently materialized jobs, as well as editing the Cluster ad so that all jobs that materialize in the future will also be edited.

4.4 Submitting Jobs Without a Shared File System: HTCondor's File Transfer Mechanism

HTCondor works well without a shared file system between the submit machines and the worker nodes. The HTCondor file transfer mechanism allows the user to explicitly select which input files are transferred to the worker node before the job starts. HTCondor will transfer these files, potentially delaying this transfer request, if starting the transfer right away would overload the access point. Queueing requests like this prevents the crashes so common with too-busy shared file servers. These input files are placed into a scratch directory on the worker node, which is the starting current directory of the job. When the job completes, by default, HTCondor detects any newly-created files at the top level of this sandbox directory, and transfers them back to the submitting machine. The input sandbox is what we call the executable and all the declared input files of a job. The set of all files created by the job is the output sandbox.

4.4.1 Specifying If and When to Transfer Files

To enable the file transfer mechanism, place this command in the job's submit description file: **should_transfer_files**

```
should_transfer_files = YES
```

Setting the **should_transfer_files** command explicitly enables or disables the file transfer mechanism. The command takes on one of three possible values:

1. YES: HTCondor transfers the input sandbox from the access point to the execute machine. The output sandbox is transferred back to the access point. The command **when_to_transfer_output** . controls when the output sandbox is transferred back, and what directory it is stored in.
2. IF_NEEDED: HTCondor only transfers sandboxes when the job is matched with a machine in a different `FileSystemDomain` than the one the access point belongs to, as if should_transfer_files = YES. If the job is matched with a machine in the same `FileSystemDomain` as the submitting machine, HTCondor will not transfer files and relies on the shared file system.
3. NO: HTCondor's file transfer mechanism is disabled. In this case is is the responsibility of the user to ensure that all data used by the job is accessible on the remote worker node.

The **when_to_transfer_output** command tells HTCondor when output files are to be transferred back to the access point. The command takes on one of three possible values:

1. `ON_EXIT` (the default): HTCondor transfers the output sandbox back to the access point only when the job exits on its own. If the job is preempted or removed, no files are transferred back.
2. `ON_EXIT_OR_EVICT`: HTCondor behaves the same as described for the value ON_EXIT when the job exits on its own. However, each time the job is evicted from a machine, the output sandbox is transferred back to the access point and placed under the **SPOOL** directory. eviction time. Before the job starts running again, the former output sandbox is copied to the job's new remote scratch directory.

   If **transfer_output_files** is specified, this list governs which files are transferred back at eviction time. If a file listed in **transfer_output_files** does not exist at eviction time, the job will go on hold.

   The purpose of saving files at eviction time is to allow the job to resume from where it left off.
3. `ON_SUCCESS`: HTCondor transfers files like `ON_EXIT`, but only if the job succeeds, as defined by the `success_exit_code` submit command. The `success_exit_code` command must be used, even for the default exit code of 0. (See the *condor_submit* man page.)

The default values for these two submit commands make sense as used together. If only **should_transfer_files** is set, and set to the value `NO`, then no output files will be transferred, and the value of **when_to_transfer_output** is

irrelevant. If only **when_to_transfer_output** is set, and set to the value `ON_EXIT_OR_EVICT`, then the default value for an unspecified **should_transfer_files** will be `YES`.

Note that the combination of

```
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT_OR_EVICT
```

would produce undefined file access semantics. Therefore, this combination is prohibited by *condor_submit*.

4.4.2 Specifying What Files to Transfer

If the file transfer mechanism is enabled, HTCondor will transfer the following files before the job is run on a remote machine as the input sandbox:

1. the executable, as defined with the **executable** command
2. the input, as defined with the **input** command
3. any jar files, for the **java** universe, as defined with the **jar_files** command

If the job requires other input files, the submit description file should have the **transfer_input_files** command. This comma-separated list specifies any other files, URLs, or directories that HTCondor is to transfer to the remote scratch directory, to set up the execution environment for the job before it is run. These files are placed in the same directory as the job's executable. For example:

```
executable = my_program
input = my_input
should_transfer_files = YES
transfer_input_files = file1,file2
```

This example explicitly enables the file transfer mechanism. By default, HTCondor will transfer the executable (`my_program`) and the file specified by the input command (`my_input`). The files `file1` and `file2` are also transferred, by explicit user instruction.

If the file transfer mechanism is enabled, HTCondor will transfer the following files from the execute machine back to the access point after the job exits, as the output sandbox.

1. the output file, as defined with the **output** command
2. the error file, as defined with the **error** command
3. any files created by the job in the remote scratch directory.

A path given for **output** and **error** commands represents a path on the access point. If no path is specified, the directory specified with **initialdir** is used, and if that is not specified, the directory from which the job was submitted is used. At the time the job is submitted, zero-length files are created on the access point, at the given path for the files defined by the **output** and **error** commands. This permits job submission failure, if these files cannot be written by HTCondor.

To restrict the output files or permit entire directory contents to be transferred, specify the exact list with **transfer_output_files** . When this comma separated list is defined, and any of the files or directories do not exist as the job exits, HTCondor considers this an error, and places the job on hold. Setting **transfer_output_files** to the empty string ("") means no files are to be transferred. When this list is defined, automatic detection of output files created by the job is disabled. Paths specified in this list refer to locations on the execute machine. The naming and placement of files and directories relies on the term base name. By example, the path `a/b/c` has the base name `c`. It is the file name or directory name with all directories leading up to that name stripped off. On the access point, the transferred files or directories are named using only the base name. Therefore, each output file or directory must have a different name, even if they originate from different paths.

If only a subset of the output sandbox should be transferred, the subset is specified by further adding a submit command of the form:

```
transfer_output_files = file1, file2
```

Here are examples of file transfer with HTCondor. Assume that the job produces the following structure within the remote scratch directory:

```
o1
o2
d1 (directory)
    o3
    o4
```

If the submit description file sets

```
transfer_output_files = o1,o2,d1
```

then transferred back to the access point will be

```
o1
o2
d1 (directory)
    o3
    o4
```

Note that the directory `d1` and all its contents are specified, and therefore transferred. If the directory `d1` is not created by the job before exit, then the job is placed on hold. If the directory `d1` is created by the job before exit, but is empty, this is not an error.

If, instead, the submit description file sets

```
transfer_output_files = o1,o2,d1/o3
```

then transferred back to the access point will be

```
o1
o2
o3
```

Note that only the base name is used in the naming and placement of the file specified with `d1/o3`.

4.4.3 File Paths for File Transfer

The file transfer mechanism specifies file names or URLs on the file system of the access point and file names on the execute machine. Care must be taken to know which machine, submit or execute, is referencing the file.

Files in the **transfer_input_files** command are specified as they are accessed on the access point. The job, as it executes, accesses files as they are found on the execute machine.

There are four ways to specify files and paths for **transfer_input_files** :

1. Relative to the current working directory as the job is submitted, if the submit command **initialdir** is not specified.
2. Relative to the initial directory, if the submit command **initialdir** is specified.
3. Absolute file paths.

4. As an URL, which should be accessible by the execute machine.

Before executing the program, HTCondor copies the input sandbox into a remote scratch directory on the execute machine, where the program runs. Therefore, the executing program must access input files relative to its working directory. Because all files and directories listed for transfer are placed into a single, flat directory, inputs must be uniquely named to avoid collision when transferred.

A job may instead set `preserve_relative_paths` (to `True`), in which case the relative paths of transferred files are preserved. For example, although the input list `dirA/file1, dirB/file1` would normally result in a collision, instead HTCondor will create the directories `dirA` and `dirB` in the input sandbox, and each will get its corresponding version of `file1`.

Both relative and absolute paths may be used in **transfer_output_files** . Relative paths are relative to the job's remote scratch directory on the execute machine. When the files and directories are copied back to the access point, they are placed in the job's initial working directory as the base name of the original path. An alternate name or path may be specified by using **transfer_output_remaps** .

The `preserve_relative_paths` command also applies to relative paths specified in **transfer_output_files** (if not remapped).

A job may create files outside the remote scratch directory but within the file system of the execute machine, in a directory such as `/tmp`, if this directory is guaranteed to exist and be accessible on all possible execute machines. However, HTCondor will not automatically transfer such files back after execution completes, nor will it clean up these files.

Here are several examples to illustrate the use of file transfer. The program executable is called *my_program*, and it uses three command-line arguments as it executes: two input file names and an output file name. The program executable and the submit description file for this job are located in directory `/scratch/test`.

Here is the directory tree as it exists on the access point, for all the examples:

```
/scratch/test (directory)
      my_program.condor (the submit description file)
      my_program (the executable)
      files (directory)
          logs2 (directory)
          in1 (file)
          in2 (file)
      logs (directory)
```

Example 1

This first example explicitly transfers input files. These input files to be transferred are specified relative to the directory where the job is submitted. An output file specified in the **arguments** command, `out1`, is created when the job is executed. It will be transferred back into the directory `/scratch/test`.

```
# file name:  my_program.condor
# HTCondor submit description file for my_program
executable      = my_program
universe        = vanilla
error           = logs/err.$(cluster)
output          = logs/out.$(cluster)
log             = logs/log.$(cluster)

should_transfer_files = YES
transfer_input_files = files/in1,files/in2

arguments       = in1 in2 out1
```

(continues on next page)

(continued from previous page)

```
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

The log file is written on the access point, and is not involved with the file transfer mechanism.

Example 2

This second example is identical to Example 1, except that absolute paths to the input files are specified, instead of relative paths to the input files.

```
# file name:  my_program.condor
# HTCondor submit description file for my_program
executable      = my_program
universe        = vanilla
error           = logs/err.$(cluster)
output          = logs/out.$(cluster)
log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1,/scratch/test/files/in2

arguments       = in1 in2 out1

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

Example 3

This third example illustrates the use of the submit command **initialdir** , and its effect on the paths used for the various files. The expected location of the executable is not affected by the **initialdir** command. All other files (specified by **input** , **output** , **error** , **transfer_input_files** , as well as files modified or created by the job and automatically transferred back) are located relative to the specified **initialdir** . Therefore, the output file, `out1`, will be placed in the files directory. Note that the `logs2` directory exists to make this example work correctly.

```
# file name:  my_program.condor
# HTCondor submit description file for my_program
executable      = my_program
universe        = vanilla
error           = logs2/err.$(cluster)
output          = logs2/out.$(cluster)
log             = logs2/log.$(cluster)

initialdir      = files

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
```

(continues on next page)

(continued from previous page)

```
transfer_input_files = in1,in2

arguments       = in1 in2 out1

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

Example 4 - Illustrates an Error

This example illustrates a job that will fail. The files specified using the **transfer_input_files** command work correctly (see Example 1). However, relative paths to files in the **arguments** command cause the executing program to fail. The file system on the submission side may utilize relative paths to files, however those files are placed into the single, flat, remote scratch directory on the execute machine.

```
# file name:  my_program.condor
# HTCondor submit description file for my_program
executable      = my_program
universe        = vanilla
error           = logs/err.$(cluster)
output          = logs/out.$(cluster)
log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1,files/in2

arguments       = files/in1 files/in2 files/out1

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

This example fails with the following error:

```
err: files/out1: No such file or directory.
```

Example 5 - Illustrates an Error

As with Example 4, this example illustrates a job that will fail. The executing program's use of absolute paths cannot work.

```
# file name:  my_program.condor
# HTCondor submit description file for my_program
executable      = my_program
universe        = vanilla
error           = logs/err.$(cluster)
output          = logs/out.$(cluster)
log             = logs/log.$(cluster)
```

(continues on next page)

(continued from previous page)

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1, /scratch/test/files/in2

arguments = /scratch/test/files/in1 /scratch/test/files/in2 /scratch/test/files/out1

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

The job fails with the following error:

```
err: /scratch/test/files/out1: No such file or directory.
```

Example 6

This example illustrates a case where the executing program creates an output file in a directory other than within the remote scratch directory that the program executes within. The file creation may or may not cause an error, depending on the existence and permissions of the directories on the remote file system.

The output file `/tmp/out1` is transferred back to the job's initial working directory as `/scratch/test/out1`.

```
# file name:  my_program.condor
# HTCondor submit description file for my_program
executable      = my_program
universe        = vanilla
error           = logs/err.$(cluster)
output          = logs/out.$(cluster)
log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT

transfer_input_files = files/in1,files/in2
transfer_output_files = /tmp/out1

arguments       = in1 in2 /tmp/out1
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K


queue
```

4.4.4 Dataflow Jobs

A **dataflow job** is a job that might not need to run because its desired outputs already exist. To skip such a job, add the following line to your submit file:

```
skip_if_dataflow = True
```

A dataflow job meets any of the following criteria:

- Output files exist, are newer than input files
- Execute file is newer than input files
- Standard input file is newer than input files

Skipping dataflow jobs can potentially save large amounts of time in long-running workflows.

4.4.5 Public Input Files

There are some cases where HTCondor's file transfer mechanism is inefficient. For jobs that need to run a large number of times, the input files need to get transferred for every job, even if those files are identical. This wastes resources on both the access point and the network, slowing overall job execution time.

Public input files allow a user to specify files to be transferred over a publicly-available HTTP web service. A system administrator can then configure caching proxies, load balancers, and other tools to dramatically improve performance. Public input files are not available by default, and need to be explicitly enabled by a system administrator.

To specify files that use this feature, the submit file should include a **public_input_files** command. This comma-separated list specifies files which HTCondor will transfer using the HTTP mechanism. For example:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = file1,file2
public_input_files = public_data1,public_data2
```

Similar to the regular **transfer_input_files** , the files specified in **public_input_files** can be relative to the submit directory, or absolute paths. You can also specify an **initialDir** , and *condor_submit* will look for files relative to that directory. The files must be world-readable on the file system (files with permissions set to 0644, directories with permissions set to 0755).

Lastly, all files transferred using this method will be publicly available and world-readable, so this feature should not be used for any sensitive data.

4.4.6 Behavior for Error Cases

This section describes HTCondor's behavior for some error cases in dealing with the transfer of files.

Disk Full on Execute Machine
When transferring any files from the access point to the remote scratch directory, if the disk is full on the execute machine, then the job is place on hold.

Error Creating Zero-Length Files on Submit Machine
As a job is submitted, HTCondor creates zero-length files as placeholders on the access point for the files defined by **output** and **error** . If these files cannot be created, then job submission fails.

This job submission failure avoids having the job run to completion, only to be unable to transfer the job's output due to permission errors.

Error When Transferring Files from Execute Machine to Submit Machine

When a job exits, or potentially when a job is evicted from an execute machine, one or more files may be transferred from the execute machine back to the machine on which the job was submitted.

During transfer, if any of the following three similar types of errors occur, the job is put on hold as the error occurs.

1. If the file cannot be opened on the access point, for example because the system is out of inodes.
2. If the file cannot be written on the access point, for example because the permissions do not permit it.
3. If the write of the file on the access point fails, for example because the system is out of disk space.

4.4.7 File Transfer Using a URL

Instead of file transfer that goes only between the access point and the execute machine, HTCondor has the ability to transfer files from a location specified by a URL for a job's input file, or from the execute machine to a location specified by a URL for a job's output file(s). This capability requires administrative set up, as described in the *Third Party/Delegated file and credential transfer* section.

URL file transfers work in most HTCondor job universes, but not grid, local or scheduler. HTCondor's file transfer mechanism must be enabled. Therefore, the submit description file for the job will define both **should_transfer_files** and **when_to_transfer_output** . In addition, the URL for any files specified with a URL are given in the **transfer_input_files** command. An example portion of the submit description file for a job that has a single file specified with a URL:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = http://www.full.url/path/to/filename
```

The destination file is given by the file name within the URL.

For the transfer of the entire contents of the output sandbox, which are all files that the job creates or modifies, HTCondor's file transfer mechanism must be enabled. In this sample portion of the submit description file, the first two commands explicitly enable file transfer, and the added **output_destination** command provides both the protocol to be used and the destination of the transfer.

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
output_destination = urltype://path/to/destination/directory
```

Note that with this feature, no files are transferred back to the submit machine. This does not interfere with the streaming of output.

Uploading to URLs using output file remaps

File transfer plugins now support uploads as well as downloads. The `transfer_output_remaps` attribute can additionally be used to upload files to specific URLs when a job completes. To do this, set the destination for an output file to a URL instead of a filename. For example:

```
transfer_output_remaps = "myresults.dat = http://destination-server.com/myresults.dat"
```

We use a HTTP PUT request to perform the upload, so the user is responsible for making sure that the destination server accepts PUT requests (which is usually disabled by default).

Passing a credential for URL file transfers

Some files served over HTTPS will require a credential in order to download. Each credential `cred` should be placed in a file in `$_CONDOR_CREDS/cred.use`. Then in order to use that credential for a download, append its name to the beginning of the URL protocol along with a + symbol. For example, to download the file https://download.com/bar using the `cred` credential, specify the following in the submit file:

```
transfer_input_files = cred+https://download.com/bar
```

If your credential file has an underscore in it, the underscore must be replaced in the `transfer_input_files` URL with a ".", e.g. for `$_CONDOR_CREDS/cred_local.use`:

```
transfer_input_files = cred.local+https://download.com/bar
```

Otherwise, the credential file must have a name that only contains alphanumeric characters (a-z, A-Z, 0-9) and/or -, except for the `.` in the `.use` extension.

If you're using a token from an OAuth service provider, the credential will be named based on the OAuth provider. For example, if your submit file has `use_oauth_services = mytokens`, you can request files using that token by doing:

```
use_oauth_services = mytokens

transfer_input_files = mytokens+https://download.com/bar
```

If you add an optional handle to the token name, append the handle name to the token name in the URL with a ".":

```
use_oauth_services = mytokens
mytokens_oauth_permissions_personal =
mytokens_oauth_permissions_group =

transfer_input_files =␣
↪mytokens.personal+https://download.com/bar, mytokens.group+https://download.com/foo
```

Note that in the above token-with-a-handle case, the token files will be stored in the job environment at `$_CONDOR_CREDS/mytokens_personal.use` and `$_CONDOR_CREDS/mytokens_group.use`.

Transferring files using file transfer plugins

HTCondor comes with file transfer plugins that can communicate with Box.com, Google Drive, Stash Cache, OSDF, and Microsoft OneDrive. Using one of these plugins requires that the HTCondor pool administrator has set up the mechanism for HTCondor to gather credentials for the desired service, and requires that your submit file contains the proper commands to obtain credentials from the desired service (see *Jobs That Require Credentials*).

To use a file transfer plugin, substitute `https` in a transfer URL with the service name (`box` for Box.com, `stash` for Stash Cache, `osdf` for OSDF, `gdrive` for Google Drive, and `onedrive` for Microsoft OneDrive) and reference a file path starting at the root directory of the service. For example, to download `bar.txt` from a Box.com account where `bar.txt` is in the `foo` folder, use:

```
use_oauth_services = box
transfer_input_files = box://foo/bar.txt
```

If your job requests multiple credentials from the same service, use `<handle>+<service>://path/to/file` to reference each specific credential. For example, for a job that uses Google Drive to download `public_files/input.txt` from one account (`public`) and to upload `output.txt` to `my_private_files/output.txt` on a second account (`private`):

```
use_oauth_services = gdrive
gdrive_oauth_permissions_public =
gdrive_oauth_permissions_private =

transfer_input_files = public+gdrive://public_files/input.txt
transfer_output_remaps = "output.txt = private+gdrive://my_private_files/output.txt"
```

Transferring files using the S3 protocol

HTCondor supports downloading files from and uploading files to storage servers using the S3 protocol via `s3://` URLs. Downloading or uploading requires a two-part credential: the "access key ID" and the "secret key ID". HTCondor does not transfer these credentials off the submit node; instead, it uses them to construct "pre-signed" `https://` URLs that temporarily allow the bearer access. (Thus, an execute node needs to support `https://` URLs for S3 URLs to work.)

To make use of this feature, you will need to specify the following information in the submit file:

- a file containing your access key ID (and nothing else)
- a file containing your secret access key (and nothing else)
- one or more S3 URLs as input values or output destinations.

See the subsections below for specific examples.

You may (like any other URL) specify an S3 URL in `transfer_input_files`, or as part of a remap in `transfer_output_remaps`. However, HTCondor does not currently support transferring entire buckets or directories. If you specify an `s3://` URL as the `output_destination`, that URL will be used a prefix for each output file's location; if you specify a URL ending a `/`, it will be treated like a directory.

S3 Transfer Recipes

Transferring files to and from Amazon S3

Specify your credential files in the submit file using the attributes `aws_access_key_id_file` and `aws_secret_access_key_file`. Amazon S3 switched from global buckets to region-specific buckets; use the first URL form for the older buckets and the second for newer buckets.

```
aws_access_key_id_file = /home/example/secrets/accessKeyID
aws_secret_access_key_file = /home/example/secrets/secretAccessKey

# For old, non-region-specific buckets.
# transfer_input_files = s3://<bucket-name>/<key-name>,
# transfer_output_remaps = "output.dat = s3://<bucket-name>/<output-key-name>"

# or, for new, region-specific buckets:
transfer_input_files = s3://<bucket-name>.s3.<region>.amazonaws.com/<key>
transfer_output_remaps =␣
↪"output.dat = s3://<bucket-name>.s3.<region>.amazonaws.com/<output-key-name>"

# Optionally, specify a region for S3 URLs which don't include one:
# aws_region = <region>
```

Transferring files to and from Google Cloud Storage

Google Cloud Storage implements an XML API which is interoperable with S3. This requires an extra step of generating HMAC credentials to access Cloud Storage. Google Cloud best practices are to create a Service Account with read/write permission to the bucket. Read HMAC keys for Cloud Storage for more details.

After generating HMAC credentials, they can be used within a job:

```
gs_access_key_id_file = /home/example/secrets/bucket_access_key_id
gs_secret_access_key_file = /home/example/secrets/bucket_secret_access_key
transfer_input_files = gs://<bucket-name>/<input-key-name>
transfer_output_remaps = "output.dat = gs://<bucket-name>/<output-key-name>"
```

If Cloud Storage is configured with Private Service Connect, then use the S3 URL approach with the private Cloud Storage endpoint. e.g.,

```
gs_access_key_id_file = /home/example/secrets/bucket_access_key_id
gs_secret_access_key_file = /home/example/secrets/bucket_secret_access_key
transfer_input_files =␣
↪s3://<cloud-storage-private-endpoint>/<bucket-name>/<input-key-name>
transfer_output_remaps =␣
↪"output.dat = s3://<cloud-storage-private-endpoint>/<bucket-name>/<output-key-name>"
```

Transferring files to and from another provider

Many other companies and institutions offer a service compatible with the S3 protocol. You can access these services using `s3://` URLs and the key files described above.

```
s3_access_key_id_file = /home/example/secrets/accessKeyID
s3_secret_access_key_file = /home/example/secrets/secretAccessKey
transfer_input_files = s3://some.other-s3-provider.org/my-bucket/large-input.file
transfer_output_remaps =␣
↪"large-output.file = s3://some.other-s3-provider.org/my-bucket/large-output.file"
```

If you need to specify a region, you may do so using `aws_region`, despite the name.

4.5 Managing a Job

This section provides a brief summary of what can be done once jobs are submitted. The basic mechanisms for monitoring a job are introduced, but the commands are discussed briefly. You are encouraged to look at the man pages of the commands referred to (located in *Command Reference Manual (man pages)*) for more information.

4.5.1 Checking on the progress of jobs

You can check on your jobs with the *condor_q* command. This command has many options, by default, it displays only your jobs queued in the local scheduler. An example of the output from *condor_q* is

```
$ condor_q

-- Schedd: submit.chtc.wisc.edu : <127.0.0.1:9618?... @ 12/31/69 23:00:00
OWNER    BATCH_NAME    SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS
nemo     batch23       4/22 20:44     _      _      _      1      _ 3671850.0
nemo     batch24       4/22 20:56     _      _      _      1      _ 3673477.0
nemo     batch25       4/22 20:57     _      _      _      1      _ 3673728.0
```

(continues on next page)

(continued from previous page)

```
nemo     batch26       4/23 10:44      _      _      _      1      _ 3750339.0
nemo     batch27       7/2  15:11      _      _      _      _      _ 7594591.0
nemo     batch28       7/10 03:22   4428      3      _      _   4434 7801943.0 ...␣
↪7858552.0
nemo     batch29       7/14 14:18   5074   1182     30     19  80064 7859129.0 ...␣
↪7885217.0
nemo     batch30       7/14 14:18   5172   1088     28     30  58310 7859106.0 ...␣
↪7885192.0

2388 jobs; 0 completed, 1 removed, 58 idle, 2276 running, 53 held, 0 suspended
```

The goal of the HTCondor system is to effectively manage many jobs. As you may have thousands of jobs in a queue, by default *condor_q* summarizes many similar jobs on one line. Depending on the types of your jobs, this output may look a little different.

Often, when you are starting out, and have few jobs, you may want to see one line of output per job. The -nobatch option to *condor_q* does this, and output might look something like:

```
$ condor_q -nobatch

-- Schedd submit.chtc.wisc.edu : <127.0.0.1:9618?...
ID          OWNER         SUBMITTED     RUN_TIME ST PRI SIZE CMD
1297254.0   nemo          5/31 18:05  14+17:40:01 R  0   7.3  condor_dagman
1297255.0   nemo          5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297256.0   nemo          5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297259.0   nemo          5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297261.0   nemo          5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1302278.0   nemo          6/4  12:22   1+00:05:37 I  0   390.6 mdrun_1.sh
1304740.0   nemo          6/5  00:14   1+00:03:43 I  0   390.6 mdrun_1.sh
1304967.0   nemo          6/5  05:08   0+00:00:00 I  0   0.0  mdrun_1.sh

14 jobs; 4 idle, 8 running, 2 held
```

This still only shows your jobs. You can display information about all the users with jobs in this scheduler by adding the -allusers option to *condor_q*.

The output contains many columns of information about the queued jobs. The ST column (for status) shows the status of current jobs in the queue:

R
: The job is currently running.

I
: The job is idle. It is not running right now, because it is waiting for a machine to become available.

H
: The job is the hold state. In the hold state, the job will not be scheduled to run until it is released. See the *condor_hold* and the *condor_release* manual pages.

The RUN_TIME time reported for a job is the time that has been committed to the job.

Another useful method of tracking the progress of jobs is through the job event log. The specification of a `log` in the submit description file causes the progress of the job to be logged in a file. Follow the events by viewing the job event log file. Various events such as execution commencement, file transfer, eviction and termination are logged in the file. Also logged is the time at which the event occurred.

When a job begins to run, HTCondor starts up a *condor_shadow* process on the access point. The shadow process is the mechanism by which the remotely executing jobs can access the environment from which it was submitted, such as input and output files.

It is normal for a machine which has submitted hundreds of jobs to have hundreds of *condor_shadow* processes running on the machine. Since the text segments of all these processes is the same, the load on the submit machine is usually not significant. If there is degraded performance, limit the number of jobs that can run simultaneously by reducing the `MAX_JOBS_RUNNING` configuration variable.

You can also find all the machines that are running your job through the *condor_status* command. For example, to find all the machines that are running jobs submitted by `breach@cs.wisc.edu`, type:

```
$ condor_status -constraint 'RemoteUser == "breach@cs.wisc.edu"'

Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

alfred.cs. INTEL    LINUX        Claimed    Busy       0.980  64    0+07:10:02
biron.cs.w INTEL    LINUX        Claimed    Busy       1.000  128   0+01:10:00
cambridge. INTEL    LINUX        Claimed    Busy       0.988  64    0+00:15:00
falcons.cs INTEL    LINUX        Claimed    Busy       0.996  32    0+02:05:03
happy.cs.w INTEL    LINUX        Claimed    Busy       0.988  128   0+03:05:00
istat03.st INTEL    LINUX        Claimed    Busy       0.883  64    0+06:45:01
istat04.st INTEL    LINUX        Claimed    Busy       0.988  64    0+00:10:00
istat09.st INTEL    LINUX        Claimed    Busy       0.301  64    0+03:45:00
...
```

To find all the machines that are running any job at all, type:

```
$ condor_status -run

Name       Arch     OpSys        LoadAv RemoteUser           ClientMachine

adriana.cs INTEL    LINUX        0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
alfred.cs. INTEL    LINUX        0.980  breach@cs.wisc.edu   neufchatel.cs.w
amul.cs.wi X86_64   LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
anfrom.cs. X86_64   LINUX        1.023  ashoks@jules.ncsa.ui jules.ncsa.uiuc
anthrax.cs INTEL    LINUX        0.285  hepcon@cs.wisc.edu   chevre.cs.wisc.
astro.cs.w INTEL    LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
aura.cs.wi X86_64   WINDOWS      0.996  nice-user.condor@cs. chevre.cs.wisc.
balder.cs. INTEL    WINDOWS      1.000  nice-user.condor@cs. chevre.cs.wisc.
bamba.cs.w INTEL    LINUX        1.574  dmarino@cs.wisc.edu  riola.cs.wisc.e
bardolph.c INTEL    LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
...
```

4.5.2 Peeking in on a running job's output files

When a job is running, you may be curious about any output it has created. The **condor_tail** command can copy output files from a running job on a remote machine back to the access point. **condor_tail** uses the same networking stack as HTCondor proper, so it will work if the execute machine is behind a firewall. Simply run, where xx.yy is the job id of a running job:

```
$ condor_tail xx.yy
```

or

```
$ condor_tail -f xx.yy
```

to continuously follow the standard output. To copy a different file, run

```
$ condor_tail xx.yy name_of_output_file
```

4.5.3 Starting an interactive shell next to a running job on a remote machine

condor_ssh_to_job is a very powerful command, but is not available on all platforms, or all installations. Some administrators disable it, so check with your local site if it does not appear to work. **condor_ssh_to_job** takes the job id of a running job as an argument, and establishes a shell running on the node next to the job. The environment of this shell is a similar to the job as possible. Users of **condor_ssh_to_job** can look at files, attach to their job with the debugger and otherwise inspect the job.

4.5.4 Removing a job from the queue

A job can be removed from the queue at any time by using the *condor_rm* command. If the job that is being removed is currently running, the job is killed, and its queue entry is removed. The following example shows the queue of jobs before and after a job is removed.

```
$ condor_q -nobatch

-- Schedd: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   raman           4/11 14:37   0+00:00:00 R  0   1.4  sleepy
 132.0   raman           4/11 16:57   0+00:00:00 R  0   1.4  hello

2 jobs; 1 idle, 1 running, 0 held

$ condor_rm 132.0
Job 132.0 removed.

$ condor_q -nobatch

-- Schedd: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   raman           4/11 14:37   0+00:00:00 R  0   1.4  sleepy

1 jobs; 1 idle, 0 running, 0 held
```

4.5.5 Placing a job on hold

A job in the queue may be placed on hold by running the command *condor_hold*. A job in the hold state remains in the hold state until later released for execution by the command *condor_release*.

Use of the *condor_hold* command causes a hard kill signal to be sent to a currently running job (one in the running state).

Jobs that are running when placed on hold will start over from the beginning when released.

The *condor_hold* and the *condor_release* manual pages contain usage details.

4.5.6 Changing the priority of jobs

In addition to the priorities assigned to each user, HTCondor also provides each user with the capability of assigning priorities to each submitted job. These job priorities are local to each queue and can be any integer value, with higher values meaning better priority.

The default priority of a job is 0, but can be changed using the *condor_prio* command. For example, to change the priority of a job to -15,

```
$ condor_q -nobatch raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  0   0.3  hello

1 jobs; 1 idle, 0 running, 0 held

$ condor_prio -p -15 126.0

$ condor_q -nobatch raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  -15 0.3  hello

1 jobs; 1 idle, 0 running, 0 held
```

It is important to note that these job priorities are completely different from the user priorities assigned by HTCondor. Job priorities do not impact user priorities. They are only a mechanism for the user to identify the relative importance of jobs among all the jobs submitted by the user to that specific queue.

4.5.7 Why is the job not running?

Users occasionally find that their jobs do not run. There are many possible reasons why a specific job is not running. The following prose attempts to identify some of the potential issues behind why a job is not running.

At the most basic level, the user knows the status of a job by using *condor_q* to see that the job is not running. By far, the most common reason (to the novice HTCondor job submitter) why the job is not running is that HTCondor has not yet been through its periodic negotiation cycle, in which queued jobs are assigned to machines within the pool and begin their execution. This periodic event occurs by default once every 5 minutes, implying that the user ought to wait a few minutes before searching for reasons why the job is not running.

Further inquiries are dependent on whether the job has never run at all, or has run for at least a little bit.

For jobs that have never run, many problems can be diagnosed by using the **-analyze** option of the *condor_q* command. Here is an example; running *condor_q* 's analyzer provided the following information:

```
$ condor_q -analyze 27497829

-- Submitter: s1.chtc.wisc.edu : <128.104.100.43:9618?sock=5557_e660_3> : s1.chtc.wisc.
↪edu
User priority for ei@chtc.wisc.edu is not available, attempting to analyze without it.
---
27497829.000:  Run analysis summary.  Of 5257 machines,
   5257 are rejected by your job's requirements
      0 reject your job because of their own requirements
      0 match and are already running your jobs
      0 match but are serving other users
      0 are available to run your job
        No successful match recorded.
        Last failed match: Tue Jun 18 14:36:25 2013

        Reason for last match failure: no match found

WARNING:  Be advised:
   No resources matched request's constraints

The Requirements expression for your job is:

    ( OpSys == "OSX" ) && ( TARGET.Arch == "X86_64" ) &&
    ( TARGET.Disk >= RequestDisk ) && ( TARGET.Memory >= RequestMemory ) &&
    ( ( TARGET.HasFileTransfer ) || ( TARGET.FileSystemDomain == MY.FileSystemDomain ) )


Suggestions:
    Condition                         Machines Matched Suggestion
    ---------                         ---------------- ----------
1   ( target.OpSys == "OSX" )         0                MODIFY TO "LINUX"
2   ( TARGET.Arch == "X86_64" )       5190
3   ( TARGET.Disk >= 1 )              5257
4   ( TARGET.Memory >= ifthenelse(MemoryUsage isnt undefined,MemoryUsage,1) )
                                      5257
5   ( ( TARGET.HasFileTransfer ) || ( TARGET.FileSystemDomain == "submit-1.chtc.wisc.edu
↪" ) )
                                      5257
```

This example also shows that the job does not run because the platform requested, Mac OS X, is not available on any of the machines in the pool. Recall that unless informed otherwise in the **Requirements** expression in the submit description file, the platform requested for an execute machine will be the same as the platform where *condor_submit* is run to submit the job. And, while Mac OS X is a Unix-type operating system, it is not the same as Linux, and thus will not match with machines running Linux.

While the analyzer can diagnose most common problems, there are some situations that it cannot reliably detect due to the instantaneous and local nature of the information it uses to detect the problem. Thus, it may be that the analyzer reports that resources are available to service the request, but the job still has not run. In most of these situations, the delay is transient, and the job will run following the next negotiation cycle.

A second class of problems represents jobs that do or did run, for at least a short while, but are no longer running. The first issue is identifying whether the job is in this category. The *condor_q* command is not enough; it only tells the current state of the job. The needed information will be in the **log** file or the **error** file, as defined in the submit description file for the job. If these files are not defined, then there is little hope of determining if the job ran at all. For

a job that ran, even for the briefest amount of time, the **log** file will contain an event of type 1, which will contain the string Job executing on host.

A job may run for a short time, before failing due to a file permission problem. The log file used by the *condor_shadow* daemon will contain more information if this is the problem. This log file is associated with the machine on which the job was submitted. The location and name of this log file may be discovered on the submitting machine, using the command

```
$ condor_config_val SHADOW_LOG
```

4.5.8 Job in the Hold State

Should HTCondor detect something about a job that would prevent it from ever running successfully, say, because the executable doesn't exist, or input files are missing, HTCondor will put the job in Hold state. A job in the Hold state will remain in the queue, and show up in the output of the *condor_q* command, but is not eligible to run. The job will stay in this state until it is released or removed. Users may also hold their jobs manually with the *condor_hold* command.

A table listing the reasons why a job may be held is at the *Job ClassAd Attributes* section. A string identifying the reason that a particular job is in the Hold state may be displayed by invoking *condor_q* -hold. For the example job ID 16.0, use:

```
$ condor_q  -hold  16.0
```

This command prints information about the job, including the job ClassAd attribute `HoldReason`.

4.5.9 In the Job Event Log File

In a job event log file are a listing of events in chronological order that occurred during the life of one or more jobs. The formatting of the events is always the same, so that they may be machine readable. Four fields are always present, and they will most often be followed by other fields that give further information that is specific to the type of event.

The first field in an event is the numeric value assigned as the event type in a 3-digit format. The second field identifies the job which generated the event. Within parentheses are the job ClassAd attributes of `ClusterId` value, `ProcId` value, and the node number for parallel universe jobs or a set of zeros (for jobs run under all other universes), separated by periods. The third field is the date and time of the event logging. The fourth field is a string that briefly describes the event. Fields that follow the fourth field give further information for the specific event type.

A complete list of these values is at *Job Event Log Codes* section.

4.5.10 Job Termination

From time to time, and for a variety of reasons, HTCondor may terminate a job before it completes. For instance, a job could be removed (via *condor_rm*), preempted (by a user a with higher priority), or killed (for using more memory than it requested). In these cases, it might be helpful to know why HTCondor terminated the job. HTCondor calls its records of these reasons "Tickets of Execution".

A ticket of execution is usually issued by the *condor_startd*, and includes:

- when the *condor_startd* was told, or otherwise decided, to terminate the job (the `when` attribute);

- who made the decision to terminate, usually a Sinful string (the `who` attribute);
- and what method was employed to command the termination, as both as string and an integer (the `How` and `HowCode` attributes).

The relevant log events include a human-readable rendition of the ToE, and the job ad is updated with the ToE after the usual delay.

As of version 8.9.4, HTCondor only issues ToE in three cases:

- when the job terminates of its own accord (issued by the starter, `HowCode` 0);
- and when the startd terminates the job because it received a `DEACTIVATE_CLAIM` command (`HowCode` 1)
- or a `DEACTIVATE_CLAIM_FORCIBLY` command (`HowCode` 2).

In both cases, HTCondor records the ToE in the job ad. In the event log(s), event 005 (job completion) includes the ToE for the first case, and event 009 (job aborted) includes the ToE for the second and third cases.

Future HTCondor releases will issue ToEs in additional cases and include them in additional log events.

4.5.11 Job Completion

When an HTCondor job completes, either through normal means or by abnormal termination by signal, HTCondor will remove it from the job queue. That is, the job will no longer appear in the output of *condor_q*, and the job will be inserted into the job history file. Examine the job history file with the *condor_history* command. If there is a log file specified in the submit description file for the job, then the job exit status will be recorded there as well, along with other information described below.

By default, HTCondor does not send an email message when the job completes. Modify this behavior with the **notification** command in the submit description file. The message will include the exit status of the job, which is the argument that the job passed to the exit system call when it completed, or it will be notification that the job was killed by a signal. Notification will also include the following statistics (as appropriate) about the job:

Submitted at:
when the job was submitted with *condor_submit*

Completed at:
when the job completed

Real Time:
the elapsed time between when the job was submitted and when it completed, given in a form of `<days> <hours>:<minutes>:<seconds>`

Virtual Image Size:
memory size of the job

Statistics about just the last time the job ran:

Run Time:
total time the job was running, given in the form `<days> <hours>:<minutes>:<seconds>`

Remote User Time:
total CPU time the job spent executing in user mode on remote machines; this does not count time spent on run attempts that were evicted. Given in the form `<days> <hours>:<minutes>:<seconds>`

Remote System Time:
total CPU time the job spent executing in system mode (the time spent at system calls); this does

not count time spent on run attempts that were evicted. Given in the form <days> <hours>:<minutes>:<seconds>

The Run Time accumulated by all run attempts are summarized with the time given in the form <days> <hours>:<minutes>:<seconds>.

And, statistics about the bytes sent and received by the last run of the job and summed over all attempts at running the job are given.

The job terminated event includes the following:

- the type of termination (normal or by signal)
- the return value (or signal number)
- local and remote usage for the last (most recent) run (in CPU-seconds)
- local and remote usage summed over all runs (in CPU-seconds)
- bytes sent and received by the job's last (most recent) run,
- bytes sent and received summed over all runs,
- a report on which partitionable resources were used, if any. Resources include CPUs, disk, and memory; all are lifetime peak values.

Your administrator may have configured HTCondor to report on other resources, particularly GPUs (lifetime average) and GPU memory usage (lifetime peak). HTCondor currently assigns all the usage of a GPU to the job running in the slot to which the GPU is assigned; if the admin allows more than one job to run on the same GPU, or non-HTCondor jobs to use the GPU, GPU usage will be misreported accordingly.

When configured to report GPU usage, HTCondor sets the following two attributes in the job:

GPUsUsage
GPU usage over the lifetime of the job, reported as a fraction of the the maximum possible utilization of one GPU.

GPUsMemoryUsage
Peak memory usage over the lifetime of the job, in megabytes.

4.5.12 Summary of all HTCondor users and their jobs

When jobs are submitted, HTCondor will attempt to find resources to run the jobs. A list of all those with jobs submitted may be obtained through *condor_status* with the *-submitters* option. An example of this would yield output similar to:

```
$ condor_status -submitters

Name                Machine      Running IdleJobs HeldJobs

ballard@cs.wisc.edu  bluebird.c         0       11        0
nice-user.condor@cs. cardinal.c         6      504        0
wright@cs.wisc.edu   finch.cs.w         1        1        0
jbasney@cs.wisc.edu  perdita.cs         0        0        5

                       RunningJobs           IdleJobs           HeldJobs
```

(continues on next page)

(continued from previous page)

```
 ballard@cs.wisc.edu          0          11          0
 jbasney@cs.wisc.edu          0           0          5
nice-user.condor@cs.          6         504          0
  wright@cs.wisc.edu          1           1          0

               Total          7         516          5
```

4.6 Automatically managing a job

While a user can manually manage an HTCondor job in ways described in the previous section, it is often better to give HTCondor policies with which it can automatically manage a job without user intervention.

4.6.1 Automatically rerunning a failed job

By default, when a job exits, HTCondor considers it completed, removes it from the job queue and places it in the history file. If a job exits with a non-zero exit code, this usually means that some error has happened. If this error is ephemeral, a user might want to re-run the job again, to see if the job succeeds on a second invocation. HTCondor can does this automatically with the **max_retries** option in the submit file, to tell HTCondor the maximum number of times to restart the job from scratch. In the rare case where some value other than zero indicates success, a submit file can set **success_exit_code** to the integer value that is considered successful.

```
# Example submit description with max_retries

executable   = myexe
arguments    = SomeArgument

# Retry this job 5 times if non-zero exit code
max_retries = 5

output       = outputfile
error        = errorfile
log          = myexe.log

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

should_transfer_files = yes

queue
```

4.6.2 Automatically removing a job in the queue

HTCondor can automatically remove a job, running or otherwise, from the queue if a given constraint is true. In the submit description file, set **periodic_remove** to a classad expression. When this expression evaluates to true, the scheduler will remove the job, just as if **condor_rm** had run on that job. See *Matchmaking with ClassAds* for information about the classad language and *ClassAd Attributes* for the list of attributes which can be used in these expressions. For example, to automatically remove a job which has been in the queue for more than 100 hours, the submit file could have

```
periodic_remove = (time() - QDate) > (100 * 3600)
```

or, to remove jobs that have been running for more than two hours:

```
periodic_remove = (JobStatus == 2) && (time() - EnteredCurrentStatus) > (2 * 3600)
```

4.6.3 Automatically placing a job on hold

Often, if a job is doing something unexpected, it is more useful to hold the job, rather than remove it. If the problem with the job can be fixed, the job can then be released and started again. Much like the **periodic_remove** command, there is a **periodic_hold** command that works in a similar way, but instead of removing the job, puts the job on hold. Unlike **periodic_remove**, there are additional commands that help to tell the user why the job was placed on hold. **periodic_hold_reason** is a string which is put into the **HoldReason** attribute to explain why we put the job on hold. **periodic_hold_subcode** is an integer that is put into the **HoldReasonSubCode** that is useful for **periodic_release** to examine. Neither **periodic_hold_subcode** nor **periodic_hold_reason** are required, but are good practice to include if **periodic_hold** is defined.

4.6.4 Automatically releasing a held job

In the same way that a job can be automatically held, jobs in the held state can be released with the **periodic_release** command. Often, using a **periodic_hold** with a paired **periodic_release** is a good way to restart a stuck job. Jobs can go into the hold state for many reasons, so best practice, when trying to release a job that was held with **periodic_hold** is to include the **HoldReasonSubCode** in the **periodic_release** expression.

```
periodic_hold = (JobStatus == 2) && (time() - EnteredCurrentStatus) > (2 * 3600)
periodic_hold_reason = "Job ran for more than two hours"
periodic_hold_subcode = 42
periodic_release = (HoldReasonSubCode == 42)
```

4.6.5 Holding a completed job

A job may exit, and HTCondor consider it completed, even though something has gone wrong with the job. A submit file may contain a **on_exit_hold** expression to tell HTCondor to put the job on hold, instead of moving it to the history. A held job informs users that there may have been a problem with the job that should be investigated. For example, if a job should never exit by a signal, the job can be put on hold if it does with

```
on_exit_hold = ExitBySignal == true
```

4.7 How To Debug an Always Idle Job

Sometimes, when you submit a job to HTCondor, it sits idle seemingly forever, *condor_q* shows it in the idle state, when you expect it should start running. This can be frustrating, but there are tools to give visibility so you can debug what is going on.

4.7.1 Jobs that start but are quickly evicted

One possibility is that the job is actually starting, but something goes wrong very quickly after it starts, so the Execution Point evicts the job, and the *condor_schedd* puts it back to idle. *condor_q* would only show it in the "R"unning state for a brief moment, so it is likely that even frequent executions of *condor_q* will show it in the Idle state.

A quick look at the HTCondor job log will help to verify that this is what is happening. Assuming your submit file contains a line like:

```
log          = my_job.log
```

Then you should see a line in my_job.log, assuming that HTCondor assigned the job id of 781.0 to your job (the job id is in parenthesis):

```
000 (781.000.000) 2022-01-30 15:15:35 Job submitted from host: <127.0.0.1:45527?
↪addrs=127.0.0.1-45527>
...
```

Many jobs can share the same job log file, so be sure to find the entries for the job in question. If there is nothing further in this log, this flapping between Running and Idle is not the problem, and you can check items further down this list.

However, if you see repeated entries like

```
001 (781.000.000) 2022-01-30 15:15:36 Job executing on host: <127.0.0.1:42089?addrs=127.
↪0.0.1-42089>
...
007 (781.000.000) 2022-01-30 15:15:37 Shadow exception!
     Error from slot1_2@bat: FATAL:    executable file not found in $PATH
     0  -  Run Bytes Sent By Job
     0  -  Run Bytes Received By Job
 ...
 001 (781.000.000) 2022-01-30 15:15:37 Job executing on host: <127.0.0.1:42089?addrs=127.
↪0.0.1-42089>
 ...
 007 (781.000.000) 2022-01-30 15:15:38 Shadow exception!
 ...
```

Then this flapping is the problem, and you'll need to figure out why. Perhaps a *condor_submit -i* interactive login, and trying to start the job by hand is useful, maybe you'll need to ask a system administrator.

4.7.2 Jobs that don't match any Execution Point

Another common cause of an always-idle job is that the job doesn't match any slot in the pool. Perhaps the memory or disk requested in the submit file is greater than any slot in the pool has. Perhaps your administrator requires jobs to set certain custom attributes to identify them, or for accounting. HTCondor has a tool we call better-analyze that simulates the matching of slots to jobs. It isn't perfect, as it doesn't have full knowledge of the system, but it is easy to run, and can help to quickly narrow down this kind of problems.

```
$ condor_q -better-analyze 781.0
```

Now, as *condor_q -better-analyze* by default, tries to simulate matching this job to all slots in the pool, this can take a while, and generate a lot of output. Sometimes, you are pretty sure that a job should match one particular slot, in that case, you can restrict the matching attempt to that one slot by running

```
$ condor_q -better-analyze 781.0 -machine machine_in_question
```

which will emit information only about a potential match to machine_in_question. If the last few lines of this look like this:

```
The Requirements expression for job 781.0 reduces to these conditions:

 Slots
  Step    Matched  Condition
  -----  --------  ---------
  [0]           1  TARGET.Arch == "X86_64"
  [1]           1  TARGET.OpSys == "LINUX"
  [3]           1  TARGET.Disk >= RequestDisk
  [5]           0  TARGET.Memory >= RequestMemory


  781.007:  Run analysis summary ignoring user priority.  Of 1 machines,
   1 are rejected by your job's requirements
   0 reject your job because of their own requirements
   0 match and are already running your jobs
   0 match but are serving other users
   0 are able to run your job

  WARNING:  Be advised:
     No machines matched the jobs's constraints
```

In this example, RequestMemory is set too high, so the job won't match any machines. Maybe it was a typo. Try setting it lower to see if the job will match. If *condor_q -better-analyze* tells you that some machines do match, then this probably isn't the problem, or, it could be that very few machines in your pool match your job, and you'll just need to wait until they are available.

4.7.3 Not enough priority

Another reason your job isn't running is that other jobs of yours are running, but your priority isn't good enough to allow any more of your jobs running. If this is a problem, the HTCondor *condor_schedd* will run your jobs in the order specified by the Job_Priority submit command. You could give your more important jobs a higher job priority. The command *condor_userprio -all* will show you your current *userprio*, which is what HTCondor uses to calculate your fair share.

4.7.4 Systemic problems

The final case is that you have done nothing wrong, but there is some problem with the system. Maybe a network is down, or a system daemon has crashed, or there is an overload somewhere. If you are an expert, there may be information in the debug logs, usually found in /usr/log/condor. In this case, you may need to consult your system administrator, or ask for help on the *condor-users* email list.

4.8 Services for Running Jobs

HTCondor provides an environment and certain services for running jobs. Jobs can use these services to provide more reliable runs, to give logging and monitoring data for users, and to synchronize with other jobs. Note that different HTCondor job universes may provide different services. The functionality below is available in the vanilla universe, unless otherwise stated.

4.8.1 Environment Variables

An HTCondor job running on a worker node does not, by default, inherit the environment variables from the machine it runs on or the machine it was submitted from. If it did, the environment might change from run to run, or machine to machine, and create non reproducible, difficult to debug problems. Rather, HTCondor is deliberate about what environment variables a job sees, and allows the user to set them in the job description file.

The user may define environment variables for the job with the **environment** command in the submit file. See within the *condor_submit* manual page for more details about this command.

Instead of defining environment variables individually, the entire set of environment variables in the condor_submit's environment can be copied into the job. The **getenv** command does this, as described on the *condor_submit* manual page.

In general, it is preferable to just declare the minimum set of needed environment variables with the **environment** command, as that clearly declares the needed environment variables. If the needed set is not known, the **getenv** command is useful. If the environment is set with both the **environment** command and **getenv** is also set to true, values specified with **environment** override values in the submitter's environment, regardless of the order of the **environment** and **getenv** commands in the submit file.

Commands within the submit description file may reference the environment variables of the submitter. Submit description file commands use $ENV(EnvironmentVariableName) to reference the value of an environment variable.

4.8.2 Extra Environment Variables HTCondor sets for Jobs

HTCondor sets several additional environment variables for each executing job that may be useful.

- `_CONDOR_SCRATCH_DIR` names the directory where the job may place temporary data files. This directory is unique for every job that is run, and its contents are deleted by HTCondor when the job stops running on a machine. When file transfer is enabled, the job is started in this directory.
- `_CONDOR_SLOT` gives the name of the slot (for multicore machines), on which the job is run. On machines with only a single slot, the value of this variable will be 1, just like the `SlotID` attribute in the machine's ClassAd. See the *Policy Configuration for Execution Points and for Access Points* section for more details about configuring multicore machines.
- `_CONDOR_JOB_AD` is the path to a file in the job's scratch directory which contains the job ad for the currently running job. The job ad is current as of the start of the job, but is not updated during the running of the job. The job may read attributes and their values out of this file as it runs, but any changes will not be acted on in any way by HTCondor. The format is the same as the output of the *condor_q* **-l** command. This environment variable may be particularly useful in a USER_JOB_WRAPPER.
- `_CONDOR_MACHINE_AD` is the path to a file in the job's scratch directory which contains the machine ad for the slot the currently running job is using. The machine ad is current as of the start of the job, but is not updated during the running of the job. The format is the same as the output of the *condor_status* **-l** command. Interesting attributes jobs may want to look at from this file include Memory and Cpus, the amount of memory and cpus provisioned for this slot.
- `_CONDOR_JOB_IWD` is the path to the initial working directory the job was born with.
- `_CONDOR_WRAPPER_ERROR_FILE` is only set when the administrator has installed a USER_JOB_WRAPPER. If this file exists, HTCondor assumes that the job wrapper has failed and copies the contents of the file to the StarterLog for the administrator to debug the problem.
- `CUBACORES` `GOMAXPROCS` `JULIA_NUM_THREADS` `MKL_NUM_THREADS` `NUMEXPR_NUM_THREADS` `OMP_NUM_THREADS` `OMP_THREAD_LIMIT` `OPENBLAS_NUM_THREADS` `ROOT_MAX_THREADS` `TF_LOOP_PARALLEL_ITERATIONS` `TF_NUM_THREADS` are set to the number of cpu cores provisioned to this job. Should be at least RequestCpus, but HTCondor may match a job to a bigger slot. Jobs should not spawn more than this number of cpu-bound threads, or their performance will suffer. Many third party libraries like OpenMP obey these environment variables.
- `BATCH_SYSTEM` All job running under a HTCondor starter have the environment variable BATCH_SYSTEM set to the string *HTCondor*. Inspecting this variable allows a job to determine if it is running under HTCondor.
- `X509_USER_PROXY` gives the full path to the X.509 user proxy file if one is associated with the job. Typically, a user will specify **x509userproxy** in the submit description file.

4.8.3 Communicating with the Submit machine via Chirp

HTCondor provides a method for running jobs to read or write information to or from the access point, called "chirp". Chirp allows jobs to

- Write to the job ad in the schedd. This can be used for long-running jobs to write progress information back to the access point, so that a *condor_q* query will reveal how far along a running job is. Or, if a job is listening on a network port, chirp can write the port number to the job ad, so that others can connect to this job.
- Read from the job ad in the schedd. While most information a job needs should be in input files, command line arguments or environment variables, a job can read dynamic information from the schedd's copy of the classad.
- Write a message to the job log. Another place to put progress information is into the job log file. This allows anyone with access to that file to see how much progress a running job has made.

- Read a file from the access point. This allows a job to read a file from the access point at runtime. While file transfer is generally a better approach, file transfer requires the submitter to know the files to be transferred at submit time.
- Write a file to the access point. Again, while file transfer is usually the better choice, with chirp, a job can write intermediate results back to the access point before the job exits.

HTCondor ships a command-line tool, called *condor_chirp* that can do these actions, and provides python bindings so that they can be done natively in Python.

4.8.4 When changes to a job made by chirp take effect

When *condor_chirp* successfully updates a job ad attribute, that change will be reflected in the copy of the job ad in the *condor_schedd* on the access point. However, most job ad attributes are read by the *condor_starter* or *condor_startd* at job start up time, and should chirp change these attributes at run time, it will not impact the running job. In particular, the attributes relating to resource requests, such as RequestCpus, RequestMemory, RequestDisk and RequestGPUS, will not cause any changes to the provisioned resources for a running job. If the job is evicted, and restarts, these new requests will then take effect in the new execution of the job. The same is true for the Requirements expression of a job.

4.8.5 Resource Limitations on a Running Job

Depending on how HTCondor has been configured, the OS platform, and other factors, HTCondor may configure the system a job runs on to prevent a job from using all the resources on a machine. This protects other jobs that may be running on the machine, and the machine itself from being harming by a running job.

Jobs may see

- A private (non-shared) /tmp and /var/tmp directory
- A private (non-shared) /dev/shm
- A limit on the amount of memory they can allocate, above which the job may be placed on hold or evicted by the system.
- A limit on the amount of CPU cores the may use, above which the job may be blocked, and will run very slowly.
- A limit on the amount of scratch disk space the job may use, above which the job may be placed on hold or evicted by the system.

4.9 Priorities and Preemption

HTCondor has two independent priority controls: job priorities and user priorities.

The HTCondor system calculate a "fair share" of machine slots to allocate to each user. Whether each user can use all of these slots depends on a number of factors. For example, if the user's jobs only match to a small number of machines, perhaps the user will be running fewer jobs than allocated. This fair share is based on the *user priority*. Each user can then specify the order in which each of their jobs should be matched and run on the fair share, this is based on the *job priority*.

4.9.1 Job Priority

Job priorities allow a user to sort their own jobs to determine which are tried to be run first. A job priority can be any integer: larger values denote better priority. So, 0 is a better job priority than -3, and 6 is a better than 5. Note that job priorities are computed per user, so that whatever job priorities one user sets has no impact at all on any other user, in terms of how many jobs users can run or in what order. Also, unmatchable high priority jobs do not block lower priority jobs. That is, a priority 10 job will try to be matched before a priority 9 job, but if the priority 10 job doesn't match any slots, HTCondor will keep going, and try the priority 9 job next.

The job priority may be specified in the submit description file by setting

```
priority = 15
```

If no priority is set, the default is 0. See the Dagman section for ways that dagman can automatically set the priority of any or all jobs in a dag.

Each job can be given a distinct priority. For an already queued job, its priority may be changed with the *condor_prio* command; see the example in the *Managing a Job* section, or the *condor_prio* manual page for details. This sets the value of job ClassAd attribute `JobPrio`. *condor_prio* can be called on a running job, but lowering a job priority will not trigger eviction of the running job. The *condor_vacate_job* command can preempt a running job.

A fine-grained categorization of jobs and their ordering is available for experts by using the job ClassAd attributes: `PreJobPrio1`, `PreJobPrio2`, `JobPrio`, `PostJobPrio1`, or `PostJobPrio2`.

4.9.2 User priority

Slots are allocated to users based upon user priority. A lower numerical value for user priority means proportionally better priority, so a user with priority 5 will be allocated 10 times the resources as someone with user priority 50. User priorities in HTCondor can be examined with the *condor_userprio* command (see the *condor_userprio* manual page). HTCondor administrators can set and change individual user priorities with the same utility.

HTCondor continuously calculates the share of available machines that each user should be allocated. This share is inversely related to the ratio between user priorities. For example, a user with a priority of 10 will get twice as many machines as a user with a priority of 20. The priority of each individual user changes according to the number of resources the individual is using. Each user starts out with the best possible priority: 0.5. If the number of machines a user currently has is greater than the user priority, the user priority will worsen by numerically increasing over time. If the number of machines is less then the priority, the priority will improve by numerically decreasing over time. The long-term result is fair-share access across all users. The speed at which HTCondor adjusts the priorities is controlled with the configuration variable `PRIORITY_HALFLIFE` , an exponential half-life value. The default is one day. If a user that has user priority of 100 and is utilizing 100 machines removes all his/her jobs, one day later that user's priority will be 50, and two days later the priority will be 25.

HTCondor enforces that each user gets his/her fair share of machines according to user priority by allocating available machines. Optionally, a pool administrator can configure the system to preempt the running jobs of users who are above their fair share in favor of users who are below their fair share, but this is not the default. For instance, if a low priority user is utilizing all available machines and suddenly a higher priority user submits jobs, HTCondor may vacate jobs belonging to the lower priority user.

User priorities are keyed on `<username>@<domain>`, for example `johndoe@cs.wisc.edu`. The domain name to use, if any, is configured by the HTCondor site administrator. Thus, user priority and therefore resource allocation is not impacted by which machine the user submits from or even if the user submits jobs from multiple machines.

The user priority system can also support backfill or nice jobs (see the *condor_submit* manual page). Nice jobs artificially boost the user priority by ten million just for the nice job. This effectively means that nice jobs will only run on

machines that no other HTCondor job (that is, non-niced job) wants. In a similar fashion, an HTCondor administrator could set the user priority of any specific HTCondor user very high. If done, for example, with a guest account, the guest could only use cycles not wanted by other users of the system.

4.9.3 Details About How HTCondor Jobs Vacate Machines

When HTCondor needs a job to vacate a machine for whatever reason, it sends the job an operating system signal specified in the `KillSig` attribute of the job's ClassAd. The value of this attribute can be specified by the user at submit time by placing the **kill_sig** option in the HTCondor submit description file.

If a program wanted to do some work when asked to vacate a machine, the program may set up a signal handler to handle this signal. This clean up signal is specified with **kill_sig**. Note that the clean up work needs to be quick. If the job takes too long to exit after getting the **kill_sig**, HTCondor sends a SIGKILL signal which immediately terminates the process.

The default value for `KillSig` is SIGTERM, the usual method to nicely terminate a Unix program.

4.10 Job Sets

Multiple jobs that share a common set of input files and/or arguments and/or index values, etc., can be organized and submitted as a **job set**. For example, if you have 10 sets of measurements that you are using as input to two different models, you might consider submitting a job set containing two different modeling jobs that use the same set of input measurement data.

4.10.1 Submitting a job set

Submitting a job set involves creating a job set description file and then using the *htcondor* command-line tool to submit the jobs described in the job set description file to the job queue. For example, if your jobs are described in a file named *my-jobs.set*:

```
$ htcondor jobset submit my-jobs.set
```

A **job set description file** must contain:

1. A **name**,
2. An **iterator**, and
3. At least one **job**.

The **name** of a job set is used to identify the set. Job set names are used to check the status of sets or to remove sets.

The **iterator** of a job set is used to describe the shared values and the values' associated variable names that are used by the jobs in the job set. Multiple iterator types are planned to be supported by HTCondor. As of HTCondor 9.4.0, only the *table* iterator type is available.

The *table* iterator type works similar to the `queue <list of varnames> from <file name or list of items>` syntax used by *condor_submit* description files. A table contains comma-separated columns (one per named variable) and line-separated rows. The table data can either be stored in a separate file and referenced by file name, or it can be stored inside the job set description file itself inside curly brackets (`{ ... }`, see example below).

The job set description file syntax for a *table iterator* is:

```
iterator = table <list of variable names> <table file name>

or

iterator = table <list of variable names> {
    <list of items>
}
```

Suppose you have four *input files*, and each input file is associated with two parameters, *foo* and *bar*, needed by your jobs. An example table in this case could be:

```
input_A.txt,0,0
input_B.txt,0,1
input_C.txt,1,0
input_D.txt,1,1
```

If this table is stored in *input_description.txt*, your iterator would be:

```
iterator = table inputfile,foo,bar input_description.txt
```

Or you could put this table directly inside in the job set description file:

```
iterator = table inputfile,foo,bar {
    input_A.txt,0,0
    input_B.txt,0,1
    input_C.txt,1,0
    input_D.txt,1,1
}
```

Each **job** in a job set is a HTCondor job and is described using the *condor_submit* submit description syntax. A job description can reference one or more of the variables described by the job set iterator. Furthermore, each job description in a job set can have its variables mapped (e.g. `foo=bar` will replace `$(foo)` with `$(bar)`). A job description can either be stored in a separate file and referenced by file name, or it can be stored inside the job set description file itself inside curly brackets (`{ ... }`, see example below).

The job set description file syntax for a *job* is:

```
job [<list of mapped variable names>] <submit file name>

or

job [<list of mapped variable names>] {
    <submit file description>
}
```

Suppose you have two jobs that you want to have use the *inputfile*, *foo*, and *bar* values defined in the *table iterator* example above. And suppose that one of these jobs already has an existing submit description in a file named `my-job.sub`, and this submit file *doesn't* use the *foo* and *bar* variable names but instead uses *x* and *y*. Your *job* descriptions could look like:

```
job x=foo,y=bar my-job.sub

job {
    executable = a.out
```

(continues on next page)

(continued from previous page)

```
    arguments = $(inputfile) $(foo) $(bar)
    transfer_input_files = $(inputfile)
}
```

Note how in the second job above that there is no `queue` statement. Job description queue statements are disregarded when using job sets. Instead, the number of jobs queued are based on the *iterator* of the job set. For the *table iterator*, the number of jobs queued will be the number of rows in the table.

Putting together the examples above, an entire example job set might look like:

```
name = MyJobSet

iterator = table inputfile,foo,bar {
    input_A.txt,0,0
    input_B.txt,0,1
    input_C.txt,1,0
    input_D.txt,1,1
}

job x=foo,y=bar my-job.sub

job {
    executable = a.out
    arguments = $(inputfile) $(foo) $(bar)
    transfer_input_files = $(inputfile)
}
```

Based on this job set description, with two job descriptions (which become two job clusters), you would expect the following output when submitting this job set:

```
$ htcondor jobset submit my-jobs.set
Submitted job set MyJobSet containing 2 job clusters.
```

4.10.2 Listing job sets

You can get a list of your active job sets (i.e. job sets with jobs that are idle, executing, or held) with the command `htcondor jobset list`:

```
$ htcondor jobset list
JOB_SET_NAME
MyJobSet
```

The argument `--allusers` will list active job sets for all users on the current access point:

```
$ htcondor jobset list --allusers
OWNER  JOB_SET_NAME
alice  MyJobSet
bob    AnotherJobSet
```

4.10.3 Checking on the progress of job sets

You can check on your job set with the `htcondor jobset status <job set name>` command.

```
$ htcondor jobset status MyJobSet

MyJobSet currently has 3 jobs idle, 5 jobs running, and 0 jobs completed.
MyJobSet contains:
    Job cluster 1234 with 4 total jobs
    Job cluster 1235 with 4 total jobs
```

4.10.4 Removing a job set

If you realize that there is a problem with a job set or you just do not need the job set to finish computing for whatever reason, you can remove an entire job set with the `htcondor jobset remove <job set name>` command:

```
$ htcondor jobset remove MyJobSet
Removed 8 jobs matching job set MyJobSet for user alice.
```

4.11 Matchmaking with ClassAds

Before you learn about how to submit a job, it is important to understand how HTCondor allocates resources. Understanding the unique framework by which HTCondor matches submitted jobs with machines is the key to getting the most from HTCondor's scheduling algorithm.

HTCondor simplifies job submission by acting as a matchmaker of ClassAds. HTCondor's ClassAds are analogous to the classified advertising section of the newspaper. Sellers advertise specifics about what they have to sell, hoping to attract a buyer. Buyers may advertise specifics about what they wish to purchase. Both buyers and sellers list constraints that need to be satisfied. For instance, a buyer has a maximum spending limit, and a seller requires a minimum purchase price. Furthermore, both want to rank requests to their own advantage. Certainly a seller would rank one offer of $50 dollars higher than a different offer of $25. In HTCondor, users submitting jobs can be thought of as buyers of compute resources and machine owners are sellers.

All machines in a HTCondor pool advertise their attributes, such as available memory, CPU type and speed, virtual memory size, current load average, along with other static and dynamic properties. This machine ClassAd also advertises under what conditions it is willing to run a HTCondor job and what type of job it would prefer. These policy attributes can reflect the individual terms and preferences by which all the different owners have graciously allowed their machine to be part of the HTCondor pool. You may advertise that your machine is only willing to run jobs at night and when there is no keyboard activity on your machine. In addition, you may advertise a preference (rank) for running jobs submitted by you or one of your co-workers.

Likewise, when submitting a job, you specify a ClassAd with your requirements and preferences. The ClassAd includes the type of machine you wish to use. For instance, perhaps you are looking for the fastest floating point performance available. You want HTCondor to rank available machines based upon floating point performance. Or, perhaps you care only that the machine has a minimum of 128 MiB of RAM. Or, perhaps you will take any machine you can get! These job attributes and requirements are bundled up into a job ClassAd.

HTCondor plays the role of a matchmaker by continuously reading all the job ClassAds and all the machine ClassAds, matching and ranking job ads with machine ads. HTCondor makes certain that all requirements in both ClassAds are satisfied.

4.11.1 Inspecting Machine ClassAds with condor_status

Once HTCondor is installed, you will get a feel for what a machine ClassAd does by trying the *condor_status* command. Try the *condor_status* command to get a summary of information from ClassAds about the resources available in your pool. Type *condor_status* and hit enter to see a summary similar to the following:

```
Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

amul.cs.wisc.edu   LINUX      INTEL  Claimed   Busy     0.990  1896  0+00:07:04
slot1@amundsen.cs. LINUX      INTEL  Owner     Idle     0.000  1456  0+00:21:58
slot2@amundsen.cs. LINUX      INTEL  Owner     Idle     0.110  1456  0+00:21:59
angus.cs.wisc.edu  LINUX      INTEL  Claimed   Busy     0.940   873  0+00:02:54
anhai.cs.wisc.edu  LINUX      INTEL  Claimed   Busy     1.400  1896  0+00:03:03
apollo.cs.wisc.edu LINUX      INTEL  Unclaimed Idle     1.000  3032  0+00:00:04
arragon.cs.wisc.ed LINUX      INTEL  Claimed   Busy     0.980   873  0+00:04:29
bamba.cs.wisc.edu  LINUX      INTEL  Owner     Idle     0.040  3032 15+20:10:19
...
```

The *condor_status* command has options that summarize machine ads in a variety of ways. For example,

condor_status -available
: shows only machines which are willing to run jobs now.

condor_status -run
: shows only machines which are currently running jobs.

condor_status -long
: lists the machine ClassAds for all machines in the pool.

Refer to the *condor_status* command reference page for a complete description of the *condor_status* command.

The following shows a portion of a machine ClassAd for a single machine: turunmaa.cs.wisc.edu. Some of the listed attributes are used by HTCondor for scheduling. Other attributes are for information purposes. An important point is that any of the attributes in a machine ClassAd can be utilized at job submission time as part of a request or preference on what machine to use. Additional attributes can be easily added. For example, your site administrator can add a physical location attribute to your machine ClassAds.

```
Machine = "turunmaa.cs.wisc.edu"
FileSystemDomain = "cs.wisc.edu"
Name = "turunmaa.cs.wisc.edu"
CondorPlatform = "$CondorPlatform: x86_rhap_5 $"
Cpus = 1
CondorVersion = "$CondorVersion: 7.6.3 Aug 18 2011 BuildID: 361356 $"
Requirements = START
EnteredCurrentActivity = 1316094896
MyAddress = "<128.105.175.125:58026>"
EnteredCurrentState = 1316094896
Memory = 1897
CkptServer = "pitcher.cs.wisc.edu"
OpSys = "LINUX"
State = "Owner"
START = true
Arch = "INTEL"
Mips = 2634
```

(continues on next page)

(continued from previous page)

```
Activity = "Idle"
StartdIpAddr = "<128.105.175.125:58026>"
TargetType = "Job"
LoadAvg = 0.210000
Disk = 92309744
VirtualMemory = 2069476
TotalSlots = 1
UidDomain = "cs.wisc.edu"
MyType = "Machine"
```

4.12 Choosing an HTCondor Universe

A universe in HTCondor defines an execution environment for a job. HTCondor supports several different universes:

- vanilla
- grid
- java
- scheduler
- local
- parallel
- vm
- container
- docker

The **universe** under which a job runs is specified in the submit description file. If a universe is not specified, the default is vanilla.

The vanilla universe is a good default, for it has the fewest restrictions on the job. The grid universe allows users to submit jobs using HTCondor's interface. These jobs are submitted for execution on grid resources. The java universe allows users to run jobs written for the Java Virtual Machine (JVM). The scheduler universe allows users to submit lightweight jobs to be spawned by the program known as a daemon on the submit host itself. The parallel universe is for programs that require multiple machines for one job. See the *Parallel Applications (Including MPI Applications)* section for more about the Parallel universe. The vm universe allows users to run jobs where the job is no longer a simple executable, but a disk image, facilitating the execution of a virtual machine. Container universe allows the user to specify a container image for one of many possible container runtimes, just as singularity or docker, and condor will run the job in the appropriate container runtimes. The docker universe runs a Docker container as an HTCondor job.

4.12.1 Vanilla Universe

The vanilla universe in HTCondor is intended for most programs. Shell scripts are another case where the vanilla universe is useful.

Access to the job's input and output files is a concern for vanilla universe jobs. One option is for HTCondor to rely on a shared file system, such as NFS or AFS. Alternatively, HTCondor has a mechanism for transferring files on behalf of the user. In this case, HTCondor will transfer any files needed by a job to the execution site, run the job, and transfer the output back to the submitting machine.

4.12.2 Grid Universe

The Grid universe in HTCondor is intended to provide the standard HTCondor interface to users who wish to start jobs intended for remote management systems. *The Grid Universe* section has details on using the Grid universe. The manual page for *condor_submit* has detailed descriptions of the grid-related attributes.

4.12.3 Java Universe

A program submitted to the Java universe may run on any sort of machine with a JVM regardless of its location, owner, or JVM version. HTCondor will take care of all the details such as finding the JVM binary and setting the classpath.

4.12.4 Scheduler Universe

The scheduler universe allows users to submit lightweight jobs to be run immediately, alongside the *condor_schedd* daemon on the submit host itself. Scheduler universe jobs are not matched with a remote machine, and will never be preempted. The job's requirements expression is evaluated against the *condor_schedd* 's ClassAd.

Originally intended for meta-schedulers such as *condor_dagman*, the scheduler universe can also be used to manage jobs of any sort that must run on the submit host.

However, unlike the local universe, the scheduler universe does not use a *condor_starter* daemon to manage the job, and thus offers limited features and policy support. The local universe is a better choice for most jobs which must run on the submit host, as it offers a richer set of job management features, and is more consistent with other universes such as the vanilla universe. The scheduler universe may be retired in the future, in favor of the newer local universe.

4.12.5 Local Universe

The local universe allows an HTCondor job to be submitted and executed with different assumptions for the execution conditions of the job. The job does not wait to be matched with a machine. It instead executes right away, on the machine where the job is submitted. The job will never be preempted. The job's requirements expression is evaluated against the *condor_schedd* 's ClassAd.

4.12.6 Parallel Universe

The parallel universe allows parallel programs, such as MPI jobs, to be run within the opportunistic HTCondor environment. Please see the *Parallel Applications (Including MPI Applications)* section for more details.

4.12.7 VM Universe

HTCondor facilitates the execution of KVM and Xen virtual machines with the vm universe.

Please see the *Virtual Machine Applications* section for details.

4.12.8 Docker Universe

The docker universe runs a docker container on an execute host as a job. Please see the *Docker Universe Applications* section for details.

4.12.9 Container Universe

The container universe runs a container on an execute host as a job. Please see the *Container Universe Jobs* section for details.

4.13 Java Applications

HTCondor allows users to access a wide variety of machines distributed around the world. The Java Virtual Machine (JVM) provides a uniform platform on any machine, regardless of the machine's architecture or operating system. The HTCondor Java universe brings together these two features to create a distributed, homogeneous computing environment.

Compiled Java programs can be submitted to HTCondor, and HTCondor can execute the programs on any machine in the pool that will run the Java Virtual Machine.

The *condor_status* command can be used to see a list of machines in the pool for which HTCondor can use the Java Virtual Machine.

```
$ condor_status -java

Name               JavaVendor Ver    State     Activity LoadAv  Mem  ActvtyTime

adelie01.cs.wisc.e Sun Micros 1.6.0_ Claimed   Busy     0.090   873  0+00:02:46
adelie02.cs.wisc.e Sun Micros 1.6.0_ Owner     Idle     0.210   873  0+03:19:32
slot10@bio.cs.wisc Sun Micros 1.6.0_ Unclaimed Idle     0.000   118  7+03:13:28
slot2@bio.cs.wisc. Sun Micros 1.6.0_ Unclaimed Idle     0.000   118  7+03:13:28
...
```

If there is no output from the *condor_status* command, then HTCondor does not know the location details of the Java Virtual Machine on machines in the pool, or no machines have Java correctly installed.

4.13.1 A Simple Example Java Application

Here is a complete, if simple, example. Start with a simple Java program, `Hello.java`:

```
public class Hello {
    public static void main( String [] args ) {
        System.out.println("Hello, world!\n");
    }
}
```

Build this program using your Java compiler. On most platforms, this is accomplished with the command

```
$ javac Hello.java
```

Submission to HTCondor requires a submit description file. If submitting where files are accessible using a shared file system, this simple submit description file works:

```
####################
#
# Example 1
# Execute a single Java class
#
####################

universe       = java
executable     = Hello.class
arguments      = Hello
output         = Hello.output
error          = Hello.error

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

The Java universe must be explicitly selected.

The main class of the program is given in the **executable** statement. This is a file name which contains the entry point of the program. The name of the main class (not a file name) must be specified as the first argument to the program.

If submitting the job where a shared file system is not accessible, the submit description file becomes:

```
####################
#
# Example 2
# Execute a single Java class,
# not on a shared file system
#
####################

universe       = java
executable     = Hello.class
```

(continues on next page)

(continued from previous page)

```
arguments       = Hello
output          = Hello.output
error           = Hello.error
should_transfer_files = YES
when_to_transfer_output = ON_EXIT

request_cpus    = 1
request_memory = 1024M
request_disk    = 10240K

queue
```

For more information about using HTCondor's file transfer mechanisms, see the *Submitting a Job* section.

To submit the job, where the submit description file is named `Hello.cmd`, execute

```
$ condor_submit Hello.cmd
```

To monitor the job, the commands *condor_q* and *condor_rm* are used as with all jobs.

4.13.2 Less Simple Java Specifications

Specifying more than 1 class file.
For programs that consist of more than one `.class` file, identify the files in the submit description file:

```
executable = Stooges.class
transfer_input_files = Larry.class,Curly.class,Moe.class
```

The **executable** command does not change. It still identifies the class file that contains the program's entry point.

JAR files.
If the program consists of a large number of class files, it may be easier to collect them all together into a single Java Archive (JAR) file. A JAR can be created with:

```
$ jar cvf Library.jar Larry.class Curly.class Moe.class Stooges.class
```

HTCondor must then be told where to find the JAR as well as to use the JAR. The JAR file that contains the entry point is specified with the **executable** command. All JAR files are specified with the **jar_files** command. For this example that collected all the class files into a single JAR file, the submit description file contains:

```
executable = Library.jar
jar_files = Library.jar
```

Note that the JVM must know whether it is receiving JAR files or class files. Therefore, HTCondor must also be informed, in order to pass the information on to the JVM. That is why there is a difference in submit description file commands for the two ways of specifying files (**transfer_input_files** and **jar_files**).

If there are multiple JAR files, the **executable** command specifies the JAR file that contains the program's entry point. This file is also listed with the **jar_files** command:

```
executable = sortmerge.jar
jar_files = sortmerge.jar,statemap.jar
```

Using a third-party JAR file.
As HTCondor requires that all JAR files (third-party or not) be available, specification of a third-party JAR file is no different than other JAR files. If the sortmerge example above also relies on version 2.1 from http://jakarta.apache.org/commons/lang/, and this JAR file has been placed in the same directory with the other JAR files, then the submit description file contains

```
executable = sortmerge.jar
jar_files = sortmerge.jar,statemap.jar,commons-lang-2.1.jar
```

An executable JAR file.
When the JAR file is an executable, specify the program's entry point in the **arguments** command:

```
executable = anexecutable.jar
jar_files  = anexecutable.jar
arguments  = some.main.ClassFile
```

Discovering the main class within a JAR file.
As of Java version 1.4, Java virtual machines have a **-jar** option, which takes a single JAR file as an argument. With this option, the Java virtual machine discovers the main class to run from the contents of the Manifest file, which is bundled within the JAR file. HTCondor's **java** universe does not support this discovery, so before submitting the job, the name of the main class must be identified.

For a Java application which is run on the command line with

```
$ java -jar OneJarFile.jar
```

the equivalent version after discovery might look like

```
$ java -classpath OneJarFile.jar TheMainClass
```

The specified value for TheMainClass can be discovered by unjarring the JAR file, and looking for the MainClass definition in the Manifest file. Use that definition in the HTCondor submit description file. Partial contents of that file Java universe submit file will appear as

```
universe   = java
executable =  OneJarFile.jar
jar_files  = OneJarFile.jar
Arguments  = TheMainClass More-Arguments
queue
```

Packages.
An example of a Java class that is declared in a non-default package is

```
package hpc;

public class CondorDriver
{
 // class definition here
}
```

The JVM needs to know the location of this package. It is passed as a command-line argument, implying the use of the naming convention and directory structure.

Therefore, the submit description file for this example will contain

```
arguments = hpc.CondorDriver
```

JVM-version specific features.
If the program uses Java features found only in certain JVMs, then the Java application submitted to HTCondor must only run on those machines within the pool that run the needed JVM. Inform HTCondor by adding a `requirements` statement to the submit description file. For example, to require version 3.2, add to the submit description file:

```
requirements = (JavaVersion=="3.2")
```

JVM options.
Options to the JVM itself are specified in the submit description file:

```
java_vm_args = -DMyProperty=Value -verbose:gc -Xmx1024m
```

These options are those which go after the java command, but before the user's main class. Do not use this to set the classpath, as HTCondor handles that itself. Setting these options is useful for setting system properties, system assertions and debugging certain kinds of problems.

4.13.3 Chirp I/O

If a job has more sophisticated I/O requirements that cannot be met by HTCondor's file transfer mechanism, then the Chirp facility may provide a solution. Chirp has two advantages over simple, whole-file transfers. First, it permits the input files to be decided upon at run-time rather than submit time, and second, it permits partial-file I/O with results than can be seen as the program executes. However, small changes to the program are required in order to take advantage of Chirp. Depending on the style of the program, use either Chirp I/O streams or UNIX-like I/O functions.

Chirp I/O streams are the easiest way to get started. Modify the program to use the objects `ChirpInputStream` and `ChirpOutputStream` instead of `FileInputStream` and `FileOutputStream`. These classes are completely documented in the HTCondor Software Developer's Kit (SDK). Here is a simple code example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         BufferedReader in = new BufferedReader(
            new InputStreamReader(
               new ChirpInputStream("input")));

         PrintWriter out = new PrintWriter(
            new OutputStreamWriter(
               new ChirpOutputStream("output")));

         while(true) {
            String line = in.readLine();
            if(line==null) break;
```

(continues on next page)

(continued from previous page)

```
            out.println(line);
         }
         out.close();
      } catch( IOException e ) {
         System.out.println(e);
      }
   }
}
```

To perform UNIX-like I/O with Chirp, create a `ChirpClient` object. This object supports familiar operations such as `open`, `read`, `write`, and `close`. Exhaustive detail of the methods may be found in the HTCondor SDK, but here is a brief example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         ChirpClient client = new ChirpClient();
         String message = "Hello, world!\n";
         byte [] buffer = message.getBytes();

         // Note that we should check that actual==length.
         // However, skip it for clarity.

         int fd = client.open("output","wct",0777);
         int actual = client.write(fd,buffer,0,buffer.length);
         client.close(fd);

         client.rename("output","output.new");
         client.unlink("output.new");

      } catch( IOException e ) {
         System.out.println(e);
      }
   }
}
```

Regardless of which I/O style, the Chirp library must be specified and included with the job. The Chirp JAR (`Chirp.jar`) is found in the `lib` directory of the HTCondor installation. Copy it into your working directory in order to compile the program after modification to use Chirp I/O.

```
$ condor_config_val LIB
/usr/local/condor/lib
$ cp /usr/local/condor/lib/Chirp.jar .
```

Rebuild the program with the Chirp JAR file in the class path.

```
$ javac -classpath Chirp.jar:. TestChirp.java
```

The Chirp JAR file must be specified in the submit description file. Here is an example submit description file that works for both of the given test programs:

```
universe = java
executable = TestChirp.class
arguments = TestChirp
jar_files = Chirp.jar
want_io_proxy = True
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

4.14 Parallel Applications (Including MPI Applications)

HTCondor's parallel universe supports jobs that span multiple machines, where the multiple processes within a job must be running concurrently on these multiple machines, perhaps communicating with each other. The parallel universe provides machine scheduling, but does not enforce a particular programming paradigm for the underlying applications. Thus, parallel universe jobs may run under various MPI implementations as well as under other programming environments.

The parallel universe supersedes the mpi universe. The mpi universe eventually will be removed from HTCondor.

4.14.1 How Parallel Jobs Run

Parallel universe jobs are submitted from the machine running the dedicated scheduler. The dedicated scheduler matches and claims a fixed number of machines (slots) for the parallel universe job, and when a sufficient number of machines are claimed, the parallel job is started on each claimed slot.

Each invocation of *condor_submit* assigns a single `ClusterId` for what is considered the single parallel job submitted. The **machine_count** submit command identifies how many machines (slots) are to be allocated. Each instance of the **queue** submit command acquires and claims the number of slots specified by **machine_count**. Each of these slots shares a common job ClassAd and will have the same `ProcId` job ClassAd attribute value.

Once the correct number of machines are claimed, the **executable** is started at more or less the same time on all machines. If desired, a monotonically increasing integer value that starts at 0 may be provided to each of these machines. The macro `$(Node)` is similar to the MPI rank construct. This macro may be used within the submit description file in either the **arguments** or **environment** command. Thus, as the executable runs, it may discover its own `$(Node)` value.

Node 0 has special meaning and consequences for the parallel job. The completion of a parallel job is implied and taken to be when the Node 0 executable exits. All other nodes that are part of the parallel job and that have not yet exited on their own are killed. This default behavior may be altered by placing the line

```
+ParallelShutdownPolicy = "WAIT_FOR_ALL"
```

in the submit description file. It causes HTCondor to wait until every node in the parallel job has completed to consider the job finished.

4.14.2 Parallel Jobs and the Dedicated Scheduler

To run parallel universe jobs, HTCondor must be configured such that machines running parallel jobs are dedicated. Note that dedicated has a very specific meaning in HTCondor: while dedicated machines can run serial jobs, they prefer to run parallel jobs, and dedicated machines never preempt a parallel job once it starts running.

A machine becomes a dedicated machine when an administrator configures it to accept parallel jobs from one specific dedicated scheduler. Note the difference between parallel and serial jobs. While any scheduler in a pool can send serial jobs to any machine, only the designated dedicated scheduler may send parallel universe jobs to a dedicated machine. Dedicated machines must be specially configured. See the *Setting Up for Special Environments* section for a description of the necessary configuration, as well as examples. Usually, a single dedicated scheduler is configured for a pool which can run parallel universe jobs, and this *condor_schedd* daemon becomes the single machine from which parallel universe jobs are submitted.

The following command line will list the execute machines in the local pool which have been configured to use a dedicated scheduler, also printing the name of that dedicated scheduler. In order to run parallel jobs, this name will be defined to be the string "`DedicatedScheduler@`", prepended to the name of the scheduler host.

```
$ condor_status -const '!isUndefined(DedicatedScheduler)' \
      -format "%s\t" Machine -format "%s\n" DedicatedScheduler

  execute1.example.com DedicatedScheduler@submit.example.com
  execute2.example.com DedicatedScheduler@submit.example.com
```

If this command emits no lines of output, then then pool is not correctly configured to run parallel jobs. Make sure that the name of the scheduler is correct. The string after the `@` sign should match the name of the *condor_schedd* daemon, as returned by the command

```
$ condor_status -schedd
```

4.14.3 Submission Examples

Simplest Example

Here is a submit description file for a parallel universe job example that is as simple as possible:

```
##############################################
##  submit description file for a parallel universe job
##############################################
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
log = log
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

This job specifies the **universe** as **parallel**, letting HTCondor know that dedicated resources are required. The **machine_count** command identifies that eight machines are required for this job.

Because no **requirements** are specified, the dedicated scheduler claims eight machines with the same architecture and operating system as the access point. When all the machines are ready, it invokes the */bin/sleep* command, with a command line argument of 30 on each of the eight machines more or less simultaneously. Job events are written to the log specified in the **log** command.

The file transfer mechanism is enabled for this parallel job, such that if any of the eight claimed execute machines does not share a file system with the access point, HTCondor will correctly transfer the executable. This */bin/sleep* example implies that the access point is running a Unix operating system, and the default assumption for submission from a Unix machine would be that there is a shared file system.

Example with Operating System Requirements

Assume that the pool contains Linux machines installed with either a RedHat or an Ubuntu operating system. If the job should run only on RedHat platforms, the requirements expression may specify this:

```
##############################################
##  submit description file for a parallel program
##  targeting RedHat machines
##############################################
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
log = log
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
requirements = (OpSysName == "RedHat")
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

The machine selection may be further narrowed, instead using the `OpSysAndVer` attribute.

```
##############################################
##  submit description file for a parallel program
##  targeting RedHat 6 machines
##############################################
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
log = log
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
requirements = (OpSysAndVer == "RedHat6")
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

Using the `$(Node)` Macro

```
######################################
## submit description file for a parallel program
## showing the $(Node) macro
######################################
universe = parallel
executable = /bin/cat
log = logfile
input = infile.$(Node)
output = outfile.$(Node)
error = errfile.$(Node)
machine_count = 4
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
queue
```

The `$(Node)` macro is expanded to values of 0-3 as the job instances are about to be started. This assigns unique names to the input and output files to be transferred or accessed from the shared file system. The `$(Node)` value is fixed for the entire length of the job.

Differing Requirements for the Machines

Sometimes one machine's part in a parallel job will have specialized needs. These can be handled with a **Requirements** submit command that also specifies the number of needed machines.

```
######################################
## Example submit description file
## with 4 total machines and differing requirements
######################################
universe = parallel
executable = special.exe
machine_count = 1
requirements = ( machine == "machine1@example.com")
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue

machine_count = 3
requirements = ( machine =!= "machine1@example.com")
queue
```

The dedicated scheduler acquires and claims four machines. All four share the same value of `ClusterId`, as this value is associated with this single parallel job. The existence of a second **queue** command causes a total of two `ProcId` values to be assigned for this parallel job. The `ProcId` values are assigned based on ordering within the submit description file. Value 0 will be assigned for the single executable that must be executed on machine1@example.com, and the value 1 will be assigned for the other three that must be executed elsewhere.

Requesting multiple cores per slot

If the parallel program has a structure that benefits from running on multiple cores within the same slot, multi-core slots may be specified.

```
######################################
## submit description file for a parallel program
## that needs 8-core slots
######################################
universe = parallel
executable = foo.sh
log = logfile
input = infile.$(Node)
output = outfile.$(Node)
error = errfile.$(Node)
machine_count = 2
request_cpus = 8
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

This parallel job causes the scheduler to match and claim two machines, where each of the machines (slots) has eight cores. The parallel job is assigned a single `ClusterId` and a single `ProcId`, meaning that there is a single job ClassAd for this job.

The executable, `foo.sh`, is started at the same time on a single core within each of the two machines (slots). It is presumed that the executable will take care of invoking processes that are to run on the other seven CPUs (cores) associated with the slot.

Potentially fewer machines are impacted with this specification, as compared with the request that contains

```
machine_count = 16
request_cpus = 1
```

The interaction of the eight cores within the single slot may be advantageous with respect to communication delay or memory access. But, 8-core slots must be available within the pool.

MPI Applications

MPI applications use a single executable, invoked on one or more machines (slots), executing in parallel. The various implementations of MPI such as Open MPI and MPICH require further framework. HTCondor supports this necessary framework through a user-modified script. This implementation-dependent script becomes the HTCondor executable. The script sets up the framework, and then it invokes the MPI application's executable.

The scripts are located in the `$(RELEASE_DIR)`/etc/examples directory. The script for the Open MPI implementation is `openmpiscript`. The scripts for MPICH implementations are `mp1script` and `mp2script`. An MPICH3 script is not available at this time. These scripts rely on running *ssh* for communication between the nodes of the MPI application. The *ssh* daemon on Unix platforms restricts connections to the approved shells listed in the `/etc/shells` file.

Here is a sample submit description file for an MPICH MPI application:

```
####################################
## Example submit description file
## for MPICH 1 MPI
## works with MPICH 1.2.4, 1.2.5 and 1.2.6
####################################
universe = parallel
executable = mp1script
arguments = my_mpich_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_mpich_linked_executable
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

The **executable** is the `mp1script` script that will have been modified for this MPI application. This script is invoked on each slot or core. The script, in turn, is expected to invoke the MPI application's executable. To know the MPI application's executable, it is the first in the list of **arguments** . And, since HTCondor must transfer this executable to the machine where it will run, it is listed with the **transfer_input_files** command, and the file transfer mechanism is enabled with the **should_transfer_files** command.

Here is the equivalent sample submit description file, but for an Open MPI application:

```
####################################
## Example submit description file
## for Open MPI
####################################
universe = parallel
executable = openmpiscript
arguments = my_openmpi_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_openmpi_linked_executable
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

Most MPI implementations require two system-wide prerequisites. The first prerequisite is the ability to run a command on a remote machine without being prompted for a password. *ssh* is commonly used. The second prerequisite is an ASCII file containing the list of machines that may utilize *ssh*. These common prerequisites are implemented in a further script called `sshd.sh`. `sshd.sh` generates ssh keys to enable password-less remote execution and starts an *sshd* daemon. Use of the *sshd.sh* script requires the definition of two HTCondor configuration variables. Configuration variable `CONDOR_SSHD` is an absolute path to an implementation of *sshd*. *sshd.sh* has been tested with *openssh* version 3.9, but should work with more recent versions. Configuration variable `CONDOR_SSH_KEYGEN` points to the corresponding *ssh-keygen* executable.

mp1script and *mp2script* require the `PATH` to the MPICH installation to be set. The variable `MPDIR` may be modified in the scripts to indicate its proper value. This directory contains the MPICH *mpirun* executable.

openmpiscript also requires the PATH to the Open MPI installation. Either the variable MPDIR can be set manually in the script, or the administrator can define MPDIR using the configuration variable OPENMPI_INSTALL_PATH . When using Open MPI on a multi-machine HTCondor cluster, the administrator may also want to consider tweaking the OPENMPI_EXCLUDE_NETWORK_INTERFACES configuration variable as well as set MOUNT_UNDER_SCRATCH = /tmp.

4.14.4 MPI Applications Within HTCondor's Vanilla Universe

The vanilla universe may be preferred over the parallel universe for parallel applications which can run entirely on one machine. The **request_cpus** command causes a claimed slot to have the required number of CPUs (cores).

There are two ways to ensure that the MPI job can run on any machine that it lands on:

1. Statically build an MPI library and statically compile the MPI code.
2. Bundle all the MPI libraries into a docker container and run MPI in the container

Here is a submit description file example assuming that MPI is installed on all machines on which the MPI job may run, or that the code was built using static libraries and a static version of mpirun is available.

```
############################################################
##   submit description file for
##   static build of MPI under the vanilla universe
############################################################
universe = vanilla
executable = /path/to/mpirun
request_cpus = 2
arguments = -np 2 my_mpi_linked_executable arg1 arg2 arg3
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_mpi_linked_executable
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

Any additional input files that will be needed for the executable that are not already in the tarball should be included in the list in **transfer_input_files** command. The corresponding script should then also be updated to move those files into the directory where the executable will be run.

4.15 Virtual Machine Applications

The **vm** universe facilitates an HTCondor job that matches and then lands a disk image on an execute machine within an HTCondor pool. This disk image is intended to be a virtual machine. In this manner, the virtual machine is the job to be executed.

This section describes this type of HTCondor job. See *Configuration File Entries Relating to Virtual Machines* for details of configuration variables.

4.15.1 The Submit Description File

Different than all other universe jobs, the **vm** universe job specifies a disk image, not an executable. Therefore, the submit commands **input** , **output** , and **error** do not apply. If specified, *condor_submit* rejects the job with an error. The **executable** command changes definition within a **vm** universe job. It no longer specifies an executable file, but instead provides a string that identifies the job for tools such as *condor_q*. Other commands specific to the type of virtual machine software identify the disk image.

Xen and KVM virtual machine software are supported. As these differ from each other, the submit description file specifies one of

```
vm_type = xen
```

or

```
vm_type = kvm
```

The job is required to specify its memory needs for the disk image with **vm_memory** , which is given in Mbytes. HTCondor uses this number to assure a match with a machine that can provide the needed memory space.

Virtual machine networking is enabled with the command

```
vm_networking = true
```

And, when networking is enabled, a definition of **vm_networking_type** as **bridge** matches the job only with a machine that is configured to use bridge networking. A definition of **vm_networking_type** as **nat** matches the job only with a machine that is configured to use NAT networking. When no definition of **vm_networking_type** is given, HTCondor may match the job with a machine that enables networking, and further, the choice of bridge or NAT networking is determined by the machine's configuration.

Modified disk images are transferred back to the machine from which the job was submitted as the **vm** universe job completes. Job completion for a **vm** universe job occurs when the virtual machine is shut down, and HTCondor notices (as the result of a periodic check on the state of the virtual machine). Should the job not want any files transferred back (modified or not), for example because the job explicitly transferred its own files, the submit command to prevent the transfer is

```
vm_no_output_vm = true
```

The required disk image must be identified for a virtual machine. This **vm_disk** command specifies a list of comma-separated files. Each disk file is specified by colon-separated fields. The first field is the path and file name of the disk file. The second field specifies the device. The third field specifies permissions, and the optional fourth specifies the format. Here is an example that identifies a single file:

```
vm_disk = swap.img:sda2:w:raw
```

If HTCondor will be transferring the disk file, then the file name given in **vm_disk** should not contain any path information. Otherwise, the full path to the file should be given.

Setting values in the submit description file for some commands have consequences for the virtual machine description file. These commands are

- **vm_memory**
- **vm_macaddr**
- **vm_networking**
- **vm_networking_type**

- **vm_disk**

HTCondor uses these values when it produces the description file.

If any files need to be transferred from the access point to the machine where the **vm** universe job will execute, HTCondor must be explicitly told to do so with the standard file transfer attributes:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /myxen/diskfile.img,/myxen/swap.img
```

Any and all needed files that will not accessible directly from the machines where the job may execute must be listed.

Further commands specify information that is specific to the virtual machine type targeted.

Xen-Specific Submit Commands

A Xen **vm** universe job requires specification of the guest kernel. The **xen_kernel** command accomplishes this, utilizing one of the following definitions.

1. `xen_kernel = included` implies that the kernel is to be found in disk image given by the definition of the single file specified in **vm_disk** .
2. `xen_kernel = path-to-kernel` gives the file name of the required kernel. If this kernel must be transferred to machine on which the **vm** universe job will execute, it must also be included in the **transfer_input_files** command.

   This form of the **xen_kernel** command also requires further definition of the **xen_root** command. **xen_root** defines the device containing files needed by root.

4.15.2 Checkpoints

Creating a checkpoint is straightforward for a virtual machine, as a checkpoint is a set of files that represent a snapshot of both disk image and memory. The checkpoint is created and all files are transferred back to the `$(SPOOL)` directory on the machine from which the job was submitted. The submit command to create checkpoints is

```
vm_checkpoint = true
```

Without this command, no checkpoints are created (by default). With the command, a checkpoint is created any time the **vm** universe jobs is evicted from the machine upon which it is executing. This occurs as a result of the machine configuration indicating that it will no longer execute this job.

Periodic creation of checkpoints is not supported at this time.

Enabling both networking and checkpointing for a **vm** universe job can cause networking problems when the job restarts, particularly if the job migrates to a different machine. *condor_submit* will normally reject such jobs. To enable both, then add the command

```
when_to_transfer_output = ON_EXIT_OR_EVICT
```

Take care with respect to the use of network connections within the virtual machine and their interaction with checkpoints. Open network connections at the time of the checkpoint will likely be lost when the checkpoint is subsequently used to resume execution of the virtual machine. This occurs whether or not the execution resumes on the same machine or a different one within the HTCondor pool.

4.15.3 Disk Images

Xen and KVM

While the following web page contains instructions specific to Fedora on how to create a virtual guest image, it should provide a good starting point for other platforms as well.

http://fedoraproject.org/wiki/Virtualization_Quick_Start

4.15.4 Job Completion in the vm Universe

Job completion for a **vm** universe job occurs when the virtual machine is shut down, and HTCondor notices (as the result of a periodic check on the state of the virtual machine). This is different from jobs executed under the environment of other universes.

Shut down of a virtual machine occurs from within the virtual machine environment. A script, executed with the proper authorization level, is the likely source of the shut down commands.

Under a Windows 2000, Windows XP, or Vista virtual machine, an administrator issues the command

```
> shutdown -s -t 01
```

Under a Linux virtual machine, the root user executes

```
$ /sbin/poweroff
```

The command `/sbin/halt` will not completely shut down some Linux distributions, and instead causes the job to hang.

Since the successful completion of the **vm** universe job requires the successful shut down of the virtual machine, it is good advice to try the shut down procedure outside of HTCondor, before a **vm** universe job is submitted.

4.15.5 Failures to Launch

It is not uncommon for a **vm** universe job to fail to launch because of a problem with the execute machine. In these cases, HTCondor will reschedule the job and note, in its user event log (if requested), the reason for the failure and that the job will be rescheduled. The reason is unlikely to be directly useful to you as an HTCondor user, but may help your HTCondor administrator understand the problem.

If the VM fails to launch for other reasons, the job will be placed on hold and the reason placed in the job ClassAd's `HoldReason` attribute. The following table may help in understanding such reasons.

VMGAHP_ERR_JOBCLASSAD_NO_VM_MEMORY_PARAM
: The attribute JobVMMemory was not set in the job ad sent to the VM GAHP. HTCondor will usually prevent you from submitting a VM universe job without JobVMMemory set. Examine your job and verify that JobVM-Memory is set. If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_KVM_NO_DISK_PARAM
: The attribute VMPARAM_vm_Disk was not set in the job ad sent to the VM GAHP. HTCondor will usually set this attribute when you submit a valid KVM job (it is derived from vm_disk). Examine your job and verify that VMPARAM_vm_Disk is set. If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_KVM_INVALID_DISK_PARAM
: The attribute vm_disk was invalid. Please consult the manual, or the condor_submit man page, for information about the syntax of vm_disk. A syntactically correct value may be invalid if the on-disk permissions of a file

specified in it do not match the requested permissions. Presently, files not transferred to the root of the working directory must be specified with full paths.

VMGAHP_ERR_JOBCLASSAD_KVM_MISMATCHED_CHECKPOINT
KVM jobs can not presently checkpoint if any of their disk files are not on a shared filesystem. Files on a shared filesystem must be specified in vm_disk with full paths.

VMGAHP_ERR_JOBCLASSAD_XEN_NO_KERNEL_PARAM
The attribute VMPARAM_Xen_Kernel was not set in the job ad sent to the VM GAHP. HTCondor will usually set this attribute when you submit a valid Xen job (it is derived from xen_kernel). Examine your job and verify that VMPARAM_Xen_Kernel is set. If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_MISMATCHED_HARDWARE_VT
Don't use 'vmx' as the name of your kernel image. Pick something else and change xen_kernel to match.

VMGAHP_ERR_JOBCLASSAD_XEN_KERNEL_NOT_FOUND
HTCondor could not read from the file specified by xen_kernel. Check the path and the file's permissions. If it's on a shared filesystem, you may need to alter your job's requirements expression to ensure the filesystem's availability.

VMGAHP_ERR_JOBCLASSAD_XEN_INITRD_NOT_FOUND
HTCondor could not read from the file specified by xen_initrd. Check the path and the file's permissions. If it's on a shared filesystem, you may need to alter your job's requirements expression to ensure the filesystem's availability.

VMGAHP_ERR_JOBCLASSAD_XEN_NO_ROOT_DEVICE_PARAM
The attribute VMPARAM_Xen_Root was not set in the job ad sent to the VM GAHP. HTCondor will usually set this attribute when you submit a valid Xen job (it is derived from xen_root). Examine your job and verify that VMPARAM_Xen_Root is set. If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_XEN_NO_DISK_PARAM
The attribute VMPARAM_vm_Disk was not set in the job ad sent to the VM GAHP. HTCondor will usually set this attribute when you submit a valid Xen job (it is derived from vm_disk). Examine your job and verify that VMPARAM_vm_Disk is set. If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_XEN_INVALID_DISK_PARAM
The attribute vm_disk was invalid. Please consult the manual, or the condor_submit man page, for information about the syntax of vm_disk. A syntactically correct value may be invalid if the on-disk permissions of a file specified in it do not match the requested permissions. Presently, files not transferred to the root of the working directory must be specified with full paths.

VMGAHP_ERR_JOBCLASSAD_XEN_MISMATCHED_CHECKPOINT
Xen jobs can not presently checkpoint if any of their disk files are not on a shared filesystem. Files on a shared filesystem must be specified in vm_disk with full paths.

4.16 Docker Universe Applications

A docker universe job instantiates a Docker container from a Docker image, and HTCondor manages the running of that container as an HTCondor job, on an execute machine. This running container can then be managed as any HTCondor job. For example, it can be scheduled, removed, put on hold, or be part of a workflow managed by DAGMan.

The docker universe job will only be matched with an execute host that advertises its capability to run docker universe jobs. When an execute machine with docker support starts, the machine checks to see if the *docker* command is available and has the correct settings for HTCondor. Docker support is advertised if available and if it has the correct settings.

The image from which the container is instantiated is defined by specifying a Docker image with the submit command **docker_image** . This image must be pre-staged on a docker hub that the execute machine can access.

The submit file command **universe** can either be optionally set to `docker` or not declared at all. If **universe** is declared and set to anything but `docker` then the job submission will fail. Regardless, the submit file command **docker_image** must be declared and set to a docker image.

After submission, the job is treated much the same way as a vanilla universe job. Details of file transfer are the same as applied to the vanilla universe. One of the benefits of Docker containers is the file system isolation they provide. Each container has a distinct file system, from the root on down, and this file system is completely independent of the file system on the host machine. The container does not share a file system with either the execute host or the submit host, with the exception of the scratch directory, which is volume mounted to the host, and is the initial working directory of the job. Optionally, the administrator may configure other directories from the host machine to be volume mounted, and thus visible inside the container. See the docker section of the administrator's manual for details.

In Docker universe (as well as vanilla), HTCondor never allows a containerized process to run as root inside the container, it always runs as a non-root user. It will run as the same non-root user that a vanilla job will. If a Docker Universe job fails in an obscure way, but runs fine in a docker container on a desktop, try running the job as a non-root user on the desktop to try to duplicate the problem.

HTCondor creates a per-job scratch directory on the execute machine, transfers any input files to that directory, bind-mounts that directory to a directory of the same name inside the container, and sets the IWD of the contained job to that directory. The assumption is that the job will look in the cwd for input files, and drop output files in the same directory. In docker terms, we docker run with the -v /some_scratch_directory -w /some_scratch_directory -user non-root-user command line options (along with many others).

The executable file can come from one of two places: either from within the container's image, or it can be a script transferred from the submit machine to the scratch directory of the execute machine. To specify the former, use an absolute path (starting with a /) for the executable. For the latter, use a relative path.

Therefore, the submit description file should contain the submit command

```
should_transfer_files = YES
```

With this command, all input and output files will be transferred as required to and from the scratch directory mounted as a Docker volume.

If no **executable** is specified in the submit description file, it is presumed that the Docker container has a default command to run.

When the job completes, is held, evicted, or is otherwise removed from the machine, the container will be removed.

Here is a complete submit description file for a sample docker universe job:

```
#universe = docker is optional
universe                = docker
docker_image            = debian
executable              = /bin/cat
arguments               = /etc/hosts
should_transfer_files   = YES
when_to_transfer_output = ON_EXIT
output                  = out.$(Process)
error                   = err.$(Process)
log                     = log.$(Process)

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K
```

(continues on next page)

(continued from previous page)

```
queue 1
```

A debian container is the HTCondor job, and it runs the */bin/cat* program on the `/etc/hosts` file before exiting.

4.16.1 Docker and Networking

By default, docker universe jobs will be run with a private, NATed network interface.

In the job submit file, if the user specifies

```
docker_network_type = none
```

then no networking will be available to the job.

In the job submit file, if the user specifies

```
docker_network_type = host
```

then, instead of a NATed interface, the job will use the host's network interface, just like a vanilla universe job. If an administrator has defined additional, custom docker networks, they will be advertised in the slot attribute *DockerNetworks*, and any value in that list can be a valid argument for this keyword.

If the *host* network type is unavailable, you can ask Docker to forward one or more ports on the host into the container. In the following example, we assume that the 'centos7_with_htcondor' image has HTCondor set up and ready to go, but doesn't turn it on by default.

```
#universe = docker is optional
universe                = docker
docker_image            = centos7_with_htcondor
executable              = /usr/sbin/condor_master
arguments               = -f
container_service_names = condor
condor_container_port   = 9618
should_transfer_files   = YES
when_to_transfer_output = ON_EXIT
output                  = out.$(Process)
error                   = err.$(Process)
log                     = log.$(Process)

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue 1
```

The `container_service_names` submit command accepts a comma- or space- separated list of service names; each service name must have a corresponding `<service-name>_container_port` submit command specifying an integer between 0 and 65535. Docker will automatically select a port on the host to forward to that port in the container; HTCondor will report that port in the job ad attribute `<service-name>_HostPort` after it becomes available, which will be (several seconds) after the job starts. HTCondor will update the job ad in the sandbox (`.job.ad`) at that time.

4.17 Container Universe Jobs

In addition to Docker, many competing container runtimes have been developed, some of which are mostly compatible with Docker, and others which provide their own feature sets. Many HTCondor users and administrators want to run jobs inside containers, but don't care which runtime is used.

HTCondor's container universe provides an abstraction where the user does not specify exactly which container runtime to use, but just aspects of their contained job, and HTCondor will select an appropriate runtime. To do this, set the job submit file command **container_image** to a specified container image.

The submit file command **universe** can either be optionally set to `container` or not declared at all. If **universe** is declared and set to anything but `container` then the job submission will fail.

Note that the container may specify the executable to run, either in the runfile option of a singularity image, or in the entrypoint option of a Dockerfile. If this is set, the executable command in the HTCondor submit file is optional, and the default command in the container will be run.

This container image may describe an image in a docker-style repo if it is prefixed with `docker://`, or a Singularity `.sif` image on disk, or a Singularity sandbox image (an exploded directory). *condor_submit* will parse this image and advertise what type of container image it is, and match with startds that can support that image.

The container image may also be specified with an URL syntax that tells HTCondor to use a file transfer plugin to transfer the image. For example with

```
container_image = http://example.com/dir/image.sif
```

A container image that would otherwise be transferred can be forced to never be transferred by setting

```
should_transfer_container = no
```

HTCondor knows that "docker://" and "oras://" (for apptainer) are special, and are never transferred by HTCondor plugins.

Here is a complete submit description file for a sample container universe job:

```
#universe = container is optional
universe                = container
container_image         = ./image.sif

executable              = /bin/cat
arguments               = /etc/hosts

should_transfer_files   = YES
when_to_transfer_output = ON_EXIT

output                  = out.$(Process)
error                   = err.$(Process)
log                     = log.$(Process)

request_cpus    = 1
request_memory  = 1024M
request_disk    = 10240K

queue 1
```

4.18 Self-Checkpointing Applications

This section is about writing jobs for an executable which periodically saves checkpoint information, and how to make HTCondor store that information safely, in case it's needed to continue the job on another machine or at a later time.

This section is *not* about how to checkpoint a given executable; that's up to you or your software provider.

4.18.1 How To Run Self-Checkpointing Jobs

The best way to run self-checkpointing code is to set `checkpoint_exit_code` in your submit file. (Any exit code will work, but if you can choose, consider error code `85`. On Linux systems, this is `ERESTART`, which seems appropriate.) If the `executable` exits with `checkpoint_exit_code`, HTCondor will transfer the checkpoint to the submit node, and then immediately restart the `executable` in the same sandbox on the same machine, with the same `arguments`. This immediate transfer makes the checkpoint available for continuing the job even if the job is interrupted in a way that doesn't allow for files to be transferred (e.g., power failure), or if the file transfer doesn't complete in the time allowed.

For a job to use `checkpoint_exit_code` successfully, its `executable` must meet a number of requirements.

4.18.2 Requirements

Your self-checkpointing code may not meet all of the following requirements. In many cases, however, you will be able to add a wrapper script, or modify an existing one, to meet these requirements. (Thus, your `executable` may be a script, rather than the code that's writing the checkpoint.) If you can not, consult *Working Around the Assumptions* and/or the *Other Options*.

1. Your executable exits after taking a checkpoint with an exit code it does not otherwise use.
    - If your executable does not exit when it takes a checkpoint, HTCondor will not transfer its checkpoint. If your executable exits normally when it takes a checkpoint, HTCondor will not be able to tell the difference between taking a checkpoint and actually finishing; that is, if the checkpoint code and the terminal exit code are the same, your job will never finish.
2. When restarted, your executable determines on its own if a checkpoint is available, and if so, uses it.
    - If your job does not look for a checkpoint each time it starts up, it will start from scratch each time; HTCondor does not run a different command line when restarting a job which has taken a checkpoint.
3. Starting your executable up from a checkpoint is relatively quick.
    - If starting your executable up from a checkpoint is relatively slow, your job may not run efficiently enough to be useful, depending on the frequency of checkpoints and interruptions.

4.18.3 Using checkpoint_exit_code

The following Python script (`example.py`) is a toy example of code that checkpoints itself. It counts from 0 to 10 (exclusive), sleeping for 10 seconds at each step. It writes a checkpoint file (containing the next number) after each nap, and exits with code 85 at count 3, 6, and 9. It exits with code 0 when complete.

```
#!/usr/bin/env python

import sys
import time

```

(continues on next page)

(continued from previous page)

```
value = 0
try:
    with open('example.checkpoint', 'r') as f:
        value = int(f.read())
except IOError:
    pass

print("Starting from {0}".format(value))
for i in range(value,10):
    print("Computing timestamp {0}".format(value))
    time.sleep(10)
    value += 1
    with open('example.checkpoint', 'w') as f:
        f.write("{0}".format(value))
    if value%3 == 0:
        sys.exit(85)

print("Computation complete")
sys.exit(0)
```

The following submit file (`example.submit`) commands HTCondor to transfer the file `example.checkpoint` to the submit node whenever the script exits with code 85. If interrupted, the job will resume from the most recent of those checkpoints. Before version 8.9.8, you *must* include your checkpoint file(s) in `transfer_output_files`; otherwise HTCondor will not transfer it (them). Starting with version 8.9.8, you may instead use `transfer_checkpoint_files`, as documented on the *condor_submit* man page.

```
checkpoint_exit_code        = 85
transfer_output_files       = example.checkpoint
should_transfer_files       = yes

executable                  = example.py
arguments                   =

output                      = example.out
error                       = example.err
log                         = example.log

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

queue 1
```

This example does not remove the "checkpoint file" generated for timestep 9 when the executable completes. This could be done in `example.py` immediately before it exits, but that would cause the final file transfer to fail, if you specified the file in `transfer_output_files`. The script could instead remove the file and then re-create it empty, it desired.

4.18.4 How Frequently to Checkpoint

Obviously, the longer the code spends writing checkpoints, and the longer your job spends transferring them, the longer it will take for you to get the job's results. Conversely, the more frequently the job transfers new checkpoints, the less time the job loses if it's interrupted. For most users and for most jobs, taking a checkpoint about once an hour works well, and it's not a bad duration to start experimenting with. A number of factors will skew this interval up or down:

- If your job(s) usually run on resources with strict time limits, you may want to adjust how often your job checkpoints to minimize wasted time. For instance, if your job writes a checkpoint after each hour, and each checkpoint takes five minutes to write out and then transfer, your fifth checkpoint will finish twenty-five minutes into the fifth hour, and you won't gain any benefit from the next thirty-five minutes of computation. If you instead write a checkpoint every eighty-four minutes, your job will only waste four minutes.
- If a particular code writes larger checkpoints, or writes smaller checkpoints unusually slowly, you may want to take a checkpoint less frequently than you would for other jobs of a similar length, to keep the total overhead (delay) the same. The opposite is also true: if the job can take checkpoints particularly quickly, or the checkpoints are particularly small, the job could checkpoint more often for the same amount of overhead.
- Some code naturally checkpoints at longer or shorter intervals. If a code writes a checkpoint every five minutes, it may make sense for the `executable` to wait for the code to write ten or more checkpoints before exiting (which asks HTCondor to transfer the checkpoint file(s)). If a job is a sequence of steps, the natural (or only possible) checkpoint interval may be between steps.
- How long it takes to restart from a checkpoint. It should never take longer to restart from a checkpoint than to recompute from the beginning, but the restart process is part of the overhead of taking a checkpoint. The longer a code takes to restart, the less often the `executable` should exit.

Measuring how long it takes to make checkpoints is left as an exercise for the reader. Since version 8.9.1, however, HTCondor will report in the job's log (if a log is enabled for that job) how long file transfers, including checkpoint transfers, took.

4.18.5 Debugging Self-Checkpointing Jobs

Because a job may be interrupted at any time, it's valid to interrupt the job at any time and see if a valid checkpoint is transferred. To do so, use `condor_vacate_job` to evict the job. When that's done (watch the user log), use `condor_hold` to put it on hold, so that it can't restart while you're looking at the checkpoint (and potentially, overwrite it). Finally, to obtain the checkpoint file(s) themselves, use the somewhat mis-named `condor_evicted_files` to ask where they are.

For example, if your job is ID `635.0`, and is logging to the file `job.log`, you can copy the files in the checkpoint to a subdirectory of the current as follows:

```
$ condor_vacate_job 635.0
```

Wait for the job to finish being evicted; hit CTRL-C when you see 'Job was evicted.' and immediately hold the job.

```
$ tail --follow job.log
$ condor_hold 635.0
```

Copy the checkpoint files from the spool. Note that `_condor_stderr` and `_condor_stdout` are the files corresponding to the job's output and error submit commands; they aren't named correctly until the the job finishes.

```
$ condor_evicted_files get 635.0
Copied to '635.0'.
$ cd 635.0
```

Now examine the checkpoint files to see if they look right. When you're done, release the job to see if it actually works right.

```
$ condor_release 635.0
$ condor_ssh_to_job 635.0
```

You may also want to remove your copy of checkpoint files:

```
$ cd ..; rm -fr 635.0
```

4.18.6 Working Around the Assumptions

The basic technique here is to write a wrapper script (or modify an existing one), so that the `executable` has the necessary behavior, even if the code does not.

1. *Your executable exits after taking a checkpoint with an exit code it does not otherwise use.*
    - If your code exits when it takes a checkpoint, but not with a unique code, your wrapper script will have to determine, when the executable exits, if it did so because it took a checkpoint. If so, the wrapper script will have to exit with a unique code. If the code could usefully exit with any code, and the wrapper script therefore can not exit with a unique code, you can instead instruct HTCondor to consider being killed by a particular signal as a sign of successful checkpoint; set `+SuccessCheckpointExitBySignal` to `TRUE` and `+SuccessCheckpointExitSignal` to the particular signal. (If you do not set `checkpoint_exit_code`, you must set `+WantFTOnCheckpoint`.)
    - If your code does not exit when it takes a checkpoint, the wrapper script will have to determine when a checkpoint has been made, kill the program, and then exit with a unique code.
2. *When restarted, your executable determines on its own if a checkpoint is available, and if so, uses it.*
    - If your code requires different arguments to start from a checkpoint, the wrapper script must check for the presence of a checkpoint and start the executable with correspondingly modified arguments.
3. *Starting your executable up from a checkpoint is relatively quick.*
    - The longer the start-up delay, the slower the job's overall progress. If your job's progress is too slow as a result of start-up delay, and your code can take checkpoints without exiting, read the 'Delayed Transfers' and 'Manual Transfers' sections below.

4.18.7 Other Options

The preceding sections of this HOWTO explain how a job meeting the requirements can take checkpoints at arbitrary intervals and transfer them back to the submit node. Although this is the method of operation most likely to result in an interrupted job continuing from a valid checkpoint, other, less reliable options exist.

Delayed Transfers

This method is risky, because it does not allow your job to recover from any failure mode other than an eviction (and sometimes not even then). It may also require changes to your `executable`. The advantage of this method is that it doesn't require your code to restart, or even a recent version of HTCondor.

The basic idea is to take checkpoints as the job runs, but not transfer them back to the submit node until the job is evicted. This implies that your `executable` doesn't exit until the job is complete (which is the normal case). If your code has long start-up delays, you'll naturally not want it to exit after it writes a checkpoint; otherwise, the wrapper script could restart the code as necessary.

To use this method, set when_to_transfer_output to ON_EXIT_OR_EVICT instead of setting checkpoint_exit_code. This will cause HTCondor to transfer your checkpoint file(s) (which you listed in transfer_output_files, as noted above) when the job is evicted. Of course, since this is the only time your checkpoint file(s) will be transferred, if the transfer fails, your job has to start over from the beginning. One reason file transfer on eviction fails is if it takes too long, so this method may not work if your transfer_output_files contain too much data.

Furthermore, eviction can happen at any time, including while the code is updating its checkpoint file(s). If the code does not update its checkpoint file(s) atomically, HTCondor will transfer the partially-updated checkpoint file(s), potentially overwriting the previous, complete one(s); this will probably prevent the code from picking up where it left off.

In some cases, you can work around this problem by using a wrapper script. The idea is that renaming a file is an atomic operation, so if your code writes checkpoints to one file, call it checkpoint, your wrapper script – when it detects that the checkpoint is complete – would rename that file checkpoint.atomic. That way, checkpoint.atomic always has a complete checkpoint in it. With a such a script, instead of putting checkpoint in transfer_output_files, you would put checkpoint.atomic, and HTCondor would never see a partially-complete checkpoint file. (The script would also, of course, have to copy checkpoint.atomic to checkpoint before running the code.)

Manual Transfers

If you're comfortable with programming, instead of running a job with checkpoint_exit_code, you could use condor_chirp, or other tools, to manage your checkpoint file(s). Your executable would be responsible for downloading the checkpoint file(s) on start-up, and periodically uploading the checkpoint file(s) during execution. We don't recommend you do this for the same reasons we recommend against managing your own input and output file transfers.

Early Checkpoint Exits

If your executable's natural checkpoint interval is half or more of your pool's max job runtime, it may make sense to checkpoint and then immediately ask to be rescheduled, rather than lower your user priority doing work you know will be thrown away. In this case, you can use the OnExitRemove job attribute to determine if your job should be rescheduled after exiting. Don't set ON_EXIT_OR_EVICT, and don't set +WantFTOnCheckpoint; just have the job exit with a unique code after its checkpoint.

4.18.8 Signals

Signals offer additional options for running self-checkpointing jobs. If you're not familiar with signals, this section may not make sense to you.

Periodic Signals

HTCondor supports transferring checkpoint file(s) for an executable which takes a checkpoint when sent a particular signal, if the executable then exits in a unique way. Set +WantCheckpointSignal to TRUE to periodically receive checkpoint signals, and +CheckpointSig to specify which one. (The interval is specified by the administrator of the execute machine.) The unique way may be a specific exit code, for which you would set checkpoint_exit_code, or a signal, for which you would set +SuccessCheckpointExitBySignal to TRUE and +SuccessCheckpointExitSignal to the particular signal. (If you do not set checkpoint_exit_code, you must set +WantFTOnCheckpoint.)

Delayed Transfer with Signals

This method is very similar to but riskier than delayed transfers, because in addition to delaying the transfer of the checkpoint files(s), it also delays their creation. Thus, this option should almost never be used; if taking and transferring your checkpoint file(s) is fast enough to reliably complete during an eviction, you're not losing much by doing so periodically, and it's unlikely that a code which takes small checkpoints quickly takes a long time to start up. However, this method will work even with very old version of HTCondor.

To use this method, set `when_to_transfer_output` to `ON_EXIT_OR_EVICT` and `KillSig` to the particular signal that causes your job to checkpoint.

4.19 Submitting a Remote Job

4.19.1 Submitting a job to a remote Access Point

Usually, when you run the *condor_submit* ` command, you are logged into an Access Point (AP) which is running a *condor_schedd*, and your submit defaults to sending the job to the *condor_schedd* running on that same AP. However, it is possible to have `condor_submit` send the job to a *condor_schedd* running on some other machine. Maybe you want to run `condor_submit` from your laptop and send the job to an AP on some server. Maybe you are building a web portal, and you want the portal to run on one machine, and the *condor_schedd* running on some other machine.

The first concern is security. When you submit locally, the *condor_schedd* can easily determine who is submitting the job, and thus what system account it should run the *condor_shadow* as. This is much more difficult with a remote, over-the-network submit. For this to work, some additional setup must happen. While this authentication can be setup with SSL, Kerberos or Windows native methods, for Linux systems, we recommend HTCondor's ID tokens, as it is easy for a user to setup, and secure.

Why remote submission?

While it isn't the usual case, there are several reasons you might want to submit from one machine to another. Maybe you want to run `condor_submit` from your laptop and send the job to an AP on some other server, because you have input data on your laptop, and don't want to manually copy it to your Access Point. Maybe you are building a web portal, and you want the portal to run on one machine, and the *condor_schedd* process running on some other machine to balance load.

Assuming that an administrator has set up signing keys (see *Token Authentication*), to create a token that can authenticate you for remote submission, login to the access point and run the command

```
$ condor_token_fetch -token name_of_your_ap
```

Note that name_of_your_ap is merely a filename, but if you have more than one AP, it is good to name the file containing the token clearly. When this command succeeds, there is no output but the access token is place into the file with that name in the tokens.d subdirectory of your personal .condor directory in your home directory.

If you copy this directory and contents from the AP (the machine you want to submit *to*, and place the directory in the same place on the machine you want to submit *from*, then `condor_submit` can submit remotely. To do so, you'll need to tell `condor_submit` the name of the pool (i.e. the name of the machine running the central manager), and the name of the Access Point that you ran *condor_token_fetch* on. If you don't know the name of the central manager, running the command `condor_config_val COLLECTOR_HOST` will tell you.

Then, to submit the job, on the remote machine, simple run

```
$ condor_submit -name name-of-ap -pool cm-name submit_file
```

and perhaps any other options you might want to pass to *condor_submit* After `condor_submit` reports the cluster id of your new job, it has been successfully submitted to the AP, and the AP is responsible for the management of the job thereafter. You can query the job with

```
$ condor_q -name name-of-ap -pool cm-name
```

and run all the related commands like `condor_rm`, `condor_hold` and `condor_release` in a similar way.

4.19.2 File transfer with remote submission

After `condor_submit` successfully completes a remote submission, the machine you ran `condor_submit` on is not involved at all in the management of the job; the remote AP manages it. Therefore, you can disconnect that machine from the network, turn it off, or hibernate it. Even if this machine is turned off, the AP will find a matching Execution Point to run the job on, and run it to completion.

This means that any input files specified in *transfer_input_files* are copied off of this access point as part of the submit process and stored in a safe place on the Access Point. This safe place is the spool directory. While a user can force spooling to happen by adding the `-spool` option to `condor_submit`, any remote submit (with the -name option) automatically turns on spooling. Note that files transferred via file transfer plugins are never spooled, they are always pulled by the worker node immediately before job execution.

Correspondingly, when the jobs complete, output files cannot be transferred to the submitting machine, as it may be off, or disconnected from the network. These files are also stored in the spool directory of the AP machine. To indicate that a completed job still has spool files it is holding on the AP machine, a remotely submitted job remained in the AP's, and is visible with the `condor_q` command after completion, and is in the 'C'ompleted state. Jobs will stay in this state for three days by default, or until you have fetched the output files off of the machine.

You can fetch the output sandbox from the AP back to your submitting machine (or anywhere that has permissions), by running the `condor_transfer_data` command. This also takes a `-name` and `-pool` option like *condor_submit*. You can specify a job or jobs in the usual way, often just with the cluster.proc syntax. When run, it copies the job's output sandbox from the spool on the AP back to the current directory of the machine `condor_transfer_data` is run.

4.20 Time Scheduling for Job Execution

Jobs may be scheduled to begin execution at a specified time in the future with HTCondor's job deferral functionality. All specifications are in a job's submit description file. Job deferral functionality is expanded to provide for the periodic execution of a job, known as the CronTab scheduling.

4.20.1 Job Deferral

Job deferral allows the specification of the exact date and time at which a job is to begin executing. HTCondor attempts to match the job to an execution machine just like any other job, however, the job will wait until the exact time to begin execution. A user can define the job to allow some flexibility in the execution of jobs that miss their execution time.

Deferred Execution Time

A job's deferral time is the exact time that HTCondor should attempt to execute the job. The deferral time attribute is defined as an expression that evaluates to a Unix Epoch timestamp (the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time). This is the time that HTCondor will begin to execute the job.

After a job is matched and all of its files have been transferred to an execution machine, HTCondor checks to see if the job's ClassAd contains a deferral time. If it does, HTCondor calculates the number of seconds between the execution machine's current system time and the job's deferral time. If the deferral time is in the future, the job waits to begin execution. While a job waits, its job ClassAd attribute `JobStatus` indicates the job is in the Running state. As the deferral time arrives, the job begins to execute. If a job misses its execution time, that is, if the deferral time is in the past, the job is evicted from the execution machine and put on hold in the queue.

The specification of a deferral time does not interfere with HTCondor's behavior. For example, if a job is waiting to begin execution when a *condor_hold* command is issued, the job is removed from the execution machine and is put on hold. If a job is waiting to begin execution when a *condor_suspend* command is issued, the job continues to wait. When the deferral time arrives, HTCondor begins execution for the job, but immediately suspends it.

The deferral time is specified in the job's submit description file with the command **deferral_time** .

Deferral Window

If a job arrives at its execution machine after the deferral time has passed, the job is evicted from the machine and put on hold in the job queue. This may occur, for example, because the transfer of needed files took too long due to a slow network connection. A deferral window permits the execution of a job that misses its deferral time by specifying a window of time within which the job may begin.

The deferral window is the number of seconds after the deferral time, within which the job may begin. When a job arrives too late, HTCondor calculates the difference in seconds between the execution machine's current time and the job's deferral time. If this difference is less than or equal to the deferral window, the job immediately begins execution. If this difference is greater than the deferral window, the job is evicted from the execution machine and is put on hold in the job queue.

The deferral window is specified in the job's submit description file with the command **deferral_window** .

Preparation Time

When a job defines a deferral time far in the future and then is matched to an execution machine, potential computation cycles are lost because the deferred job has claimed the machine, but is not actually executing. Other jobs could execute during the interval when the job waits for its deferral time. To make use of the wasted time, a job defines a **deferral_prep_time** with an integer expression that evaluates to a number of seconds. At this number of seconds before the deferral time, the job may be matched with a machine.

Deferral Usage Examples

Here are examples of how the job deferral time, deferral window, and the preparation time may be used.

The job's submit description file specifies that the job is to begin execution on January 1st, 2006 at 12:00 pm:

```
deferral_time = 1136138400
```

The Unix *date* program may be used to calculate a Unix epoch time. The syntax of the command to do this depends on the options provided within that flavor of Unix. In some, it appears as

```
$ date --date "MM/DD/YYYY HH:MM:SS" +%s
```

and in others, it appears as

```
$ date -d "YYYY-MM-DD HH:MM:SS" +%s
```

MM is a 2-digit month number, DD is a 2-digit day of the month number, and YYYY is a 4-digit year. HH is the 2-digit hour of the day, MM is the 2-digit minute of the hour, and SS are the 2-digit seconds within the minute. The characters +%s tell the *date* program to give the output as a Unix epoch time.

The job always waits 60 seconds after submission before beginning execution:

```
deferral_time = (QDate + 60)
```

In this example, assume that the deferral time is 45 seconds in the past as the job is available. The job begins execution, because 75 seconds remain in the deferral window:

```
deferral_window = 120
```

In this example, a job is scheduled to execute far in the future, on January 1st, 2010 at 12:00 pm. The **deferral_prep_time** attribute delays the job from being matched until 60 seconds before the job is to begin execution.

```
deferral_time      = 1262368800
deferral_prep_time = 60
```

Deferral Limitations

There are some limitations to HTCondor's job deferral feature.

- Job deferral is not available for scheduler universe jobs. A scheduler universe job defining the `deferral_time` produces a fatal error when submitted.
- The time that the job begins to execute is based on the execution machine's system clock, and not the submission machine's system clock. Be mindful of the ramifications when the two clocks show dramatically different times.
- A job's `JobStatus` attribute is always in the Running state when job deferral is used. There is currently no way to distinguish between a job that is executing and a job that is waiting for its deferral time.

4.20.2 CronTab Scheduling

HTCondor's CronTab scheduling functionality allows jobs to be scheduled to execute periodically. A job's execution schedule is defined by commands within the submit description file. The notation is much like that used by the Unix *cron* daemon. As such, HTCondor developers are fond of referring to CronTab scheduling as Crondor. The scheduling of jobs using HTCondor's CronTab feature calculates and utilizes the `DeferralTime` ClassAd attribute.

Also, unlike the Unix *cron* daemon, HTCondor never runs more than one instance of a job at the same time.

The capability for repetitive or periodic execution of the job is enabled by specifying an **on_exit_remove** command for the job, such that the job does not leave the queue until desired.

Semantics for CronTab Specification

A job's execution schedule is defined by a set of specifications within the submit description file. HTCondor uses these to calculate a `DeferralTime` for the job.

Table 2.3 lists the submit commands and acceptable values for these commands. At least one of these must be defined in order for HTCondor to calculate a `DeferralTime` for the job. Once one CronTab value is defined, the default for all the others uses all the values in the allowed values ranges.

cron_minute	0 - 59
cron_hour	0 - 23
cron_day_of_month	1 - 31
cron_month	1 - 12
cron_day_of_week	0 - 7 (Sunday is 0 or 7)

Table 2.3: The list of submit commands and their value ranges.

The day of a job's execution can be specified by both the **cron_day_of_month** and the **cron_day_of_week** attributes. The day will be the logical or of both.

The semantics allow more than one value to be specified by using the * operator, ranges, lists, and steps (strides) within ranges.

The asterisk operator
The * (asterisk) operator specifies that all of the allowed values are used for scheduling. For example,

```
cron_month = *
```

becomes any and all of the list of possible months: (1,2,3,4,5,6,7,8,9,10,11,12). Thus, a job runs any month in the year.

Ranges
A range creates a set of integers from all the allowed values between two integers separated by a hyphen. The specified range is inclusive, and the integer to the left of the hyphen must be less than the right hand integer. For example,

```
cron_hour = 0-4
```

represents the set of hours from 12:00 am (midnight) to 4:00 am, or (0,1,2,3,4).

Lists
A list is the union of the values or ranges separated by commas. Multiple entries of the same value are ignored. For example,

```
cron_minute = 15,20,25,30
cron_hour   = 0-3,9-12,15
```

where this **cron_minute** example represents (15,20,25,30) and **cron_hour** represents (0,1,2,3,9,10,11,12,15).

Steps

Steps select specific numbers from a range, based on an interval. A step is specified by appending a range or the asterisk operator with a slash character (/), followed by an integer value. For example,

```
cron_minute = 10-30/5
cron_hour = */3
```

where this **cron_minute** example specifies every five minutes within the specified range to represent (10,15,20,25,30), and **cron_hour** specifies every three hours of the day to represent (0,3,6,9,12,15,18,21).

Preparation Time and Execution Window

The **cron_prep_time** command is analogous to the deferral time's **deferral_prep_time** command. It specifies the number of seconds before the deferral time that the job is to be matched and sent to the execution machine. This permits HTCondor to make necessary preparations before the deferral time occurs.

Consider the submit description file example that includes

```
cron_minute = 0
cron_hour = *
cron_prep_time = 300
```

The job is scheduled to begin execution at the top of every hour. Note that the setting of **cron_hour** in this example is not required, as the default value will be *, specifying any and every hour of the day. The job will be matched and sent to an execution machine no more than five minutes before the next deferral time. For example, if a job is submitted at 9:30am, then the next deferral time will be calculated to be 10:00am. HTCondor may attempt to match the job to a machine and send the job once it is 9:55am.

As the CronTab scheduling calculates and uses deferral time, jobs may also make use of the deferral window. The submit command **cron_window** is analogous to the submit command **deferral_window** . Consider the submit description file example that includes

```
cron_minute = 0
cron_hour = *
cron_window = 360
```

As the previous example, the job is scheduled to begin execution at the top of every hour. Yet with no preparation time, the job is likely to miss its deferral time. The 6-minute window allows the job to begin execution, as long as it arrives and can begin within 6 minutes of the deferral time, as seen by the time kept on the execution machine.

Scheduling

When a job using the CronTab functionality is submitted to HTCondor, use of at least one of the submit description file commands beginning with **cron_** causes HTCondor to calculate and set a deferral time for when the job should run. A deferral time is determined based on the current time rounded later in time to the next minute. The deferral time is the job's `DeferralTime` attribute. A new deferral time is calculated when the job first enters the job queue, when the job is re-queued, or when the job is released from the hold state. New deferral times for all jobs in the job queue using the CronTab functionality are recalculated when a *condor_reconfig* or a *condor_restart* command that affects the job queue is issued.

A job's deferral time is not always the same time that a job will receive a match and be sent to the execution machine. This is because HTCondor operates on the job queue at times that are independent of job events, such as when job execution completes. Therefore, HTCondor may operate on the job queue just after a job's deferral time states that it is to begin execution. HTCondor attempts to start a job when the following pseudo-code boolean expression evaluates to `True`:

```
( time() + SCHEDD_INTERVAL ) >= ( DeferralTime - CronPrepTime )
```

If the `time()` plus the number of seconds until the next time HTCondor checks the job queue is greater than or equal to the time that the job should be submitted to the execution machine, then the job is to be matched and sent now.

Jobs using the CronTab functionality are not automatically re-queued by HTCondor after their execution is complete. The submit description file for a job must specify an appropriate **on_exit_remove** command to ensure that a job remains in the queue. This job maintains its original `ClusterId` and `ProcId`.

Submit Commands Usage Examples

Here are some examples of the submit commands necessary to schedule jobs to run at multifarious times. Please note that it is not necessary to explicitly define each attribute; the default value is *.

Run 23 minutes after every two hours, every day of the week:

```
on_exit_remove = false
cron_minute = 23
cron_hour = 0-23/2
cron_day_of_month = *
cron_month = *
cron_day_of_week = *
```

Run at 10:30pm on each of May 10th to May 20th, as well as every remaining Monday within the month of May:

```
on_exit_remove = false
cron_minute = 30
cron_hour = 20
cron_day_of_month = 10-20
cron_month = 5
cron_day_of_week = 2
```

Run every 10 minutes and every 6 minutes before noon on January 18th with a 2-minute preparation time:

```
on_exit_remove = false
cron_minute = */10,*/6
cron_hour = 0-11
cron_day_of_month = 18
cron_month = 1
```

(continues on next page)

(continued from previous page)

```
cron_day_of_week = *
cron_prep_time = 120
```

Submit Commands Limitations

The use of the CronTab functionality has all of the same limitations of deferral times, because the mechanism is based upon deferral times.

- It is impossible to schedule vanilla universe jobs at intervals that are smaller than the interval at which HTCondor evaluates jobs. This interval is determined by the configuration variable `SCHEDD_INTERVAL` . As a vanilla universe job completes execution and is placed back into the job queue, it may not be placed in the idle state in time. This problem does not afflict local universe jobs.
- HTCondor cannot guarantee that a job will be matched in order to make its scheduled deferral time. A job must be matched with an execution machine just as any other HTCondor job; if HTCondor is unable to find a match, then the job will miss its chance for executing and must wait for the next execution time specified by the CronTab schedule.

4.21 Special Environment Considerations

4.21.1 AFS

The HTCondor daemons do not run authenticated to AFS; they do not possess AFS tokens. Therefore, no child process of HTCondor will be AFS authenticated. The implication of this is that you must set file permissions so that your job can access any necessary files residing on an AFS volume without relying on having your AFS permissions.

If a job you submit to HTCondor needs to access files residing in AFS, you have the following choices:

1. If the files must be kept on AFS, then set a host ACL (using the AFS *fs setacl* command) on the subdirectory to serve as the current working directory for the job. Set the ACL such that any host in the pool can access the files without being authenticated. If you do not know how to use an AFS host ACL, ask the person at your site responsible for the AFS configuration.

The Center for High Throughput Computing hopes to improve upon how HTCondor deals with AFS authentication in a subsequent release.

Please see the *Using HTCondor with AFS* section for further discussion of this problem.

4.21.2 NFS

If the current working directory when a job is submitted is accessed via an NFS automounter, HTCondor may have problems if the automounter later decides to unmount the volume before the job has completed. This is because *condor_submit* likely has stored the dynamic mount point as the job's initial current working directory, and this mount point could become automatically unmounted by the automounter.

There is a simple work around. When submitting the job, use the submit command **initialdir** to point to the stable access point. For example, suppose the NFS automounter is configured to mount a volume at mount point `/a/myserver.company.com/vol1/johndoe` whenever the directory `/home/johndoe` is accessed. Adding the following line to the submit description file solves the problem.

```
initialdir = /home/johndoe
```

HTCondor attempts to flush the NFS cache on a access point in order to refresh a job's initial working directory. This allows files written by the job into an NFS mounted initial working directory to be immediately visible on the access point. Since the flush operation can require multiple round trips to the NFS server, it is expensive. Therefore, a job may disable the flushing by setting

```
+IwdFlushNFSCache = False
```

in the job's submit description file. See the *Job ClassAd Attributes* page for a definition of the job ClassAd attribute.

4.21.3 HTCondor Daemons That Do Not Run as root

HTCondor is normally installed such that the HTCondor daemons have root permission. This allows HTCondor to run the *condor_shadow* daemon and the job with the submitting user's UID and file access rights. When HTCondor is started as root, HTCondor jobs can access whatever files the user that submits the jobs can.

However, it is possible that the HTCondor installation does not have root access, or has decided not to run the daemons as root. That is unfortunate, since HTCondor is designed to be run as root. To see if HTCondor is running as root on a specific machine, use the command

```
$ condor_status -master -l <machine-name>
```

where <machine-name> is the name of the specified machine. This command displays the full condor_master ClassAd; if the attribute `RealUid` equals zero, then the HTCondor daemons are indeed running with root access. If the `RealUid` attribute is not zero, then the HTCondor daemons do not have root access.

NOTE: The Unix program *ps* is not an effective method of determining if HTCondor is running with root access. When using *ps*, it may often appear that the daemons are running as the condor user instead of root. However, note that the *ps* command shows the current effective owner of the process, not the real owner. (See the *getuid* (2) and *geteuid* (2) Unix man pages for details.) In Unix, a process running under the real UID of root may switch its effective UID. (See the *seteuid* (2) man page.) For security reasons, the daemons only set the effective UID to root when absolutely necessary, as it will be to perform a privileged operation.

If daemons are not running with root access, make any and all files and/or directories that the job will touch readable and/or writable by the UID (user id) specified by the `RealUid` attribute. Often this may mean using the Unix command chmod 777 on the directory from which the HTCondor job is submitted.

4.21.4 Job Leases

A job lease specifies how long a given job will attempt to run on a remote resource, even if that resource loses contact with the submitting machine. Similarly, it is the length of time the submitting machine will spend trying to reconnect to the (now disconnected) execution host, before the submitting machine gives up and tries to claim another resource to run the job. The goal aims at run only once semantics, so that the *condor_schedd* daemon does not allow the same job to run on multiple sites simultaneously.

If the submitting machine is alive, it periodically renews the job lease, and all is well. If the submitting machine is dead, or the network goes down, the job lease will no longer be renewed. Eventually the lease expires. While the lease has not expired, the execute host continues to try to run the job, in the hope that the access point will come back to life and reconnect. If the job completes and the lease has not expired, yet the submitting machine is still dead, the

condor_starter daemon will wait for a *condor_shadow* daemon to reconnect, before sending final information on the job, and its output files. Should the lease expire, the *condor_startd* daemon kills off the *condor_starter* daemon and user job.

A default value equal to 40 minutes exists for a job's ClassAd attribute `JobLeaseDuration`, or this attribute may be set in the submit description file, using **job_lease_duration** , to keep a job running in the case that the submit side no longer renews the lease. There is a trade off in setting the value of **job_lease_duration** . Too small a value, and the job might get killed before the submitting machine has a chance to recover. Forward progress on the job will be lost. Too large a value, and an execute resource will be tied up waiting for the job lease to expire. The value should be chosen based on how long the user is willing to tie up the execute machines, how quickly access points come back up, and how much work would be lost if the lease expires, the job is killed, and the job must start over from its beginning.

As a special case, a submit description file setting of

```
job_lease_duration = 0
```

as well as utilizing submission other than *condor_submit* that do not set `JobLeaseDuration` (such as using the web services interface) results in the corresponding job ClassAd attribute to be explicitly undefined. This has the further effect of changing the duration of a claim lease, the amount of time that the execution machine waits before dropping a claim due to missing keep alive messages.

4.21.5 Heterogeneous Submit: Execution on Differing Architectures

If executables are available for the different platforms of machines in the HTCondor pool, HTCondor can be allowed the choice of a larger number of machines when allocating a machine for a job. Modifications to the submit description file allow this choice of platforms.

A simplified example is a cross submission. An executable is available for one platform, but the submission is done from a different platform. Given the correct executable, the `requirements` command in the submit description file specifies the target architecture. For example, an executable compiled for a 32-bit Intel processor running Windows Vista, submitted from an Intel architecture running Linux would add the `requirement`

```
requirements = Arch == "INTEL" && OpSys == "WINDOWS"
```

Without this `requirement`, *condor_submit* will assume that the program is to be executed on a machine with the same platform as the machine where the job is submitted.

Vanilla Universe Example for Execution on Differing Architectures

A more complex example of a heterogeneous submission occurs when a job may be executed on many different architectures to gain full use of a diverse architecture and operating system pool. If the executables are available for the different architectures, then a modification to the submit description file will allow HTCondor to choose an executable after an available machine is chosen.

A special-purpose Machine Ad substitution macro can be used in string attributes in the submit description file. The macro has the form

```
$$(MachineAdAttribute)
```

The $$() informs HTCondor to substitute the requested `MachineAdAttribute` from the machine where the job will be executed.

An example of the heterogeneous job submission has executables available for two platforms: RHEL 3 on both 32-bit and 64-bit Intel processors. This example uses *povray* to render images using a popular free rendering engine.

The substitution macro chooses a specific executable after a platform for running the job is chosen. These executables must therefore be named based on the machine attributes that describe a platform. The executables named

```
povray.LINUX.INTEL
povray.LINUX.X86_64
```

will work correctly for the macro

```
povray.$$(OpSys).$$(Arch)
```

The executables or links to executables with this name are placed into the initial working directory so that they may be found by HTCondor. A submit description file that queues three jobs for this example:

```
# Example of heterogeneous submission

universe     = vanilla
executable   = povray.$$(OpSys).$$(Arch)
log          = povray.log
output       = povray.out.$(Process)
error        = povray.err.$(Process)

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

requirements = (Arch == "INTEL" && OpSys == "LINUX") || \
               (Arch == "X86_64" && OpSys =="LINUX")

arguments    = +W1024 +H768 +Iimage1.pov
queue

arguments    = +W1024 +H768 +Iimage2.pov
queue

arguments    = +W1024 +H768 +Iimage3.pov
queue
```

These jobs are submitted to the vanilla universe to assure that once a job is started on a specific platform, it will finish running on that platform. Switching platforms in the middle of job execution cannot work correctly.

There are two common errors made with the substitution macro. The first is the use of a non-existent `MachineAdAttribute`. If the specified `MachineAdAttribute` does not exist in the machine's ClassAd, then HTCondor will place the job in the held state until the problem is resolved.

The second common error occurs due to an incomplete job set up. For example, the submit description file given above specifies three available executables. If one is missing, HTCondor reports back that an executable is missing when it happens to match the job with a resource that requires the missing binary.

Vanilla Universe Example for Execution on Differing Operating Systems

The addition of several related OpSys attributes assists in selection of specific operating systems and versions in heterogeneous pools.

```
# Example targeting only RedHat platforms

universe     = vanilla
Executable   = /bin/date
Log          = distro.log
Output       = distro.out
Error        = distro.err

Requirements = (OpSysName == "RedHat")

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

Queue
```

```
# Example targeting RedHat 6 platforms in a heterogeneous Linux pool

universe     = vanilla
executable   = /bin/date
log          = distro.log
output       = distro.out
error        = distro.err

requirements = ( OpSysName == "RedHat" && OpSysMajorVer == 6 )

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

queue
```

Here is a more compact way to specify a RedHat 6 platform.

```
# Example targeting RedHat 6 platforms in a heterogeneous Linux pool

universe     = vanilla
executable   = /bin/date
log          = distro.log
output       = distro.out
error        = distro.err

request_cpus            = 1
request_memory          = 512M
request_disk            = 1G

requirements = (OpSysAndVer == "RedHat6")

queue
```

CHAPTER

FIVE

ADMINISTRATORS' MANUAL

5.1 Introduction

This is the HTCondor Administrator's Manual. Its purpose is to aid in the installation and administration of an HTCondor pool. For help on using HTCondor, see the HTCondor User's Manual.

An HTCondor pool is comprised of a single machine which serves as the central manager, and an arbitrary number of other machines. Machines intended to run work are called Execution Points (EP)s, also known as worker nodes. Machines that hold a queue of jobs ready to run, or the results of jobs that have run are called Access Points (AP)s, also known as submit machines. The role of HTCondor is to match waiting requests with available resources. Every part of HTCondor sends periodic updates to the central manager, the centralized repository of information about the state of the pool. Periodically, the central manager assesses the current state of the pool and tries to match pending requests with the appropriate resources.

Each resource has an owner, the one who sets the policy for the use of the machine. This person has absolute power over the use of the machine, and HTCondor goes out of its way to minimize the impact on this owner caused by HTCondor. It is up to the resource owner to define a policy for when HTCondor requests will serviced and when they will be denied.

Each resource request has an owner as well: the user who submitted the job. These people want HTCondor to provide as many CPU cycles as possible for their work. Often the interests of the resource owners are in conflict with the interests of the resource requesters. The job of the HTCondor administrator is to configure the HTCondor pool to find the happy medium that keeps both resource owners and users of resources satisfied. The purpose of this manual is to relate the mechanisms that HTCondor provides to enable the administrator to find this happy medium.

5.1.1 The Different Roles a Machine Can Play

Every machine in an HTCondor pool can serve a variety of roles. Most machines serve more than one role simultaneously. Certain roles can only be performed by a single machine in the pool. The following list describes what these roles are and what resources are required on the machine that is providing that service:

Central Manager
: There can be only one central manager for the pool. This machine is the collector of information, and the negotiator between resources and resource requests. These two halves of the central manager's responsibility are performed by separate daemons, so it would be possible to have different machines providing those two services. However, normally they both live on the same machine. This machine plays a very important part in the HTCondor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the HTCondor system, although all current matches remain in effect until they are broken by either party involved in the match. Therefore, choose for central manager a machine that is likely to be up and running all the time, or at least one that will be rebooted quickly if something goes wrong. The central manager will ideally have a good network connection to all the machines in the pool, since these pool machines all send updates over the network to the central manager.

Note:

Fig. 1: Daemons for Central Manager, both managed by a *condor_master*

Execution Point
Any machine in the pool, including the central manager, can be configured as to whether or not it should execute HTCondor jobs. Obviously, some of the machines will have to serve this function, or the pool will not be useful. Being an execute machine does not require lots of resources. About the only resource that might matter is disk space. In general the more resources a machine has in terms of swap space, memory, number of CPUs, the larger variety of resource requests it can serve.

Note:

Fig. 2: Daemons for a Execution Point, one *condor_starter* per running job.

Access Point
Any machine in the pool, including the central manager, can be configured as to whether or not it should allow HTCondor jobs to be submitted. The resource requirements for an access point are actually much greater than the resource requirements for an execute machine. Every submitted job that is currently running on a remote machine runs a process on the access point. As a result, lots of running jobs will need a fair amount of swap space and/or real memory. HTCondor pools can scale out horizontally by adding additional access points. Older terminology called these submit machines or scheduler machine.

Note:

Fig. 3: Daemons for an Access Point, one *condor_shadow* per running job.

5.1.2 The HTCondor Daemons

The following list describes all the daemons and programs that could be started under HTCondor and what they do:

condor_master
This daemon is responsible for keeping all the rest of the HTCondor daemons running on each machine in the pool. It spawns the other daemons, and it periodically checks to see if there are new binaries installed for any of them. If there are, the *condor_master* daemon will restart the affected daemons. In addition, if any daemon crashes, the *condor_master* will send e-mail to the HTCondor administrator of the pool and restart the daemon. The *condor_master* also supports various administrative commands that enable the administrator to start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in the pool, regardless of the functions that each machine is performing.

condor_startd
This daemon represents a given resource to the HTCondor pool, as a machine capable of running jobs. It advertises certain attributes about machine that are used to match it with pending resource requests. The *condor_startd* will run on any machine in the pool that is to be able to execute jobs. It is responsible for enforcing the policy that

the resource owner configures, which determines under what conditions jobs will be started, suspended, resumed, vacated, or killed. When the *condor_startd* is ready to execute an HTCondor job, it spawns the *condor_starter*.

condor_starter
This daemon is the entity that actually spawns the HTCondor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the *condor_starter* notices this, sends back any status information to the submitting machine, and exits.

condor_schedd
This daemon represents resource requests to the HTCondor pool. Any machine that is to be an access point needs to have a *condor_schedd* running. When users submit jobs, the jobs go to the *condor_schedd*, where they are stored in the job queue. The *condor_schedd* manages the job queue. Various tools to view and manipulate the job queue, such as *condor_submit*, *condor_q*, and *condor_rm*, all must connect to the *condor_schedd* to do their work. If the *condor_schedd* is not running on a given machine, none of these commands will work.

The *condor_schedd* advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a job has been matched with a given resource, the *condor_schedd* spawns a *condor_shadow* daemon to serve that particular request.

condor_shadow
This daemon runs on the machine where a given request was submitted and acts as the resource manager for the request.

condor_collector
This daemon is responsible for collecting all the information about the status of an HTCondor pool. All other daemons periodically send ClassAd updates to the *condor_collector*. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool. The *condor_status* command can be used to query the *condor_collector* for specific information about various parts of HTCondor. In addition, the HTCondor daemons themselves query the *condor_collector* for important information, such as what address to use for sending commands to a remote machine.

condor_negotiator
This daemon is responsible for all the match making within the HTCondor system. Periodically, the *condor_negotiator* begins a negotiation cycle, where it queries the *condor_collector* for the current state of all the resources in the pool. It contacts each *condor_schedd* that has waiting resource requests in priority order, and tries to match available resources with those requests. The *condor_negotiator* is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the *condor_negotiator* can preempt that resource and match it with the user with better priority.

Note: A higher numerical value of the user priority in HTCondor translate into worse priority for that user. The best priority is 0.5, the lowest numerical value, and this priority gets worse as this number grows.

---

condor_kbdd
This daemon is used on both Linux and Windows platforms. On those platforms, the *condor_startd* frequently cannot determine console (keyboard or mouse) activity directly from the system, and requires a separate process to do so. On Linux, the *condor_kbdd* connects to the X Server and periodically checks to see if there has been any activity. On Windows, the *condor_kbdd* runs as the logged-in user and registers with the system to receive keyboard and mouse events. When it detects console activity, the *condor_kbdd* sends a command to the *condor_startd*. That way, the *condor_startd* knows the machine owner is using the machine again and can perform whatever actions are necessary, given the policy it has been configured to enforce.

condor_gridmanager
This daemon handles management and execution of all **grid** universe jobs. The *condor_schedd* invokes the *condor_gridmanager* when there are **grid** universe jobs in the queue, and the *condor_gridmanager* exits when

there are no more **grid** universe jobs in the queue.

condor_credd
This daemon runs on Windows platforms to manage password storage in a secure manner.

condor_had
This daemon implements the high availability of a pool's central manager through monitoring the communication of necessary daemons. If the current, functioning, central manager machine stops working, then this daemon ensures that another machine takes its place, and becomes the central manager of the pool.

condor_replication
This daemon assists the *condor_had* daemon by keeping an updated copy of the pool's state. This state provides a better transition from one machine to the next, in the event that the central manager machine stops working.

condor_transferer
This short lived daemon is invoked by the *condor_replication* daemon to accomplish the task of transferring a state file before exiting.

condor_procd
This daemon controls and monitors process families within HTCondor. Its use is optional in general, but it must be used if group-ID based tracking (see the *Setting Up for Special Environments* section) is enabled.

condor_job_router
This daemon transforms **vanilla** universe jobs into **grid** universe jobs, such that the transformed jobs are capable of running elsewhere, as appropriate.

condor_lease_manager
This daemon manages leases in a persistent manner. Leases are represented by ClassAds.

condor_rooster
This daemon wakes hibernating machines based upon configuration details.

condor_defrag
This daemon manages the draining of machines with fragmented partitionable slots, so that they become available for jobs requiring a whole machine or larger fraction of a machine.

condor_shared_port
This daemon listens for incoming TCP packets on behalf of HTCondor daemons, thereby reducing the number of required ports that must be opened when HTCondor is accessible through a firewall.

5.2 Starting Up, Shutting Down and Reconfiguring the System

If you installed HTCondor with administrative privileges, HTCondor will start up when the machine boots and shut down when the machine does, using the usual mechanism for the machine's operating system. You can generally use those mechanisms in the usual way if you need to manually control whether or not HTCondor is running. There are two situations in which you might want to run *condor_master*, *condor_on*, or *condor_off* from the command line.

1. If you installed HTCondor without administrative privileges, you'll have to run *condor_master* from the command line to turn on HTCondor:

   ```
   $ condor_master
   ```

   Then run the following command to turn HTCondor completely off:

   ```
   $ condor_off -master
   ```

2. If the usual OS-specific method of controlling HTCondor is inconvenient to use remotely, you may be able to use the *condor_on* and *condor_off* tools instead.

5.2.1 Using HTCondor's Remote Management Features

All of the commands described in this section are subject to the security policy chosen for the HTCondor pool. As such, the commands must be either run from a machine that has the proper authorization, or run by a user that is authorized to issue the commands. The *Security* section details the implementation of security in HTCondor.

Shutting Down HTCondor

There are a variety of ways to shut down all or parts of an HTCondor pool. All utilize the *condor_off* tool.

To stop a single execute machine from running jobs, the *condor_off* command specifies the machine by host name.

```
$ condor_off -startd <hostname>
```

Jobs will be killed. If it is instead desired that the machine stops running jobs only after the currently executing job completes, the command is

```
$ condor_off -startd -peaceful <hostname>
```

Note that this waits indefinitely for the running job to finish, before the *condor_startd* daemon exits.

Th shut down all execution machines within the pool,

```
$ condor_off -all -startd
```

To wait indefinitely for each machine in the pool to finish its current HTCondor job, shutting down all of the execute machines as they no longer have a running job,

```
$ condor_off -all -startd -peaceful
```

To shut down HTCondor on a machine from which jobs are submitted,

```
$ condor_off -schedd <hostname>
```

If it is instead desired that the access point (which runs the *condor_schedd*) shuts down only after all jobs that are currently in the queue are finished, first disable new submissions to the queue by setting the configuration variable

```
MAX_JOBS_SUBMITTED = 0
```

See instructions below in *Reconfiguring an HTCondor Pool* for how to reconfigure a pool. After the reconfiguration, the command to wait for all jobs to complete and shut down the submission of jobs is

```
$ condor_off -schedd -peaceful <hostname>
```

Substitute the option **-all** for the host name, if all submit machines in the pool are to be shut down.

Restarting HTCondor, If HTCondor Daemons Are Not Running

If HTCondor is not running, perhaps because one of the *condor_off* commands was used, then starting HTCondor daemons back up depends on which part of HTCondor is currently not running.

If no HTCondor daemons are running, then starting HTCondor is a matter of executing the *condor_master* daemon. The *condor_master* daemon will then invoke all other specified daemons on that machine. The *condor_master* daemon executes on every machine that is to run HTCondor.

If a specific daemon needs to be started up, and the *condor_master* daemon is already running, then issue the command on the specific machine with

```
$ condor_on -subsystem <subsystemname>
```

where <subsystemname> is replaced by the daemon's subsystem name. Or, this command might be issued from another machine in the pool (which has administrative authority) with

```
$ condor_on <hostname> -subsystem <subsystemname>
```

where <subsystemname> is replaced by the daemon's subsystem name, and <hostname> is replaced by the host name of the machine where this *condor_on* command is to be directed.

Restarting HTCondor, If HTCondor Daemons Are Running

If HTCondor daemons are currently running, but need to be killed and newly invoked, the *condor_restart* tool does this. This would be the case for a new value of a configuration variable for which using *condor_reconfig* is inadequate.

To restart all daemons on all machines in the pool,

```
$ condor_restart -all
```

To restart all daemons on a single machine in the pool,

```
$ condor_restart <hostname>
```

where <hostname> is replaced by the host name of the machine to be restarted.

Reconfiguring an HTCondor Pool

To change a global configuration variable and have all the machines start to use the new setting, change the value within the file, and send a *condor_reconfig* command to each host. Do this with a single command,

```
$ condor_reconfig -all
```

If the global configuration file is not shared among all the machines, as it will be if using a shared file system, the change must be made to each copy of the global configuration file before issuing the *condor_reconfig* command.

Issuing a *condor_reconfig* command is inadequate for some configuration variables. For those, a restart of HTCondor is required. Those configuration variables that require a restart are listed in the *Macros That Will Require a Restart When Changed* section. You can also refer to the *condor_restart* manual page.

5.3 Introduction to Configuration

This section of the manual contains general information about HTCondor configuration, relating to all parts of the HTCondor system. If you're setting up an HTCondor pool, you should read this section before you read the other configuration-related sections:

- The *Configuration Templates* section contains information about configuration templates, which are now the preferred way to set many configuration macros.

- The *Configuration Macros* section contains information about the hundreds of individual configuration macros. In general, it is best to try to achieve your desired configuration using configuration templates before resorting to setting individual configuration macros, but it is sometimes necessary to set individual configuration macros.
- The settings that control the policy under which HTCondor will start, suspend, resume, vacate or kill jobs are described in the *Policy Configuration for Execution Points and for Access Points* section on Policy Configuration for the *condor_startd*.

5.3.1 HTCondor Configuration Files

The HTCondor configuration files are used to customize how HTCondor operates at a given site. The basic configuration as shipped with HTCondor can be used as a starting point, but most likely you will want to modify that configuration to some extent.

Each HTCondor program will, as part of its initialization process, configure itself by calling a library routine which parses the various configuration files that might be used, including pool-wide, platform-specific, and machine-specific configuration files. Environment variables may also contribute to the configuration.

The result of configuration is a list of key/value pairs. Each key is a configuration variable name, and each value is a string literal that may utilize macro substitution (as defined below). Some configuration variables are evaluated by HTCondor as ClassAd expressions; some are not. Consult the documentation for each specific case. Unless otherwise noted, configuration values that are expected to be numeric or boolean constants can be any valid ClassAd expression of operators on constants. Example:

```
MINUTE           = 60
HOUR             = (60 * $(MINUTE))
SHUTDOWN_GRACEFUL_TIMEOUT = ($(HOUR)*24)
```

5.3.2 Ordered Evaluation to Set the Configuration

Multiple files, as well as a program's environment variables, determine the configuration. The order in which attributes are defined is important, as later definitions override earlier definitions. The order in which the (multiple) configuration files are parsed is designed to ensure the security of the system. Attributes which must be set a specific way must appear in the last file to be parsed. This prevents both the naive and the malicious HTCondor user from subverting the system through its configuration. The order in which items are parsed is:

1. a single initial configuration file, which has historically been known as the global configuration file (see below);
2. other configuration files that are referenced and parsed due to specification within the single initial configuration file (these files have historically been known as local configuration files);
3. if HTCondor daemons are not running as root on Unix platforms, the file `$(HOME)/.condor/user_config` if it exists, or the file defined by configuration variable `USER_CONFIG_FILE` ;

   if HTCondor daemons are not running as Local System on Windows platforms, the file %USERPROFILE\.condor\user_config if it exists, or the file defined by configuration variable `USER_CONFIG_FILE` ;
4. specific environment variables whose names are prefixed with `_CONDOR_` (note that these environment variables directly define macro name/value pairs, not the names of configuration files).

Some HTCondor tools utilize environment variables to set their configuration; these tools search for specifically-named environment variables. The variable names are prefixed by the string `_CONDOR_` or `_condor_`. The tools strip off the prefix, and utilize what remains as configuration. As the use of environment variables is the last within the ordered evaluation, the environment variable definition is used. The security of the system is not compromised, as only specific variables are considered for definition in this manner, not any environment variables with the `_CONDOR_` prefix.

The location of the single initial configuration file differs on Windows from Unix platforms. For Unix platforms, the location of the single initial configuration file starts at the top of the following list. The first file that exists is used, and then remaining possible file locations from this list become irrelevant.

1. the file specified by the CONDOR_CONFIG environment variable. If there is a problem reading that file, HTCondor will print an error message and exit right away.
2. /etc/condor/condor_config
3. /usr/local/etc/condor_config
4. ~condor/condor_config

For Windows platforms, the location of the single initial configuration file is determined by the contents of the environment variable CONDOR_CONFIG. If this environment variable is not defined, then the location is the registry value of HKEY_LOCAL_MACHINE/Software/Condor/CONDOR_CONFIG.

The single, initial configuration file may contain the specification of one or more other configuration files, referred to here as local configuration files. Since more than one file may contain a definition of the same variable, and since the last definition of a variable sets the value, the parse order of these local configuration files is fully specified here. In order:

1. The value of configuration variable LOCAL_CONFIG_DIR lists one or more directories which contain configuration files. The list is parsed from left to right. The leftmost (first) in the list is parsed first. Within each directory, a lexicographical ordering by file name determines the ordering of file consideration.
2. The value of configuration variable LOCAL_CONFIG_FILE lists one or more configuration files. These listed files are parsed from left to right. The leftmost (first) in the list is parsed first.
3. If one of these steps changes the value (right hand side) of LOCAL_CONFIG_DIR, then LOCAL_CONFIG_DIR is processed for a second time, using the changed list of directories.

The parsing and use of configuration files may be bypassed by setting environment variable CONDOR_CONFIG with the string ONLY_ENV. With this setting, there is no attempt to locate or read configuration files. This may be useful for testing where the environment contains all needed information.

5.3.3 Configuration File Macros

Macro definitions are of the form:

```
<macro_name> = <macro_definition>
```

The macro name given on the left hand side of the definition is a case insensitive identifier. There may be white space between the macro name, the equals sign (=), and the macro definition. The macro definition is a string literal that may utilize macro substitution.

Macro invocations are of the form:

```
$(macro_name[:<default if macro_name not defined>])
```

The colon and default are optional in a macro invocation. Macro definitions may contain references to other macros, even ones that are not yet defined, as long as they are eventually defined in the configuration files. All macro expansion is done after all configuration files have been parsed, with the exception of macros that reference themselves.

```
A = xxx
C = $(A)
```

is a legal set of macro definitions, and the resulting value of C is xxx. Note that C is actually bound to $(A), not its value.

As a further example,

```
A = xxx
C = $(A)
A = yyy
```

is also a legal set of macro definitions, and the resulting value of C is yyy.

A macro may be incrementally defined by invoking itself in its definition. For example,

```
A = xxx
B = $(A)
A = $(A)yyy
A = $(A)zzz
```

is a legal set of macro definitions, and the resulting value of A is xxxyyyzzz. Note that invocations of a macro in its own definition are immediately expanded. $(A) is immediately expanded in line 3 of the example. If it were not, then the definition would be impossible to evaluate.

Recursively defined macros such as

```
A = $(B)
B = $(A)
```

are not allowed. They create definitions that HTCondor refuses to parse.

A macro invocation where the macro name is not defined results in a substitution of the empty string. Consider the example

```
MAX_ALLOC_CPUS = $(NUMCPUS)-1
```

If NUMCPUS is not defined, then this macro substitution becomes

```
MAX_ALLOC_CPUS = -1
```

The default value may help to avoid this situation. The default value may be a literal

```
MAX_ALLOC_CPUS = $(NUMCPUS:4)-1
```

such that if NUMCPUS is not defined, the result of macro substitution becomes

```
MAX_ALLOC_CPUS = 4-1
```

The default may be another macro invocation:

```
MAX_ALLOC_CPUS = $(NUMCPUS:$(DETECTED_CPUS_LIMIT))-1
```

These default specifications are restricted such that a macro invocation with a default can not be nested inside of another default. An alternative way of stating this restriction is that there can only be one colon character per line. The effect of nested defaults can be achieved by placing the macro definitions on separate lines of the configuration.

All entries in a configuration file must have an operator, which will be an equals sign (=). Identifiers are alphanumerics combined with the underscore character, optionally with a subsystem name and a period as a prefix. As a special case, a line without an operator that begins with a left square bracket will be ignored. The following two-line example treats the first line as a comment, and correctly handles the second line.

```
[HTCondor Settings]
my_classad = [ foo=bar ]
```

To simplify pool administration, any configuration variable name may be prefixed by a subsystem (see the $(SUBSYSTEM) macro in *Pre-Defined Macros* for the list of subsystems) and the period (.) character. For configuration variables defined this way, the value is applied to the specific subsystem. For example, the ports that HTCondor may use can be restricted to a range using the HIGHPORT and LOWPORT configuration variables.

```
MASTER.LOWPORT    = 20000
MASTER.HIGHPORT   = 20100
```

Note that all configuration variables may utilize this syntax, but nonsense configuration variables may result. For example, it makes no sense to define

```
NEGOTIATOR.MASTER_UPDATE_INTERVAL = 60
```

since the *condor_negotiator* daemon does not use the MASTER_UPDATE_INTERVAL variable.

It makes little sense to do so, but HTCondor will configure correctly with a definition such as

```
MASTER.MASTER_UPDATE_INTERVAL = 60
```

The *condor_master* uses this configuration variable, and the prefix of MASTER. causes this configuration to be specific to the *condor_master* daemon.

As of HTCondor version 8.1.1, evaluation works in the expected manner when combining the definition of a macro with use of a prefix that gives the subsystem name and a period. Consider the example

```
FILESPEC = A
MASTER.FILESPEC = B
```

combined with a later definition that incorporates FILESPEC in a macro:

```
USEFILE = mydir/$(FILESPEC)
```

When the *condor_master* evaluates variable USEFILE, it evaluates to mydir/B. Previous to HTCondor version 8.1.1, it evaluated to mydir/A. When any other subsystem evaluates variable USEFILE, it evaluates to mydir/A.

This syntax has been further expanded to allow for the specification of a local name on the command line using the command line option

```
-local-name <local-name>
```

This allows multiple instances of a daemon to be run by the same *condor_master* daemon, each instance with its own local configuration variable.

The ordering used to look up a variable, called <parameter name>:

1. <subsystem name>.<local name>.<parameter name>
2. <local name>.<parameter name>
3. <subsystem name>.<parameter name>
4. <parameter name>

If this local name is not specified on the command line, numbers 1 and 2 are skipped. As soon as the first match is found, the search is completed, and the corresponding value is used.

This example configures a *condor_master* to run 2 *condor_schedd* daemons. The *condor_master* daemon needs the configuration:

```
XYZZY           = $(SCHEDD)
XYZZY_ARGS      = -local-name xyzzy
DAEMON_LIST     = $(DAEMON_LIST) XYZZY
DC_DAEMON_LIST  = + XYZZY
XYZZY_LOG       = $(LOG)/SchedLog.xyzzy
```

Using this example configuration, the *condor_master* starts up a second *condor_schedd* daemon, where this second *condor_schedd* daemon is passed **-local-name** *xyzzy* on the command line.

Continuing the example, configure the *condor_schedd* daemon named `xyzzy`. This *condor_schedd* daemon will share all configuration variable definitions with the other *condor_schedd* daemon, except for those specified separately.

```
SCHEDD.XYZZY.SCHEDD_NAME = XYZZY
SCHEDD.XYZZY.SCHEDD_LOG  = $(XYZZY_LOG)
SCHEDD.XYZZY.SPOOL       = $(SPOOL).XYZZY
```

Note that the example `SCHEDD_NAME` and `SPOOL` are specific to the *condor_schedd* daemon, as opposed to a different daemon such as the *condor_startd*. Other HTCondor daemons using this feature will have different requirements for which parameters need to be specified individually. This example works for the *condor_schedd*, and more local configuration can, and likely would be specified.

Also note that each daemon's log file must be specified individually, and in two places: one specification is for use by the *condor_master*, and the other is for use by the daemon itself. In the example, the `XYZZY` *condor_schedd* configuration variable `SCHEDD.XYZZY.SCHEDD_LOG` definition references the *condor_master* daemon's `XYZZY_LOG`.

5.3.4 Comments and Line Continuations

An HTCondor configuration file may contain comments and line continuations. A comment is any line beginning with a pound character (#). A continuation is any entry that continues across multiples lines. Line continuation is accomplished by placing the backslash character (\) at the end of any line to be continued onto another. Valid examples of line continuation are

```
START = (KeyboardIdle > 15 * $(MINUTE)) && \
((LoadAvg - CondorLoadAvg) <= 0.3)
```

and

```
ADMIN_MACHINES = condor.cs.wisc.edu, raven.cs.wisc.edu, \
stork.cs.wisc.edu, ostrich.cs.wisc.edu, \
bigbird.cs.wisc.edu
ALLOW_ADMINISTRATOR = $(ADMIN_MACHINES)
```

Where a line continuation character directly precedes a comment, the entire comment line is ignored, and the following line is used in the continuation. Line continuation characters within comments are ignored.

Both this example

```
A = $(B) \
# $(C)
$(D)
```

and this example

```
A = $(B) \
# $(C) \
$(D)
```

result in the same value for A:

```
A = $(B) $(D)
```

5.3.5 Multi-Line Values

As of version 8.5.6, the value for a macro can comprise multiple lines of text. The syntax for this is as follows:

```
<macro_name> @=<tag>
<macro_definition lines>
@<tag>
```

For example:

```
# modify routed job attributes:
# remove it if it goes on hold or stays idle for over 6 hours
JOB_ROUTER_DEFAULTS @=jrd
  [
    requirements = target.WantJobRouter is true;
    MaxIdleJobs = 10;
    MaxJobs = 200;

    set_PeriodicRemove = JobStatus == 5 || (JobStatus == 1 && (time() - QDate) > 3600*6);
    delete_WantJobRouter = true;
    set_requirements = true;
  ]
  @jrd
```

Note that in this example, the square brackets are part of the JOB_ROUTER_DEFAULTS value.

5.3.6 Executing a Program to Produce Configuration Macros

Instead of reading from a file, HTCondor can run a program to obtain configuration macros. The vertical bar character (|) as the last character defining a file name provides the syntax necessary to tell HTCondor to run a program. This syntax may only be used in the definition of the CONDOR_CONFIG environment variable, or the LOCAL_CONFIG_FILE configuration variable.

The command line for the program is formed by the characters preceding the vertical bar character. The standard output of the program is parsed as a configuration file would be.

An example:

```
LOCAL_CONFIG_FILE = /bin/make_the_config|
```

Program */bin/make_the_config* is executed, and its output is the set of configuration macros.

Note that either a program is executed to generate the configuration macros or the configuration is read from one or more files. The syntax uses space characters to separate command line elements, if an executed program produces the configuration macros. Space characters would otherwise separate the list of files. This syntax does not permit distinguishing one from the other, so only one may be specified.

(Note that the `include command` syntax (see below) is now the preferred way to execute a program to generate configuration macros.)

5.3.7 Including Configuration from Elsewhere

Externally defined configuration can be incorporated using the following syntax:

```
include [ifexist] : <file>
include : <cmdline>|
include [ifexist] command [into <cache-file>] : <cmdline>
```

(Note that the `ifexist` and `into` options were added in version 8.5.7. Also note that the command option must be specified in order to use the `into` option - just using the bar after <cmdline> will not work.)

In the file form of the `include` command, the <file> specification must describe a single file, the contents of which will be parsed and incorporated into the configuration. Unless the `ifexist` option is specified, the non-existence of the file is a fatal error.

In the command line form of the `include` command (specified with either the command option or by appending a bar (|) character after the <cmdline> specification), the <cmdline> specification must describe a command line (program and arguments); the command line will be executed, and the output will be parsed and incorporated into the configuration.

If the `into` option is not used, the command line will be executed every time the configuration file is referenced. This may well be undesirable, and can be avoided by using the `into` option. The `into` keyword must be followed by the full pathname of a file into which to write the output of the command line. If that file exists, it will be read and the command line will not be executed. If that file does not exist, the output of the command line will be written into it and then the cache file will be read and incorporated into the configuration. If the command line produces no output, a zero length file will be created. If the command line returns a non-zero exit code, configuration will abort and the cache file will not be created unless the `ifexist` keyword is also specified.

The `include` key word is case insensitive. There are no requirements for white space characters surrounding the colon character.

Consider the example

```
FILE = config.$(FULL_HOSTNAME)
include : $(LOCAL_DIR)/$(FILE)
```

Values are acquired for configuration variables `FILE`, and `LOCAL_DIR` by immediate evaluation, causing variable `FULL_HOSTNAME` to also be immediately evaluated. The resulting value forms a full path and file name. This file is read and parsed. The resulting configuration is incorporated into the current configuration. This resulting configuration may contain further nested `include` specifications, which are also parsed, evaluated, and incorporated. Levels of nested `include` are limited, such that infinite nesting is discovered and thwarted, while still permitting nesting.

Consider the further example

```
SCRIPT_FILE = script.$(IP_ADDRESS)
include : $(RELEASE_DIR)/$(SCRIPT_FILE) |
```

In this example, the bar character at the end of the line causes a script to be invoked, and the output of the script is incorporated into the current configuration. The same immediate parsing and evaluation occurs in this case as when a file's contents are included.

For pools that are transitioning to using this new syntax in configuration, while still having some tools and daemons with HTCondor versions earlier than 8.1.6, special syntax in the configuration will cause those daemons to fail upon

startup, rather than continuing, but incorrectly parsing the new syntax. Newer daemons will ignore the extra syntax. Placing the `@` character before the `include` key word causes the older daemons to fail when they attempt to parse this syntax.

Here is the same example, but with the syntax that causes older daemons to fail when reading it.

```
FILE = config.$(FULL_HOSTNAME)
@include : $(LOCAL_DIR)/$(FILE)
```

A daemon older than version 8.1.6 will fail to start. Running an older *condor_config_val* identifies the `@include` line as being bad. A daemon of HTCondor version 8.1.6 or more recent sees:

```
FILE = config.$(FULL_HOSTNAME)
include : $(LOCAL_DIR)/$(FILE)
```

and starts up successfully.

Here is an example using the new `ifexist` and `into` options:

```
# stuff.pl writes "STUFF=1" to stdout
include ifexist command into $(LOCAL_DIR)/stuff.config : perl $(LOCAL_DIR)/stuff.pl
```

5.3.8 Reporting Errors and Warnings

As of version 8.5.7, warning and error messages can be included in HTCondor configuration files.

The syntax for warning and error messages is as follows:

```
warning : <warning message>
error : <error message>
```

The warning and error messages will be printed when the configuration file is used (when almost any HTCondor command is run, for example). Error messages (unlike warnings) will prevent the successful use of the configuration file. This will, for example, prevent a daemon from starting, and prevent *condor_config_val* from returning a value.

Here's an example of using an error message in a configuration file (combined with some of the new include features documented above):

```
# stuff.pl writes "STUFF=1" to stdout
include command into $(LOCAL_DIR)/stuff.config : perl $(LOCAL_DIR)/stuff.pl
if ! defined stuff
  error : stuff is needed!
endif
```

5.3.9 Conditionals in Configuration

Conditional if/else semantics are available in a limited form. The syntax:

```
if <simple condition>
   <statement>
   . . .
   <statement>
else
   <statement>
   . . .
   <statement>
endif
```

An else key word and statements are not required, such that simple if semantics are implemented. The <simple condition> does not permit compound conditions. It optionally contains the exclamation point character (!) to represent the not operation, followed by

- the defined keyword followed by the name of a variable. If the variable is defined, the statement(s) are incorporated into the expanded input. If the variable is not defined, the statement(s) are not incorporated into the expanded input. As an example,

  ```
  if defined MY_UNDEFINED_VARIABLE
     X = 12
  else
     X = -1
  endif
  ```

  results in `X = -1`, when `MY_UNDEFINED_VARIABLE` is not yet defined.

- the version keyword, representing the version number of of the daemon or tool currently reading this conditional. This keyword is followed by an HTCondor version number. That version number can be of the form x.y.z or x.y. The version of the daemon or tool is compared to the specified version number. The comparison operators are

  - == for equality. Current version 8.2.3 is equal to 8.2.
  - >= to see if the current version number is greater than or equal to. Current version 8.2.3 is greater than 8.2.2, and current version 8.2.3 is greater than or equal to 8.2.
  - <= to see if the current version number is less than or equal to. Current version 8.2.0 is less than 8.2.2, and current version 8.2.3 is less than or equal to 8.2.

  As an example,

  ```
  if version >= 8.1.6
     DO_X = True
  else
     DO_Y = True
  endif
  ```

  results in defining `DO_X` as `True` if the current version of the daemon or tool reading this if statement is 8.1.6 or a more recent version.

- True or yes or the value 1. The statement(s) are incorporated.
- False or no or the value 0 The statement(s) are not incorporated.

- $(<variable>) may be used where the immediately evaluated value is a simple boolean value. A value that evaluates to the empty string is considered False, otherwise a value that does not evaluate to a simple boolean value is a syntax error.

The syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
elif <simple condition>
   <statement>
   . . .
   <statement>
endif
```

is the same as syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
else
   if <simple condition>
      <statement>
      . . .
      <statement>
   endif
endif
```

5.3.10 Function Macros in Configuration

A set of predefined functions increase flexibility. Both submit description files and configuration files are read using the same parser, so these functions may be used in both submit description files and configuration files.

Case is significant in the function's name, so use the same letter case as given in these definitions.

$CHOICE(index, listname) or $CHOICE(index, item1, item2, ...)
An item within the list is returned. The list is represented by a parameter name, or the list items are the parameters. The `index` parameter determines which item. The first item in the list is at index 0. If the index is out of bounds for the list contents, an error occurs.

$ENV(environment-variable-name[:default-value])
Evaluates to the value of environment variable `environment-variable-name`. If there is no environment variable with that name, Evaluates to UNDEFINED unless the optional :default-value is used; in which case it evaluates to default-value. For example,

```
A = $ENV(HOME)
```

binds `A` to the value of the `HOME` environment variable.

$F[fpduwnxbqa](filename)
One or more of the lower case letters may be combined to form the function name and thus, its functionality. Each letter operates on the `filename` in its own way.

- f convert relative path to full path by prefixing the current working directory to it. This option works only in *condor_submit* files.
- p refers to the entire directory portion of filename, with a trailing slash or backslash character. Whether a slash or backslash is used depends on the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified.
- d refers to the last portion of the directory within the path, if specified. It will have a trailing slash or backslash, as appropriate to the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified unless u or w is used. if b is used the trailing slash or backslash will be omitted.
- u convert path separators to Unix style slash characters
- w convert path separators to Windows style backslash characters
- n refers to the file name at the end of any path, but without any file name extension. As an example, the return value from $Fn(/tmp/simulate.exe) will be simulate (without the .exe extension).
- x refers to a file name extension, with the associated period (.). As an example, the return value from $Fn(/tmp/simulate.exe) will be .exe.
- b when combined with the d option, causes the trailing slash or backslash to be omitted. When combined with the x option, causes the leading period (.) to be omitted.
- q causes the return value to be enclosed within quotes. Double quote marks are used unless a is also specified.
- a When combined with the q option, causes the return value to be enclosed within single quotes.

$DIRNAME(filename) is the same as $Fp(filename)

$BASENAME(filename) is the same as $Fnx(filename)

$INT(item-to-convert) or $INT(item-to-convert, format-specifier)
Expands, evaluates, and returns a string version of item-to-convert. The format-specifier has the same syntax as a C language or Perl format specifier. If no format-specifier is specified, "%d" is used as the format specifier. The format is everything after the comma, including spaces. It can include other text.

```
X = 2
Y = 6
XYArea = $(X) * $(Y)
```

- $INT(XYArea) is 12
- $INT(XYArea,%04d) is 0012
- $INT(XYArea,Area=%d) is Area=12

$RANDOM_CHOICE(choice1, choice2, choice3, ...)
A random choice of one of the parameters in the list of parameters is made. For example, if one of the integers 0-8 (inclusive) should be randomly chosen:

```
$RANDOM_CHOICE(0,1,2,3,4,5,6,7,8)
```

$RANDOM_INTEGER(min, max [, step])
A random integer within the range min and max, inclusive, is selected. The optional step parameter controls the stride within the range, and it defaults to the value 1. For example, to randomly chose an even integer in the range 0-8 (inclusive):

```
$RANDOM_INTEGER(0, 8, 2)
```

$REAL(item-to-convert) or $REAL(item-to-convert, format-specifier)
Expands, evaluates, and returns a string version of `item-to-convert` for a floating point type. The `format-specifier` is a C language or Perl format specifier. If no `format-specifier` is specified, "%16G" is used as a format specifier.

$SUBSTR(name, start-index) or $SUBSTR(name, start-index, length)
Expands name and returns a substring of it. The first character of the string is at index 0. The first character of the substring is at index start-index. If the optional length is not specified, then the substring includes characters up to the end of the string. A negative value of start-index works back from the end of the string. A negative value of length eliminates use of characters from the end of the string. Here are some examples that all assume

```
Name = abcdef
```

- `$SUBSTR(Name, 2)` is `cdef`.
- `$SUBSTR(Name, 0, -2)` is `abcd`.
- `$SUBSTR(Name, 1, 3)` is `bcd`.
- `$SUBSTR(Name, -1)` is `f`.
- `$SUBSTR(Name, 4, -3)` is the empty string, as there are no characters in the substring for this request.

$STRING(item-to-convert) or $STRING(item-to-convert, format-specifier)
Expands, evaluates, and returns a string version of `item-to-convert` for a string type. The `format-specifier` is a C language or Perl format specifier. If no `format-specifier` is specified, "%s" is used as a format specifier. The format is everything after the comma, including spaces. It can include other text besides %s.

```
FULL_HOSTNAME = host.DOMAIN
LCFullHostname = toLower("$(FULL_HOSTNAME)")
```

- `$STRING(LCFullHostname)` is `host.domain`
- `$STRING(LCFullHostname,Name: %s)` is `Name: host.domain`

$EVAL(item-to-convert)
Expands, evaluates, and returns an classad unparsed version of `item-to-convert` for any classad type, the resulting value is formatted using the equivalent of the "%v" format specifier - If it is a string it is printed without quotes, otherwise it is unparsed as a classad value. Due to the way the parser works, you must use a variable to hold the expression to be evaluated if the expression has a close brace ')' character.

```
slist = "a,B,c"
lcslist = tolower($(slist))
list = split($(slist))
clist = size($(list)) * 10
semilist = join(";",split($(lcslist)))
```

- `$EVAL(slist)` is `a,B,c`
- `$EVAL(lcslist)` is `a,b,c`
- `$EVAL(list)` is `{"a", "B", "c"}`
- `$EVAL(clist)` is `30`
- `$EVAL(semilist)` is `a;b;c`

Environment references are not currently used in standard HTCondor configurations. However, they can sometimes be useful in custom configurations.

5.3.11 Macros That Will Require a Restart When Changed

The HTCondor daemons will generally not undo any work they have already done when the configuration changes so any change that would require undoing of work will require a restart before it takes effect. There a very few exceptions to this rule. The *condor_master* will pick up changes to `DAEMON_LIST` on a reconfig. Although it may take hours for a *condor_startd* to drain and exit when it is removed from the daemon list.

Examples of changes requiring a restart would any change to how HTCondor uses the network. A configuration change to `NETWORK_INTERFACE`, `NETWORK_HOSTNAME`, `ENABLE_IPV4` and `ENABLE_IPV6` require a restart. A change in the way daemons locate each other, such as `PROCD_ADDRESS`, `BIND_ALL_INTERFACES`, `USE_SHARED_PORT` or `SHARED_PORT_PORT` require a restart of the *condor_master* and all of the daemons under it.

The *condor_startd* requires a restart to make any change to the slot resource configuration, This would include `MEMORY`, `NUM_CPUS` and `NUM_SLOTS_TYPE_<n>`. It would also include resource detection like GPUs and Docker support. A general rule of thumb is that changes to the *condor_startd* require a restart, but there are a few exceptions. `STARTD_ATTRS` as well as `START`, `PREEMPT`, and other policy expressions take effect on reconfig.

For more information about specific configuration variables and whether a restart is required, refer to the documentation of the individual variables.

5.3.12 Pre-Defined Macros

HTCondor provides pre-defined macros that help configure HTCondor. Pre-defined macros are listed as `$(macro_name)`.

This first set are entries whose values are determined at run time and cannot be overwritten. These are inserted automatically by the library routine which parses the configuration files. This implies that a change to the underlying value of any of these variables will require a full restart of HTCondor in order to use the changed value.

$(FULL_HOSTNAME)
: The fully qualified host name of the local machine, which is host name plus domain name.

$(HOSTNAME)
: The host name of the local machine, without a domain name.

$(IP_ADDRESS)
: The ASCII string version of the local machine's "most public" IP address. This address may be IPv4 or IPv6, but the macro will always be set.

  HTCondor selects the "most public" address heuristically. Your configuration should not depend on HTCondor picking any particular IP address for this macro; this macro's value may not even be one of the IP addresses HTCondor is configured to advertise.

$(IPV4_ADDRESS)
: The ASCII string version of the local machine's "most public" IPv4 address; unset if the local machine has no IPv4 address.

  See `IP_ADDRESS` about "most public".

$(IPV6_ADDRESS)
: The ASCII string version of the local machine's "most public" IPv6 address; unset if the local machine has no IPv6 address.

See IP_ADDRESS about "most public".

$(IP_ADDRESS_IS_V6)
A boolean which is true if and only if IP_ADDRESS is an IPv6 address. Useful for conditional configuration.

$(TILDE)
The full path to the home directory of the Unix user condor, if such a user exists on the local machine.

$(SUBSYSTEM)
The subsystem name of the daemon or tool that is evaluating the macro. This is a unique string which identifies a given daemon within the HTCondor system. The possible subsystem names are:

GAHPs	C_GAHP	C_GAHP_WORKER_THREAD
	EC2_GAHP	GCE_GAHP
Daemons	MASTER	SHARED_PORT
	COLLECTOR	NEGOTIATOR
	SCHEDD	SHADOW
	STARTD	STARTER
	HAD	GRIDMANAGER
	KBDD	DEFRAG
	GANGLIAD	DAGMAN
	ROOSTER	
Other	REPLICATION	JOB_ROUTER
	SUBMIT	TOOL

$(DETECTED_CPUS)
The integer number of hyper-threaded CPUs, as given by $(DETECTED_CORES), when COUNT_HYPERTHREAD_CPUS is True. The integer number of physical (non hyper-threaded) CPUs, as given by $(DETECTED_PHYSICAL_CPUS), when COUNT_HYPERTHREAD_CPUS is False.

$(DETECTED_PHYSICAL_CPUS)
The integer number of physical (non hyper-threaded) CPUs. This will be equal the number of unique CPU IDs.

$(DETECTED_CPUS_LIMIT)
An integer value which is set to the minimum of $(DETECTED_CPUS) and values from the environment variables OMP_THREAD_LIMIT and SLURM_CPUS_ON_NODE. It intended for use as the value of NUM_CPUS to insure that the number of CPUS that a *condor_startd* will provision does not exceed the limits indicated by the environment. Defaults to $(DETECTED_CPUS) when there is no environment variable that sets a lower value.

This second set of macros are entries whose default values are determined automatically at run time but which can be overwritten.

$(ARCH)
Defines the string used to identify the architecture of the local machine to HTCondor. The *condor_startd* will advertise itself with this attribute so that users can submit binaries compiled for a given platform and force them to run on the correct machines. *condor_submit* will append a requirement to the job ClassAd that it must run on the same ARCH and OPSYS of the machine where it was submitted, unless the user specifies ARCH and/or OPSYS explicitly in their submit file. See the *condor_submit* manual page (doc:*/man-pages/condor_submit*) for details.

$(OPSYS)
Defines the string used to identify the operating system of the local machine to HTCondor. If it is not defined in the configuration file, HTCondor will automatically insert the operating system of this machine as determined by *uname*.

$(OPSYS_VER)
Defines the integer used to identify the operating system version number.

$(OPSYS_AND_VER)
Defines the string used prior to HTCondor version 7.7.2 as $(OPSYS).

$(UNAME_ARCH)
The architecture as reported by *uname* (2)'s `machine` field. Always the same as `ARCH` on Windows.

$(UNAME_OPSYS)
The operating system as reported by *uname* (2)'s `sysname` field. Always the same as `OPSYS` on Windows.

$(DETECTED_MEMORY)
The amount of detected physical memory (RAM) in MiB.

$(DETECTED_CORES)
The number of CPU cores that the operating system schedules. On machines that support hyper-threading, this will be the number of hyper-threads.

$(PID)
The process ID for the daemon or tool.

$(PPID)
The process ID of the parent process for the daemon or tool.

$(USERNAME)
The user name of the UID of the daemon or tool. For daemons started as root, but running under another UID (typically the user condor), this will be the other UID.

$(FILESYSTEM_DOMAIN)
Defaults to the fully qualified host name of the machine it is evaluated on. See the *Configuration Macros* section, Shared File System Configuration File Entries for the full description of its use and under what conditions it could be desirable to change it.

$(UID_DOMAIN)
Defaults to the fully qualified host name of the machine it is evaluated on. See the *Configuration Macros* section for the full description of this configuration variable.

$(CONFIG_ROOT)
Set to the directory where the the main config file will be read prior to reading any config files. The value will usually be `/etc/condor` for an RPM install, `C:\Condor` for a Windows MSI install and the directory part of the `CONDOR_CONFIG` environment variable for a tarball install. This variable will not be set when `CONDOR_CONFIG` is set to `ONLY_ENV` so that no configuration files are read.

Since $(ARCH) and $(OPSYS) will automatically be set to the correct values, we recommend that you do not overwrite them.

5.4 Configuration Templates

Achieving certain behaviors in an HTCondor pool often requires setting the values of a number of configuration macros in concert with each other. We have added configuration templates as a way to do this more easily, at a higher level, without having to explicitly set each individual configuration macro.

Configuration templates are pre-defined; users cannot define their own templates.

Note that the value of an individual configuration macro that is set by a configuration template can be overridden by setting that configuration macro later in the configuration.

Detailed information about configuration templates (such as the macros they set) can be obtained using the *condor_config_val* `use` option (see the *condor_config_val* manual page). (This document does not contain such information because the *condor_config_val* command is a better way to obtain it.)

5.4.1 Configuration Templates: Using Predefined Sets of Configuration

Predefined sets of configuration can be identified and incorporated into the configuration using the syntax

```
use <category name> : <template name>
```

The `use` key word is case insensitive. There are no requirements for white space characters surrounding the colon character. More than one `<template name>` identifier may be placed within a single `use` line. Separate the names by a space character. There is no mechanism by which the administrator may define their own custom `<category name>` or `<template name>`.

Each predefined `<category name>` has a fixed, case insensitive name for the sets of configuration that are predefined. Placement of a `use` line in the configuration brings in the predefined configuration it identifies.

As of version 8.5.6, some of the configuration templates take arguments (as described below).

5.4.2 Available Configuration Templates

There are four `<category name>` values. Within a category, a predefined, case insensitive name identifies the set of configuration it incorporates.

`ROLE category`
Describes configuration for the various roles that a machine might play within an HTCondor pool. The configuration will identify which daemons are running on a machine.

- `Personal`

  Settings needed for when a single machine is the entire pool.

- `Submit`

  Settings needed to allow this machine to submit jobs to the pool. May be combined with `Execute` and `CentralManager` roles.

- `Execute`

  Settings needed to allow this machine to execute jobs. May be combined with `Submit` and `CentralManager` roles.

- `CentralManager`

  Settings needed to allow this machine to act as the central manager for the pool. May be combined with `Submit` and `Execute` roles.

`FEATURE category`
Describes configuration for implemented features.

- `Remote_Runtime_Config`

  Enables the use of *condor_config_val* **-rset** to the machine with this configuration. Note that there are security implications for use of this configuration, as it potentially permits the arbitrary modification of configuration. Variable `SETTABLE_ATTRS_CONFIG` must also be defined.

- `Remote_Config`

  Enables the use of *condor_config_val* **-set** to the machine with this configuration. Note that there are security implications for use of this configuration, as it potentially permits the arbitrary modification of configuration. Variable `SETTABLE_ATTRS_CONFIG` must also be defined.

- `GPUs([discovery_args])`

  Sets configuration based on detection with the *condor_gpu_discovery* tool, and defines a custom resource using the name `GPUs`. Supports both OpenCL and CUDA, if detected. Automatically includes the `GPUsMonitor` feature. Optional discovery_args are passed to *condor_gpu_discovery*

- `GPUsMonitor`

  Also adds configuration to report the usage of NVidia GPUs.

- `Monitor( resource_name, mode, period, executable, metric[, metric]+ )`

  Configures a custom machine resource monitor with the given name, mode, period, executable, and metrics. See *Startd Cron and Schedd Cron* for the definitions of these terms.

- `PartitionableSlot( slot_type_num [, allocation] )`

  Sets up a partitionable slot of the specified slot type number and allocation (defaults for slot_type_num and allocation are 1 and 100% respectively). See the *condor_startd Policy Configuration* for information on partitionable slot policies.

- `StaticSlots( slot_type_num [, num_slots, [, allocation] ] )`

  Sets up a number of static slots of the specified slot type number (defaults for slot_type_num and num_slots are 1 and `$(NUM_CPUS)` respectively). The number of slots will be equal to `num_slots`. If no value is provided for the allocation, the default is to divide 100% of the machine resources evenly across the slots.

- `AssignAccountingGroup( map_filename [, check_request] )` Sets up a *condor_schedd* job transform that assigns an accounting group to each job as it is submitted. The accounting group is determined by mapping the Owner attribute of the job using the given map file, which should specify the allowed accounting groups each Owner is permitted to use. If the submitted job has an accounting group, that is treated as a requested accounting group and validated against the map. If the optional `check_request` argument is true or not present submission will fail if the requested accounting group is present and not valid. If the argument is false, the requested accounting group will be ignored if it is not valid.

- `ScheddUserMapFile( map_name, map_filename )` Defines a *condor_schedd* usermap named map_name using the given map file.

- `SetJobAttrFromUserMap( dst_attr, src_attr, map_name [, map_filename] )` Sets up a *condor_schedd* job transform that sets the dst_attr attribute of each job as it is submitted. The value of dst_attr is determined by mapping the src_attr of the job using the usermap named map_name. If the optional map_filename argument is specified, then this metaknob also defines a *condor_schedd* usermap named map_Name using the given map file.

- `StartdCronOneShot( job_name, exe [, hook_args] )`

  Create a one-shot *condor_startd* job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `StartdCronPeriodic( job_name, period, exe [, hook_args] )`

  Create a periodic-shot *condor_startd* job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `StartdCronContinuous( job_name, exe [, hook_args] )`

  Create a (nearly) continuous *condor_startd* job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `ScheddCronOneShot( job_name, exe [, hook_args] )`

  Create a one-shot *condor_schedd* job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `ScheddCronPeriodic( job_name, period, exe [, hook_args] )`

  Create a periodic-shot *condor_schedd* job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `ScheddCronContinuous( job_name, exe [, hook_args] )`

  Create a (nearly) continuous *condor_schedd* job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `OneShotCronHook( STARTD_CRON | SCHEDD_CRON, job_name, hook_exe [,hook_args] )`

  Create a one-shot job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `PeriodicCronHook( STARTD_CRON | SCHEDD_CRON , job_name, period, hook_exe [, hook_args] )`

  Create a periodic job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `ContinuousCronHook( STARTD_CRON | SCHEDD_CRON , job_name, hook_exe [,hook_args] )`

  Create a (nearly) continuous job hook. (See *Startd Cron and Schedd Cron* for more information about job hooks.)

- `OAuth`

  Sets configuration that enables the *condor_credd* and *condor_credmon_oauth* daemons, which allow for the automatic renewal of user-supplied OAuth2 credentials. See section *Enabling the Fetching and Use of OAuth2 Credentials* for more information.

- `Adstash`

  Sets configuration that enables *condor_adstash* to run as a daemon. *condor_adstash* polls job history ClassAds and pushes them to an Elasticsearch index, see section *Elasticsearch* for more information.

- `UWCS_Desktop_Policy_Values`

  Configuration values used in the `UWCS_DESKTOP` policy. (Note that these values were previously in the parameter table; configuration that uses these values will have to use the `UWCS_Desktop_Policy_Values` template. For example, `POLICY : UWCS_Desktop` uses the `FEATURE : UWCS_Desktop_Policy_Values` template.)

- `CommonCloudAttributesAWS`
- `CommonCloudAttributesGoogle`

  Sets configuration that will put some common cloud-related attributes in the slot ads. Use the version which specifies the cloud you're using. See *Common Cloud Attributes* for details.

- `JobsHaveInstanceIDs`

  Sets configuration that will cause job ads to track the instance IDs of slots that they ran on (if available).

`POLICY category`

Describes configuration for the circumstances under which machines choose to run jobs.

- `Always_Run_Jobs`

  Always start jobs and run them to completion, without consideration of *condor_negotiator* generated preemption or suspension. This is the default policy, and it is intended to be used with dedicated resources. If this policy is used together with the `Limit_Job_Runtimes` policy, order the specification by placing this `Always_Run_Jobs` policy first.

- UWCS_Desktop

  This was the default policy before HTCondor version 8.1.6. It is intended to be used with desktop machines not exclusively running HTCondor jobs. It injects UWCS into the name of some configuration variables.

- Desktop

  An updated and re-implementation of the UWCS_Desktop policy, but without the UWCS naming of some configuration variables.

- Limit_Job_Runtimes(limit_in_seconds)

  Limits running jobs to a maximum of the specified time using preemption. (The default limit is 24 hours.) This policy does not work while the machine is draining; use the following policy instead.

  If this policy is used together with the Always_Run_Jobs policy, order the specification by placing this Limit_Job_Runtimes policy second.

- Preempt_if_Runtime_Exceeds(limit_in_seconds)

  Limits running jobs to a maximum of the specified time using preemption. (The default limit is 24 hours).

- Hold_if_Runtime_Exceeds(limit_in_seconds)

  Limits running jobs to a maximum of the specified time by placing them on hold immediately (ignoring any job retirement time). (The default limit is 24 hours).

- Preempt_If_Cpus_Exceeded

  If the startd observes the number of CPU cores used by the job exceed the number of cores in the slot by more than 0.8 on average over the past minute, preempt the job immediately ignoring any job retirement time.

- Hold_If_Cpus_Exceeded

  If the startd observes the number of CPU cores used by the job exceed the number of cores in the slot by more than 0.8 on average over the past minute, immediately place the job on hold ignoring any job retirement time. The job will go on hold with a reasonable hold reason in job attribute HoldReason and a value of 101 in job attribute HoldReasonCode. The hold reason and code can be customized by specifying HOLD_REASON_CPU_EXCEEDED and HOLD_SUBCODE_CPU_EXCEEDED respectively.

- Preempt_If_Disk_Exceeded

  If the startd observes the amount of disk space used by the job exceed the disk in the slot, preempt the job immediately ignoring any job retirement time.

- Hold_If_Disk_Exceeded

  If the startd observes the amount of disk space used by the job exceed the disk in the slot, immediately place the job on hold ignoring any job retirement time. The job will go on hold with a reasonable hold reason in job attribute HoldReason and a value of 104 in job attribute HoldReasonCode. The hold reason and code can be customized by specifying HOLD_REASON_DISK_EXCEEDED and HOLD_SUBCODE_DISK_EXCEEDED respectively.

- Preempt_If_Memory_Exceeded

  If the startd observes the memory usage of the job exceed the memory provisioned in the slot, preempt the job immediately ignoring any job retirement time.

- Hold_If_Memory_Exceeded

  If the startd observes the memory usage of the job exceed the memory provisioned in the slot, immediately place the job on hold ignoring any job retirement time. The job will go on hold with a reasonable hold reason in job attribute HoldReason and a value of 102 in job attribute HoldReasonCode.

The hold reason and code can be customized by specifying `HOLD_REASON_MEMORY_EXCEEDED` and `HOLD_SUBCODE_MEMORY_EXCEEDED` respectively.

- `Preempt_If( policy_variable )`

  Preempt jobs according to the specified policy. `policy_variable` must be the name of a configuration macro containing an expression that evaluates to `True` if the job should be preempted.

  See an example here: *Configuration Template Examples*.

- `Want_Hold_If( policy_variable, subcode, reason_text )`

  Add the given policy to the `WANT_HOLD` expression; if the `WANT_HOLD` expression is defined, `policy_variable` is prepended to the existing expression; otherwise `WANT_HOLD` is simply set to the value of the policy_variable macro.

  See an example here: *Configuration Template Examples*.

- `Startd_Publish_CpusUsage`

  Publish the number of CPU cores being used by the job into the slot ad as attribute `CpusUsage`. This value will be the average number of cores used by the job over the past minute, sampling every 5 seconds.

`SECURITY category`
Describes configuration for an implemented security model.

- `Host_Based`

  The default security model (based on IPs and DNS names). Do not combine with `User_Based` security.

- `User_Based`

  Grants permissions to an administrator and uses `With_Authentication`. Do not combine with `Host_Based` security.

- `With_Authentication`

  Requires both authentication and integrity checks.

- `Strong`

  Requires authentication, encryption, and integrity checks.

5.4.3 Configuration Template Transition Syntax

For pools that are transitioning to using this new syntax in configuration, while still having some tools and daemons with HTCondor versions earlier than 8.1.6, special syntax in the configuration will cause those daemons to fail upon start up, rather than use the new, but misinterpreted, syntax. Newer daemons will ignore the extra syntax. Placing the @ character before the `use` key word causes the older daemons to fail when they attempt to parse this syntax.

As an example, consider the *condor_startd* as it starts up. A *condor_startd* previous to HTCondor version 8.1.6 fails to start when it sees:

```
@use feature : GPUs
```

Running an older *condor_config_val* also identifies the `@use` line as being bad. A *condor_startd* of HTCondor version 8.1.6 or more recent sees

```
use feature : GPUs
```

5.4.4 Configuration Template Examples

- Preempt a job if its memory usage exceeds the requested memory:

```
MEMORY_EXCEEDED = (isDefined(MemoryUsage) && MemoryUsage > RequestMemory)
use POLICY : PREEMPT_IF(MEMORY_EXCEEDED)
```

- Put a job on hold if its memory usage exceeds the requested memory:

```
MEMORY_EXCEEDED = (isDefined(MemoryUsage) && MemoryUsage > RequestMemory)
use POLICY : WANT_HOLD_IF(MEMORY_EXCEEDED, 102, memory usage exceeded request_
↪memory)
```

- Update dynamic GPU information every 15 minutes:

```
use FEATURE : StartdCronPeriodic(DYNGPU, 15*60, $(LOCAL_DIR)\dynamic_gpu_info.pl,
↪$(LIBEXEC)\condor_gpu_discovery -dynamic)
```

  where `dynamic_gpu_info.pl` is a simple perl script that strips off the DetectedGPUs line from *condor_gpu_discovery*:

```
#!/usr/bin/env perl
my @attrs = `@ARGV`;
for (@attrs) {
    next if ($_ =~ /^Detected/i);
    print $_;
}
```

5.5 Configuration Macros

The section contains a list of the individual configuration macros for HTCondor. Before attempting to set up HTCondor configuration, you should probably read the *Introduction to Configuration* section and possibly the *Configuration Templates* section.

The settings that control the policy under which HTCondor will start, suspend, resume, vacate or kill jobs are described in *condor_startd Policy Configuration*, not in this section.

5.5.1 HTCondor-wide Configuration File Entries

This section describes settings which affect all parts of the HTCondor system. Other system-wide settings can be found in *Network-Related Configuration File Entries* and *Shared File System Configuration File Macros*.

SUBSYSTEM
: Various configuration macros described below may include <SUBSYS> in the macro name. This allows for one general macro name to apply to specific subsystems via a common pattern. Just replace the <SUBSYS> part of the given macro with a valid HTCondor subsystem name to apply that macro. Note that some configuration macros with <SUBSYS> only work for select subsystems. List of HTCondor Subsystems:

GAHPs	C_GAHP	C_GAHP_WORKER_THREAD
	EC2_GAHP	GCE_GAHP
Daemons	MASTER	SHARED_PORT
	COLLECTOR	NEGOTIATOR
	SCHEDD	SHADOW
	STARTD	STARTER
	HAD	GRIDMANAGER
	KBDD	DEFRAG
	GANGLIAD	DAGMAN
	ROOSTER	
Other	REPLICATION	JOB_ROUTER
	SUBMIT	TOOL

CONDOR_HOST
This macro is used to define the `$(COLLECTOR_HOST)` macro. Normally the *condor_collector* and *condor_negotiator* would run on the same machine. If for some reason they were not run on the same machine, `$(CONDOR_HOST)` would not be needed. Some of the host-based security macros use `$(CONDOR_HOST)` by default. See the *Host-Based Security in HTCondor* section on Setting up IP/host-based security in HTCondor for details.

COLLECTOR_HOST
The host name of the machine where the *condor_collector* is running for your pool. Normally, it is defined relative to the `$(CONDOR_HOST)` macro. There is no default value for this macro; `COLLECTOR_HOST` must be defined for the pool to work properly.

In addition to defining the host name, this setting can optionally be used to specify the network port of the *condor_collector*. The port is separated from the host name by a colon (':'). For example,

```
COLLECTOR_HOST = $(CONDOR_HOST):1234
```

If no port is specified, the default port of 9618 is used. Using the default port is recommended for most sites. It is only changed if there is a conflict with another service listening on the same network port. For more information about specifying a non-standard port for the *condor_collector* daemon, see *Port Usage in HTCondor*.

Multiple *condor_collector* daemons may be running simultaneously, if `COLLECTOR_HOST` is defined with a comma separated list of hosts. Multiple *condor_collector* daemons may run for the implementation of high availability; see *The High Availability of Daemons* for details. With more than one running, updates are sent to all. With more than one running, queries are sent to one of the *condor_collector* daemons, chosen at random.

COLLECTOR_PORT
The default port used when contacting the *condor_collector* and the default port the *condor_collector* listens on if no port is specified. This variable is referenced if no port is given and there is no other means to find the *condor_collector* port. The default value is 9618.

NEGOTIATOR_HOST
This configuration variable is no longer used. It previously defined the host name of the machine where the *condor_negotiator* is running. At present, the port where the *condor_negotiator* is listening is dynamically allocated.

CONDOR_VIEW_HOST
A list of HTCondorView servers, separated by commas and/or spaces. Each HTCondorView server is denoted by the host name of the machine it is running on, optionally appended by a colon and the port number. This service is optional, and requires additional configuration to enable it. There is no default value for `CONDOR_VIEW_HOST`. If `CONDOR_VIEW_HOST` is not defined, no HTCondorView server is used. See *Configuring The HTCondorView Server* for more details.

SCHEDD_HOST
The host name of the machine where the *condor_schedd* is running for your pool. This is the host that queues submitted jobs. If the host specifies `SCHEDD_NAME` or `MASTER_NAME` , that name must be included in the form name@hostname. In most condor installations, there is a *condor_schedd* running on each host from which jobs are submitted. The default value of `SCHEDD_HOST` is the current host with the optional name included. For most pools, this macro is not defined, nor does it need to be defined..

RELEASE_DIR
The full path to the HTCondor release directory, which holds the `bin`, `etc`, `lib`, and `sbin` directories. Other macros are defined relative to this one. There is no default value for `RELEASE_DIR` .

BIN
This directory points to the HTCondor directory where user-level programs are installed. The default value is `$(RELEASE_DIR)`/bin.

LIB
This directory points to the HTCondor directory containing its libraries. On Windows, libraries are located in `BIN`.

LIBEXEC
This directory points to the HTCondor directory where support commands that HTCondor needs will be placed. Do not add this directory to a user or system-wide path.

INCLUDE
This directory points to the HTCondor directory where header files reside. The default value is `$(RELEASE_DIR)`/include. It can make inclusion of necessary header files for compilation of programs (such as those programs that use `libcondorapi.a`) easier through the use of *condor_config_val*.

SBIN
This directory points to the HTCondor directory where HTCondor's system binaries (such as the binaries for the HTCondor daemons) and administrative tools are installed. Whatever directory `$(SBIN)` points to ought to be in the `PATH` of users acting as HTCondor administrators. The default value is `$(BIN)` in Windows and `$(RELEASE_DIR)`/sbin on all other platforms.

LOCAL_DIR
The location of the local HTCondor directory on each machine in your pool. The default value is `$(RELEASE_DIR)` on Windows and `$(RELEASE_DIR)`/hosts/`$(HOSTNAME)` on all other platforms.

Another possibility is to use the condor user's home directory, which may be specified with `$(TILDE)`. For example:

```
LOCAL_DIR = $(tilde)
```

LOG
Used to specify the directory where each HTCondor daemon writes its log files. The names of the log files themselves are defined with other macros, which use the `$(LOG)` macro by default. The log directory also acts as the current working directory of the HTCondor daemons as the run, so if one of them should produce a core file for any reason, it would be placed in the directory defined by this macro. The default value is `$(LOCAL_DIR)`/log.

Do not stage other files in this directory; any files not created by HTCondor in this directory are subject to removal.

RUN
A path and directory name to be used by the HTCondor init script to specify the directory where the *condor_master* should write its process ID (PID) file. The default if not defined is `$(LOG)`.

SPOOL
The spool directory is where certain files used by the *condor_schedd* are stored, such as the job queue file. The spool also stores all input and output files for remotely-submitted jobs and all intermediate or checkpoint files. Therefore, you will want to ensure that the spool directory is located on a partition with enough disk space. If a

given machine is only set up to execute HTCondor jobs and not submit them, it would not need a spool directory (or this macro defined). The default value is $(LOCAL_DIR)/spool. The *condor_schedd* will not function if SPOOL is not defined.

Do not stage other files in this directory; any files not created by HTCondor in this directory are subject to removal.

EXECUTE
This directory acts as a place to create the scratch directory of any HTCondor job that is executing on the local machine. The scratch directory is the destination of any input files that were specified for transfer. It also serves as the job's working directory if the job is using file transfer mode and no other working directory was specified. If a given machine is set up to only submit jobs and not execute them, it would not need an execute directory, and this macro need not be defined. The default value is $(LOCAL_DIR)/execute. The *condor_startd* will not function if EXECUTE is undefined. To customize the execute directory independently for each batch slot, use SLOT<N>_EXECUTE.

Do not stage other files in this directory; any files not created by HTCondor in this directory are subject to removal.

Ideally, this directory should not be placed under /tmp or /var/tmp, if it is, HTCondor loses the ability to make private instances of /tmp and /var/tmp for jobs.

TMP_DIR
A directory path to a directory where temporary files are placed by various portions of the HTCondor system. The daemons and tools that use this directory are the *condor_gridmanager*, *condor_config_val* when using the **-rset** option, systems that use lock files when configuration variable CREATE_LOCKS_ON_LOCAL_DISK is True, the Web Service API, and the *condor_credd* daemon. There is no default value.

If both TMP_DIR and TEMP_DIR are defined, the value set for TMP_DIR is used and TEMP_DIR is ignored.

TEMP_DIR
A directory path to a directory where temporary files are placed by various portions of the HTCondor system. The daemons and tools that use this directory are the *condor_gridmanager*, *condor_config_val* when using the **-rset** option, systems that use lock files when configuration variable CREATE_LOCKS_ON_LOCAL_DISK is True, the Web Service API, and the *condor_credd* daemon. There is no default value.

If both TMP_DIR and TEMP_DIR are defined, the value set for TMP_DIR is used and TEMP_DIR is ignored.

SLOT<N>_EXECUTE
Specifies an execute directory for use by a specific batch slot. <N> represents the number of the batch slot, such as 1, 2, 3, etc. This execute directory serves the same purpose as EXECUTE , but it allows the configuration of the directory independently for each batch slot. Having slots each using a different partition would be useful, for example, in preventing one job from filling up the same disk that other jobs are trying to write to. If this parameter is undefined for a given batch slot, it will use EXECUTE as the default. Note that each slot will advertise TotalDisk and Disk for the partition containing its execute directory.

LOCAL_CONFIG_FILE
Identifies the location of the local, machine-specific configuration file for each machine in the pool. The two most common choices would be putting this file in the $(LOCAL_DIR), or putting all local configuration files for the pool in a shared directory, each one named by host name. For example,

```
LOCAL_CONFIG_FILE = $(LOCAL_DIR)/condor_config.local
```

or,

```
LOCAL_CONFIG_FILE = $(release_dir)/etc/$(hostname).local
```

or, not using the release directory

```
LOCAL_CONFIG_FILE = /full/path/to/configs/$(hostname).local
```

The value of LOCAL_CONFIG_FILE is treated as a list of files, not a single file. The items in the list are delimited by either commas or space characters. This allows the specification of multiple files as the local configuration file, each one processed in the order given (with parameters set in later files overriding values from previous files). This allows the use of one global configuration file for multiple platforms in the pool, defines a platform-specific configuration file for each platform, and uses a local configuration file for each machine. If the list of files is changed in one of the later read files, the new list replaces the old list, but any files that have already been processed remain processed, and are removed from the new list if they are present to prevent cycles. See *Executing a Program to Produce Configuration Macros* for directions on using a program to generate the configuration macros that would otherwise reside in one or more files as described here. If LOCAL_CONFIG_FILE is not defined, no local configuration files are processed. For more information on this, see *Configuring HTCondor for Multiple Platforms*.

If all files in a directory are local configuration files to be processed, then consider using .

REQUIRE_LOCAL_CONFIG_FILE
A boolean value that defaults to True. When True, HTCondor exits with an error, if any file listed in LOCAL_CONFIG_FILE cannot be read. A value of False allows local configuration files to be missing. This is most useful for sites that have both large numbers of machines in the pool and a local configuration file that uses the $(HOSTNAME) macro in its definition. Instead of having an empty file for every host in the pool, files can simply be omitted.

LOCAL_CONFIG_DIR
A directory may be used as a container for local configuration files. The files found in the directory are sorted into lexicographical order by file name, and then each file is treated as though it was listed in LOCAL_CONFIG_FILE. LOCAL_CONFIG_DIR is processed before any files listed in LOCAL_CONFIG_FILE, and is checked again after processing the LOCAL_CONFIG_FILE list. It is a list of directories, and each directory is processed in the order it appears in the list. The process is not recursive, so any directories found inside the directory being processed are ignored. See also LOCAL_CONFIG_DIR_EXCLUDE_REGEXP.

USER_CONFIG_FILE
The file name of a configuration file to be parsed after other local configuration files and before environment variables set configuration. Relevant only if HTCondor daemons are not run as root on Unix platforms or Local System on Windows platforms. The default is $(HOME)/.condor/user_config on Unix platforms. The default is %USERPROFILE\.condor\user_config on Windows platforms. If a fully qualified path is given, that is used. If a fully qualified path is not given, then the Unix path $(HOME)/.condor/ prefixes the file name given on Unix platforms, or the Windows path %USERPROFILE\.condor\ prefixes the file name given on Windows platforms.

The ability of a user to use this user-specified configuration file can be disabled by setting this variable to the empty string:

```
USER_CONFIG_FILE =
```

LOCAL_CONFIG_DIR_EXCLUDE_REGEXP
A regular expression that specifies file names to be ignored when looking for configuration files within the directories specified via LOCAL_CONFIG_DIR. The default expression ignores files with names beginning with a '.' or a '#', as well as files with names ending in '~'. This avoids accidents that can be caused by treating temporary files created by text editors as configuration files.

CONDOR_IDS
The User ID (UID) and Group ID (GID) pair that the HTCondor daemons should run as, if the daemons are spawned as root. This value can also be specified in the CONDOR_IDS environment variable. If the HTCondor daemons are not started as root, then neither this CONDOR_IDS configuration macro nor the CONDOR_IDS environment variable are used. The value is given by two integers, separated by a period. For example, CONDOR_IDS = 1234.1234. If this pair is not specified in either the configuration file or in the environment, and the HTCondor

daemons are spawned as root, then HTCondor will search for a condor user on the system, and run as that user's UID and GID. See *User Accounts in HTCondor on Unix Platforms* on UIDs in HTCondor for more details.

CONDOR_ADMIN
The email address that HTCondor will send mail to if something goes wrong in the pool. For example, if a daemon crashes, the *condor_master* can send an obituary to this address with the last few lines of that daemon's log file and a brief message that describes what signal or exit status that daemon exited with. The default value is root@$(FULL_HOSTNAME).

<SUBSYS>_ADMIN_EMAIL
The email address that HTCondor will send mail to if something goes wrong with the named <SUBSYS>. Identical to CONDOR_ADMIN, but done on a per subsystem basis. There is no default value.

List of possible subsystems to set <SUBSYS> can be found at .

CONDOR_SUPPORT_EMAIL
The email address to be included at the bottom of all email HTCondor sends out under the label "Email address of the local HTCondor administrator:". This is the address where HTCondor users at your site should send their questions about HTCondor and get technical support. If this setting is not defined, HTCondor will use the address specified in CONDOR_ADMIN (described above).

EMAIL_SIGNATURE
Every e-mail sent by HTCondor includes a short signature line appended to the body. By default, this signature includes the URL to the global HTCondor project website. When set, this variable defines an alternative signature line to be used instead of the default. Note that the value can only be one line in length. This variable could be used to direct users to look at local web site with information specific to the installation of HTCondor.

MAIL
The full path to a mail sending program that uses **-s** to specify a subject for the message. On all platforms, the default shipped with HTCondor should work. Only if you installed things in a non-standard location on your system would you need to change this setting. The default value is $(BIN)/condor_mail.exe on Windows and /usr/bin/mail on all other platforms. The *condor_schedd* will not function unless MAIL is defined. For security reasons, non-Windows platforms should not use this setting and should use SENDMAIL instead.

SENDMAIL
The full path to the *sendmail* executable. If defined, which it is by default on non-Windows platforms, *sendmail* is used instead of the mail program defined by MAIL.

MAIL_FROM
The e-mail address that notification e-mails appear to come from. Contents is that of the From header. There is no default value; if undefined, the From header may be nonsensical.

SMTP_SERVER
For Windows platforms only, the host name of the server through which to route notification e-mail. There is no default value; if undefined and the debug level is at FULLDEBUG, an error message will be generated.

RESERVED_SWAP
The amount of swap space in MiB to reserve for this machine. HTCondor will not start up more *condor_shadow* processes if the amount of free swap space on this machine falls below this level. The default value is 0, which disables this check. It is anticipated that this configuration variable will no longer be used in the near future. If RESERVED_SWAP is not set to 0, the value of SHADOW_SIZE_ESTIMATE is used.

DISK
Tells HTCondor how much disk space (in kB) to advertise as being available for use by jobs. If DISK is not specified, HTCondor will advertise the amount of free space on your execute partition, minus RESERVED_DISK.

RESERVED_DISK
Determines how much disk space (in MB) you want to reserve for your own machine. When HTCondor is reporting the amount of free disk space in a given partition on your machine, it will always subtract this amount.

An example is the *condor_startd*, which advertises the amount of free space in the `$(EXECUTE)` directory. The default value of `RESERVED_DISK` is zero.

LOCK
HTCondor needs to create lock files to synchronize access to various log files. Because of problems with network file systems and file locking over the years, we highly recommend that you put these lock files on a local partition on each machine. If you do not have your `$(LOCAL_DIR)` on a local partition, be sure to change this entry.

Whatever user or group HTCondor is running as needs to have write access to this directory. If you are not running as root, this is whatever user you started up the *condor_master* as. If you are running as root, and there is a condor account, it is most likely condor. Otherwise, it is whatever you set in the `CONDOR_IDS` environment variable, or whatever you define in the `CONDOR_IDS` setting in the HTCondor config files. See *User Accounts in HTCondor on Unix Platforms* on UIDs in HTCondor for details.

If no value for `LOCK` is provided, the value of `LOG` is used.

HISTORY
Defines the location of the HTCondor history file, which stores information about all HTCondor jobs that have completed on a given machine. This macro is used by both the *condor_schedd* which appends the information and *condor_history*, the user-level program used to view the history file. This configuration macro is given the default value of `$(SPOOL)/history` in the default configuration. If not defined, no history file is kept.

ENABLE_HISTORY_ROTATION
If this is defined to be true, then the history file will be rotated. If it is false, then it will not be rotated, and it will grow indefinitely, to the limits allowed by the operating system. If this is not defined, it is assumed to be true. The rotated files will be stored in the same directory as the history file.

MAX_HISTORY_LOG
Defines the maximum size for the history file, in bytes. It defaults to 20MB. This parameter is only used if history file rotation is enabled.

MAX_HISTORY_ROTATIONS
When history file rotation is turned on, this controls how many backup files there are. It default to 2, which means that there may be up to three history files (two backups, plus the history file that is being currently written to). When the history file is rotated, and this rotation would cause the number of backups to be too large, the oldest file is removed.

HISTORY_CONTAINS_JOB_ENVIRONMENT
This parameter defaults to true. When set to false, the job's environment attribute (which can be very large) is not written to the history file. This may allow many more jobs to be kept in the history before rotation.

HISTORY_HELPER_MAX_CONCURRENCY
Specifies the maximum number of concurrent remote *condor_history* queries allowed at a time; defaults to 50. When this maximum is exceeded, further queries will be queued in a non-blocking manner. Setting this option to 0 disables remote history access. A remote history access is defined as an invocation of *condor_history* that specifies a **-name** option to query a *condor_schedd* running on a remote machine.

HISTORY_HELPER_MAX_HISTORY
Specifies the maximum number of ClassAds to parse on behalf of remote history clients. The default is 10,000. This allows the system administrator to indirectly manage the maximum amount of CPU time spent on each client. Setting this option to 0 disables remote history access.

MAX_JOB_QUEUE_LOG_ROTATIONS
The *condor_schedd* daemon periodically rotates the job queue database file, in order to save disk space. This option controls how many rotated files are saved. It defaults to 1, which means there may be up to two history files (the previous one, which was rotated out of use, and the current one that is being written to). When the job queue file is rotated, and this rotation would cause the number of backups to be larger the the maximum specified, the oldest file is removed.

CLASSAD_LOG_STRICT_PARSING
A boolean value that defaults to `True`. When `True`, ClassAd log files will be read using a strict syntax checking for ClassAd expressions. ClassAd log files include the job queue log and the accountant log. When `False`, ClassAd log files are read without strict expression syntax checking, which allows some legacy ClassAd log data to be read in a backward compatible manner. This configuration variable may no longer be supported in future releases, eventually requiring all ClassAd log files to pass strict ClassAd syntax checking.

DEFAULT_DOMAIN_NAME
The value to be appended to a machine's host name, representing a domain name, which HTCondor then uses to form a fully qualified host name. This is required if there is no fully qualified host name in file `/etc/hosts` or in NIS. Set the value in the global configuration file, as HTCondor may depend on knowing this value in order to locate the local configuration file(s). The default value as given in the sample configuration file of the HTCondor download is bogus, and must be changed. If this variable is removed from the global configuration file, or if the definition is empty, then HTCondor attempts to discover the value.

NO_DNS
A boolean value that defaults to `False`. When `True`, HTCondor constructs host names using the host's IP address together with the value defined for `DEFAULT_DOMAIN_NAME`.

CM_IP_ADDR
If neither `COLLECTOR_HOST` nor `COLLECTOR_IP_ADDR` macros are defined, then this macro will be used to determine the IP address of the central manager (collector daemon). This macro is defined by an IP address.

EMAIL_DOMAIN
By default, if a user does not specify `notify_user` in the submit description file, any email HTCondor sends about that job will go to "username@UID_DOMAIN". If your machines all share a common UID domain (so that you would set `UID_DOMAIN` to be the same across all machines in your pool), but email to user@UID_DOMAIN is not the right place for HTCondor to send email for your site, you can define the default domain to use for email. A common example would be to set `EMAIL_DOMAIN` to the fully qualified host name of each machine in your pool, so users submitting jobs from a specific machine would get email sent to user@machine.your.domain, instead of user@your.domain. You would do this by setting `EMAIL_DOMAIN` to `$(FULL_HOSTNAME)`. In general, you should leave this setting commented out unless two things are true: 1) `UID_DOMAIN` is set to your domain, not `$(FULL_HOSTNAME)`, and 2) email to user@UID_DOMAIN will not work.

CREATE_CORE_FILES
Defines whether or not HTCondor daemons are to create a core file in the `LOG` directory if something really bad happens. It is used to set the resource limit for the size of a core file. If not defined, it leaves in place whatever limit was in effect when the HTCondor daemons (normally the *condor_master*) were started. This allows HTCondor to inherit the default system core file generation behavior at start up. For Unix operating systems, this behavior can be inherited from the parent shell, or specified in a shell script that starts HTCondor. If this parameter is set and `True`, the limit is increased to the maximum. If it is set to `False`, the limit is set at 0 (which means that no core files are created). Core files greatly help the HTCondor developers debug any problems you might be having. By using the parameter, you do not have to worry about tracking down where in your boot scripts you need to set the core limit before starting HTCondor. You set the parameter to whatever behavior you want HTCondor to enforce. This parameter defaults to undefined to allow the initial operating system default value to take precedence, and is commented out in the default configuration file.

ABORT_ON_EXCEPTION
When HTCondor programs detect a fatal internal exception, they normally log an error message and exit. If you have turned on `CREATE_CORE_FILES` , in some cases you may also want to turn on `ABORT_ON_EXCEPTION` so that core files are generated when an exception occurs. Set the following to True if that is what you want.

Q_QUERY_TIMEOUT
Defines the timeout (in seconds) that *condor_q* uses when trying to connect to the *condor_schedd*. Defaults to 20 seconds.

DEAD_COLLECTOR_MAX_AVOIDANCE_TIME
Defines the interval of time (in seconds) between checks for a failed primary *condor_collector* daemon. If

connections to the dead primary *condor_collector* take very little time to fail, new attempts to query the primary *condor_collector* may be more frequent than the specified maximum avoidance time. The default value equals one hour. This variable has relevance to flocked jobs, as it defines the maximum time they may be reporting to the primary *condor_collector* without the *condor_negotiator* noticing.

PASSWD_CACHE_REFRESH
HTCondor can cause NIS servers to become overwhelmed by queries for uid and group information in large pools. In order to avoid this problem, HTCondor caches UID and group information internally. This integer value allows pool administrators to specify (in seconds) how long HTCondor should wait until refreshes a cache entry. The default is set to 72000 seconds, or 20 hours, plus a random number of seconds between 0 and 60 to avoid having lots of processes refreshing at the same time. This means that if a pool administrator updates the user or group database (for example, `/etc/passwd` or `/etc/group`), it can take up to 6 minutes before HTCondor will have the updated information. This caching feature can be disabled by setting the refresh interval to 0. In addition, the cache can also be flushed explicitly by running the command *condor_reconfig*. This configuration variable has no effect on Windows.

SYSAPI_GET_LOADAVG
If set to False, then HTCondor will not attempt to compute the load average on the system, and instead will always report the system load average to be 0.0. Defaults to True.

NETWORK_MAX_PENDING_CONNECTS
This specifies a limit to the maximum number of simultaneous network connection attempts. This is primarily relevant to *condor_schedd*, which may try to connect to large numbers of startds when claiming them. The negotiator may also connect to large numbers of startds when initiating security sessions used for sending MATCH messages. On Unix, the default for this parameter is eighty percent of the process file descriptor limit. On windows, the default is 1600.

WANT_UDP_COMMAND_SOCKET
This setting, added in version 6.9.5, controls if HTCondor daemons should create a UDP command socket in addition to the TCP command socket (which is required). The default is `True`, and modifying it requires restarting all HTCondor daemons, not just a *condor_reconfig* or SIGHUP.

Normally, updates sent to the *condor_collector* use UDP, in addition to certain keep alive messages and other non-essential communication. However, in certain situations, it might be desirable to disable the UDP command port.

Unfortunately, due to a limitation in how these command sockets are created, it is not possible to define this setting on a per-daemon basis, for example, by trying to set `STARTD.WANT_UDP_COMMAND_SOCKET`. At least for now, this setting must be defined machine wide to function correctly.

If this setting is set to true on a machine running a *condor_collector*, the pool should be configured to use TCP updates to that collector (see *Using TCP to Send Updates to the condor_collector* for more information).

ALLOW_SCRIPTS_TO_RUN_AS_EXECUTABLES
A boolean value that, when `True`, permits scripts on Windows platforms to be used in place of the **executable** in a job submit description file, in place of a *condor_dagman* pre or post script, or in producing the configuration, for example. Allows a script to be used in any circumstance previously limited to a Windows executable or a batch file. The default value is `True`. See *Using Windows Scripts as Job Executables* for further description.

OPEN_VERB_FOR_<EXT>_FILES
A string that defines a Windows verb for use in a root hive registry look up. <EXT> defines the file name extension, which represents a scripting language, also needed for the look up. See *Using Windows Scripts as Job Executables* for a more complete description.

ENABLE_CLASSAD_CACHING
A boolean value that controls the caching of ClassAds. Caching saves memory when an HTCondor process contains many ClassAds with the same expressions. The default value is `True` for all daemons other than the *condor_shadow*, *condor_starter*, and *condor_master*. A value of `True` enables caching.

STRICT_CLASSAD_EVALUATION
A boolean value that controls how ClassAd expressions are evaluated. If set to `True`, then New ClassAd evaluation semantics are used. This means that attribute references without a `MY.` or `TARGET.` prefix are only looked up in the local ClassAd. If set to the default value of `False`, Old ClassAd evaluation semantics are used. See *ClassAds: Old and New* for details.

CLASSAD_USER_LIBS
A comma separated list of paths to shared libraries that contain additional ClassAd functions to be used during ClassAd evaluation.

CLASSAD_USER_PYTHON_MODULES
A comma separated list of python modules to load, which are to be used during ClassAd evaluation. If module `foo` is in this list, then function `bar` can be invoked in ClassAds via the expression `python_invoke("foo", "bar", ...)`. Any further arguments are converted from ClassAd expressions to python; the function return value is converted back to ClassAds. The python modules are loaded at configuration time, so any module-level statements are executed. Module writers can invoke `classad.register` at the module-level in order to use python functions directly.

Functions executed by ClassAds should be non-blocking and have no side-effects; otherwise, unpredictable HTCondor behavior may occur.

CLASSAD_USER_PYTHON_LIB
Specifies the path to the python libraries, which is needed when `CLASSAD_USER_PYTHON_MODULES` is set. Defaults to `$(LIBEXEC)/libclassad_python_user.so`, and would rarely be changed from the default value.

CONDOR_FSYNC
A boolean value that controls whether HTCondor calls fsync() when writing the user job and transaction logs. Setting this value to `False` will disable calls to fsync(), which can help performance for *condor_schedd* log writes at the cost of some durability of the log contents, should there be a power or hardware failure. The default value is `True`.

STATISTICS_TO_PUBLISH
A comma and/or space separated list that identifies which statistics collections are to place attributes in ClassAds. Additional information specifies a level of verbosity and other identification of which attributes to include and which to omit from ClassAds. The special value `NONE` disables all publishing, so no statistics will be published; no option is included. For other list items that define this variable, the syntax defines the two aspects by separating them with a colon. The first aspect defines a collection, which may specify which daemon is to publish the statistics, and the second aspect qualifies and refines the details of which attributes to publish for the collection, including a verbosity level. If the first aspect is `ALL`, the option is applied to all collections. If the first aspect is `DEFAULT`, the option is applied to all collections, with the intent that further list items will specify publishing that is to be different than the default. This first aspect may be `SCHEDD` or `SCHEDULER` to publish Statistics attributes in the ClassAd of the *condor_schedd*. It may be `TRANSFER` to publish file transfer statistics. It may be `STARTER` to publish Statistics attributes in the ClassAd of the *condor_starter*. Or, it may be `DC` or `DAEMONCORE` to publish DaemonCore statistics. One or more options are specified after the colon.

Option	Description
0	turns off the publishing of any statistics attributes
1	the default level, where some statistics attributes are and others are omitted
2	the verbose level, where all statistics attributes are published
3	the super verbose level, which is currently unused, but intended to be all statistics attributes published at the verbose level plus extra information
R	include attributes from the most recent time interval; the default
!R	omit attributes from the most recent time interval
D	include attributes for debugging
!D	omit attributes for debugging; the default
Z	include attributes even if the attribute's value is 0
!Z	omit attributes when the attribute's value is 0
L	include attributes that represent the lifetime value; the default
!L	omit attributes that represent the lifetime value

If this variable is not defined, then the default for each collection is used. If this variable is defined, and the definition does not specify each possible collection, then no statistics are published for those collections not defined. If an option specifies conflicting possibilities, such as `R!R`, then the last one takes precedence and is applied.

As an example, to cause a verbose setting of the publication of Statistics attributes only for the *condor_schedd*, and do not publish any other Statistics attributes:

```
STATISTICS_TO_PUBLISH = SCHEDD:2
```

As a second example, to cause all collections other than those for `DAEMONCORE` to publish at a verbosity setting of `1`, and omit lifetime values, where the `DAEMONCORE` includes all statistics at the verbose level:

```
STATISTICS_TO_PUBLISH = DEFAULT:1!L, DC:2RDZL
```

STATISTICS_TO_PUBLISH_LIST
A comma and/or space separated list of statistics attribute names that should be published in updates to the *condor_collector* daemon, even though the verbosity specified in `STATISTICS_TO_PUBLISH` would not normally send them. This setting has the effect of redefining the verbosity level of the statistics attributes that it mentions, so that they will always match the current statistics publication level as specified in `STATISTICS_TO_PUBLISH`.

STATISTICS_WINDOW_SECONDS
An integer value that controls the time window size, in seconds, for collecting windowed daemon statistics. These statistics are, by convention, those attributes with names that are of the form `Recent<attrname>`. Any data contributing to a windowed statistic that is older than this number of seconds is dropped from the statistic. For example, if `STATISTICS_WINDOW_SECONDS = 300`, then any jobs submitted more than 300 seconds ago are not counted in the windowed statistic `RecentJobsSubmitted`. Defaults to 1200 seconds, which is 20 minutes.

The window is broken into smaller time pieces called quantum. The window advances one quantum at a time.

STATISTICS_WINDOW_SECONDS_<collection>
The same as `STATISTICS_WINDOW_SECONDS`, but used to override the global setting for a particular statistic collection. Collection names currently implemented are `DC` or `DAEMONCORE` and `SCHEDD` or `SCHEDULER`.

STATISTICS_WINDOW_QUANTUM
For experts only, an integer value that controls the time quantization that form a time window, in seconds, for the data structures that maintain windowed statistics. Defaults to 240 seconds, which is 6 minutes. This default is purposely set to be slightly smaller than the update rate to the *condor_collector*. Setting a smaller value than the default increases the memory requirement for the statistics. Graphing of statistics at the level of the quantum expects to see counts that appear like a saw tooth.

STATISTICS_WINDOW_QUANTUM_<collection>
The same as STATISTICS_WINDOW_QUANTUM, but used to override the global setting for a particular statistic collection. Collection names currently implemented are DC or DAEMONCORE and SCHEDD or SCHEDULER.

TCP_KEEPALIVE_INTERVAL
The number of seconds specifying a keep alive interval to use for any HTCondor TCP connection. The default keep alive interval is 360 (6 minutes); this value is chosen to minimize the likelihood that keep alive packets are sent, while still detecting dead TCP connections before job leases expire. A smaller value will consume more operating system and network resources, while a larger value may cause jobs to fail unnecessarily due to network disconnects. Most users will not need to tune this configuration variable. A value of 0 will use the operating system default, and a value of -1 will disable HTCondor's use of a TCP keep alive.

ENABLE_IPV4
A boolean with the additional special value of auto. If true, HTCondor will use IPv4 if available, and fail otherwise. If false, HTCondor will not use IPv4. If auto, which is the default, HTCondor will use IPv4 if it can find an interface with an IPv4 address, and that address is (a) public or private, or (b) no interface's IPv6 address is public or private. If HTCondor finds more than one address of each protocol, only the most public address is considered for that protocol.

ENABLE_IPV6
A boolean with the additional special value of auto. If true, HTCondor will use IPv6 if available, and fail otherwise. If false, HTCondor will not use IPv6. If auto, which is the default, HTCondor will use IPv6 if it can find an interface with an IPv6 address, and that address is (a) public or private, or (b) no interface's IPv4 address is public or private. If HTCondor finds more than one address of each protocol, only the most public address is considered for that protocol.

PREFER_IPV4
A boolean which will cause HTCondor to prefer IPv4 when it is able to choose. HTCondor will otherwise prefer IPv6. The default is True.

ADVERTISE_IPV4_FIRST
A string (treated as a boolean). If ADVERTISE_IPV4_FIRST evaluates to True, HTCondor will advertise its IPv4 addresses before its IPv6 addresses; otherwise the IPv6 addresses will come first. Defaults to $(PREFER_IPV4).

IGNORE_TARGET_PROTOCOL_PREFERENCE
A string (treated as a boolean). If IGNORE_TARGET_PROTOCOL_PREFERENCE evaluates to True, the target's listed protocol preferences will be ignored; otherwise they will not. Defaults to $(PREFER_IPV4).

IGNORE_DNS_PROTOCOL_PREFERENCE
A string (treated as a boolean). IGNORE_DNS_PROTOCOL_PREFERENCE evaluates to True, the protocol order returned by the DNS will be ignored; otherwise it will not. Defaults to $(PREFER_IPV4).

PREFER_OUTBOUND_IPV4
A string (treated as a boolean). PREFER_OUTBOUND_IPV4 evaluates to True, HTCondor will prefer IPv4; otherwise it will not. Defaults to $(PREFER_IPV4).

<SUBSYS>_CLASSAD_USER_MAP_NAMES
A string defining a list of names for username-to-accounting group mappings for the specified daemon. Names must be separated by spaces or commas.

List of possible subsystems to set <SUBSYS> can be found at .

CLASSAD_USER_MAPFILE_<name>
A string giving the name of a file to parse to initialize the map for the given username. Note that this macro is only used if <SUBSYS>_CLASSAD_USER_MAP_NAMES is defined for the relevant daemon.

The format for the map file is the same as the format for CLASSAD_USER_MAPDATA_<name>, below.

CLASSAD_USER_MAPDATA_<name>
A string containing data to be used to initialize the map for the given username. Note that this

macro is only used if <SUBSYS>_CLASSAD_USER_MAP_NAMES is defined for the relevant daemon, and CLASSAD_USER_MAPFILE_<name> is not defined for the given name.

The format for the map data is the same as the format for the security unified map file (see *The Unified Map File for Authentication* for details).

The first field must be * (or a subset name - see below), the second field is a regex that we will match against the input, and the third field will be the output if the regex matches, the 3 and 4 argument form of the ClassAd userMap() function (see *ClassAd Syntax*) expect that the third field will be a comma separated list of values. For example:

```
# file: groups.mapdata
* John  chemistry,physics,glassblowing
* Juan  physics,chemistry
* Bob   security
* Alice security,math
```

Here is simple example showing how to configure CLASSAD_USER_MAPDATA_<name> for testing and experimentation.

```
# configuration statements to create a simple userMap that
# can be used by the Schedd as well as by tools like condor_q
#
SCHEDD_CLASSAD_USER_MAP_NAMES = Trust $(SCHEDD_CLASSAD_USER_MAP_NAMES)
TOOL_CLASSAD_USER_MAP_NAMES = Trust $(TOOL_CLASSAD_USER_MAP_NAMES)
CLASSAD_USER_MAPDATA_Trust @=end
  * Bob   User
  * Alice Admin
  * /.*/  Nobody
@end
#
# test with
#   condor_q -af:j 'Owner' 'userMap("Trust",Owner)'
```

Optional submaps: If the first field of the mapfile contains something other than *, then a submap is defined. To select a submap for lookup, the first argument for userMap() should be "mapname.submap". For example:

```
# mapdata 'groups' with submaps
*   Bob   security
*   Alice security,math
alt Alice math,hacking
```

SIGN_S3_URLS
A boolean value that, when True, tells HTCondor to convert s3:// URLs into pre-signed https:// URLs. This allows execute nodes to download from or upload to secure S3 buckets without access to the user's API tokens, which remain on the submit node at all times. This value defaults to TRUE but can be disabled if the administrator has already provided an s3:// plug-in. This value must be set on both the submit node and on the execute node.

5.5.2 Daemon Logging Configuration File Entries

These entries control how and where the HTCondor daemons write to log files. Many of the entries in this section represents multiple macros. There is one for each subsystem (listed in). The macro name for each substitutes <SUBSYS> with the name of the subsystem corresponding to the daemon.

<SUBSYS>_LOG
Defines the path and file name of the log file for a given subsystem. For example, `$(STARTD_LOG)` gives the location of the log file for the *condor_startd* daemon. The default value for most daemons is the daemon's name in camel case, concatenated with `Log`. For example, the default log defined for the *condor_master* daemon is `$(LOG)/MasterLog`. The default value for other subsystems is `$(LOG)/<SUBSYS>LOG`. The special value `SYSLOG` causes the daemon to log via the syslog facility on Linux. If the log file cannot be written to, then the daemon will attempt to log this into a new file of the name `$(LOG)/dprintf_failure.<SUBSYS>` before the daemon exits.

List of possible subsystems to set <SUBSYS> can be found at .

LOG_TO_SYSLOG
A boolean value that is `False` by default. When `True`, all daemon logs are routed to the syslog facility on Linux.

MAX_<SUBSYS>_LOG
Controls the maximum size in bytes or amount of time that a log will be allowed to grow. For any log not specified, the default is `$(MAX_DEFAULT_LOG)` , which currently defaults to 10 MiB in size. Values are specified with the same syntax as `MAX_DEFAULT_LOG` .

Note that a log file for the *condor_procd* does not use this configuration variable definition. Its implementation is separate. See .

List of possible subsystems to set <SUBSYS> can be found at .

MAX_DEFAULT_LOG
Controls the maximum size in bytes or amount of time that any log not explicitly specified using `MAX_<SUBSYS>_LOG` will be allowed to grow. When it is time to rotate a log file, it will be saved to a file with an ISO timestamp suffix. The oldest rotated file receives the ending `.old`. The `.old` files are overwritten each time the maximum number of rotated files (determined by the value of `MAX_NUM_<SUBSYS>_LOG`) is exceeded. The default value is 10 MiB in size. A value of 0 specifies that the file may grow without bounds. A single integer value is specified; without a suffix, it defaults to specifying a size in bytes. A suffix is case insensitive, except for `Mb` and `Min`; these both start with the same letter, and the implementation attaches meaning to the letter case when only the first letter is present. Therefore, use the following suffixes to qualify the integer: `Bytes` for bytes `Kb` for KiB, 2^{10} numbers of bytes `Mb` for MiB, 2^{20} numbers of bytes `Gb` for GiB, 2^{30} numbers of bytes `Tb` for TiB, 2^{40} numbers of bytes `Sec` for seconds `Min` for minutes `Hr` for hours `Day` for days `Wk` for weeks

MAX_NUM_<SUBSYS>_LOG
An integer that controls the maximum number of rotations a log file is allowed to perform before the oldest one will be rotated away. Thus, at most `MAX_NUM_<SUBSYS>_LOG + 1` log files of the same program coexist at a given time. The default value is 1.

List of possible subsystems to set <SUBSYS> can be found at .

TRUNC_<SUBSYS>_LOG_ON_OPEN
If this macro is defined and set to `True`, the affected log will be truncated and started from an empty file with each invocation of the program. Otherwise, new invocations of the program will append to the previous log file. By default this setting is `False` for all daemons.

List of possible subsystems to set <SUBSYS> can be found at .

<SUBSYS>_LOG_KEEP_OPEN
A boolean value that controls whether or not the log file is kept open between writes. When `True`, the daemon

will not open and close the log file between writes. Instead the daemon will hold the log file open until the log needs to be rotated. When `False`, the daemon reverts to the previous behavior of opening and closing the log file between writes. When the `$(<SUBSYS>_LOCK)` macro is defined, setting `$(<SUBSYS>_LOG_KEEP_OPEN)` has no effect, as the daemon will unconditionally revert back to the open/close between writes behavior. On Windows platforms, the value defaults to `True` for all daemons. On Linux platforms, the value defaults to `True` for all daemons, except the *condor_shadow*, due to a global file descriptor limit.

List of possible subsystems to set <SUBSYS> can be found at .

<SUBSYS>_LOCK
This macro specifies the lock file used to synchronize append operations to the log file for this subsystem. It must be a separate file from the `$(<SUBSYS>_LOG)` file, since the `$(<SUBSYS>_LOG)` file may be rotated and you want to be able to synchronize access across log file rotations. A lock file is only required for log files which are accessed by more than one process. Currently, this includes only the `SHADOW` subsystem. This macro is defined relative to the `$(LOCK)` macro.

List of possible subsystems to set <SUBSYS> can be found at .

JOB_QUEUE_LOG
A full path and file name, specifying the job queue log. The default value, when not defined is `$(SPOOL)`/job_queue.log. This specification can be useful, if there is a solid state drive which is big enough to hold the frequently written to `job_queue.log`, but not big enough to hold the whole contents of the spool directory.

FILE_LOCK_VIA_MUTEX
This macro setting only works on Win32 - it is ignored on Unix. If set to be `True`, then log locking is implemented via a kernel mutex instead of via file locking. On Win32, mutex access is FIFO, while obtaining a file lock is non-deterministic. Thus setting to `True` fixes problems on Win32 where processes (usually shadows) could starve waiting for a lock on a log file. Defaults to `True` on Win32, and is always `False` on Unix.

LOCK_DEBUG_LOG_TO_APPEND
A boolean value that defaults to `False`. This variable controls whether a daemon's debug lock is used when appending to the log. When `False`, the debug lock is only used when rotating the log file. This is more efficient, especially when many processes share the same log file. When `True`, the debug lock is used when writing to the log, as well as when rotating the log file. This setting is ignored under Windows, and the behavior of Windows platforms is as though this variable were `True`. Under Unix, the default value of `False` is appropriate when logging to file systems that support the POSIX semantics of `O_APPEND`. On non-POSIX-compliant file systems, it is possible for the characters in log messages from multiple processes sharing the same log to be interleaved, unless locking is used. Since HTCondor does not support sharing of debug logs between processes running on different machines, many non-POSIX-compliant file systems will still avoid interleaved messages without requiring HTCondor to use a lock. Tests of AFS and NFS have not revealed any problems when appending to the log without locking.

ENABLE_USERLOG_LOCKING
A boolean value that defaults to `False` on Unix platforms and `True` on Windows platforms. When `True`, a user's job event log will be locked before being written to. If `False`, HTCondor will not lock the file before writing.

ENABLE_USERLOG_FSYNC
A boolean value that is `True` by default. When `True`, writes to the user's job event log are sync-ed to disk before releasing the lock.

USERLOG_FILE_CACHE_MAX
The integer number of job event log files that the *condor_schedd* will keep open for writing during an interval of time (specified by `USERLOG_FILE_CACHE_CLEAR_INTERVAL`). The default value is 0, causing no files to remain open; when 0, each job event log is opened, the event is written, and then the file is closed. Individual file descriptors are removed from this count when the *condor_schedd* detects that no jobs are currently using them. Opening a file is a relatively time consuming operation on a networked file system (NFS), and therefore, allowing a set of files to remain open can improve performance. The value of this variable needs to be set low enough

such that the *condor_schedd* daemon process does not run out of file descriptors by leaving these job event log files open. The Linux operating system defaults to permitting 1024 assigned file descriptors per process; the *condor_schedd* will have one file descriptor per running job for the *condor_shadow*.

USERLOG_FILE_CACHE_CLEAR_INTERVAL
The integer number of seconds that forms the time interval within which job event logs will be permitted to remain open when USERLOG_FILE_CACHE_MAX is greater than zero. The default is 60 seconds. When the interval has passed, all job event logs that the *condor_schedd* has permitted to stay open will be closed, and the interval within which job event logs may remain open between writes of events begins anew. This time interval may be set to a longer duration if the administrator determines that the *condor_schedd* will not exceed the maximum number of file descriptors; a longer interval may yield higher performance due to fewer files being opened and closed.

CREATE_LOCKS_ON_LOCAL_DISK
A boolean value utilized only for Unix operating systems, that defaults to True. This variable is only relevant if ENABLE_USERLOG_LOCKING is True. When True, lock files are written to a directory named condorLocks, thereby using a local drive to avoid known problems with locking on NFS. The location of the condorLocks directory is determined by

1. The value of TEMP_DIR, if defined.
2. The value of TMP_DIR, if defined and TEMP_DIR is not defined.
3. The default value of /tmp, if neither TEMP_DIR nor TMP_DIR is defined.

TOUCH_LOG_INTERVAL
The time interval in seconds between when daemons touch their log files. The change in last modification time for the log file is useful when a daemon restarts after failure or shut down. The last modification date is printed, and it provides an upper bound on the length of time that the daemon was not running. Defaults to 60 seconds.

LOGS_USE_TIMESTAMP
This macro controls how the current time is formatted at the start of each line in the daemon log files. When True, the Unix time is printed (number of seconds since 00:00:00 UTC, January 1, 1970). When False (the default value), the time is printed like so: <Month>/<Day> <Hour>:<Minute>:<Second> in the local timezone.

DEBUG_TIME_FORMAT
This string defines how to format the current time printed at the start of each line in the daemon log files. The value is a format string is passed to the C strftime() function, so see that manual page for platform-specific details. If not defined, the default value is

```
"%m/%d/%y %H:%M:%S"
```

<SUBSYS>_DEBUG
All of the HTCondor daemons can produce different levels of output depending on how much information is desired. The various levels of verbosity for a given daemon are determined by this macro. Settings are a comma, vertical bar, or space-separated list of categories and options. Each category can be followed by a colon and a single digit indicating the verbosity for that category :1 is assumed if there is no verbosity modifier. Permitted verbosity values are :1 for normal, :2 for extra messages, and :0 to disable logging of that category of messages. The primary daemon log will always include category and verbosity D_ALWAYS:1, unless D_ALWAYS:0 is added to this list. Category and option names are:

D_ANY
This flag turns on all cagetories of messages Be warned: this will generate about a HUGE amount of output. To obtain a higher level of output than the default, consider using D_FULLDEBUG before using this option.

D_ALL
This is equivalent to D_ANY D_PID D_FDS D_CAT Be warned: this will generate about a HUGE amount of output. To obtain a higher level of output than the default, consider using D_FULLDEBUG before using this option.

D_FAILURE
: This category is used for messages that indicate the daemon is unable to continue running. These message are "always" printed unless `D_FAILURE:0` is added to the list

D_STATUS
: This category is used for messages that indicate what task the daemon is currently doing or progress. Messages of this category will be always printed unless `D_STATUS:0` is added to the list

D_ALWAYS
: This category is used for messages that are "always" printed unless `D_ALWAYS:0` is configured. These can be progress or status message, as well as failures that do not prevent the daemon from continuing to operate such as a failure to start a job. At verbosity 2 this category is equivalent to `D_FULLDEBUG` below.

D_FULLDEBUG
: This level provides verbose output of a general nature into the log files. Frequent log messages for very specific debugging purposes would be excluded. In those cases, the messages would be viewed by having that other flag and `D_FULLDEBUG` both listed in the configuration file. This is equivalent to `D_ALWAYS:2`

D_DAEMONCORE
: Provides log file entries specific to DaemonCore, such as timers the daemons have set and the commands that are registered. If `D_DAEMONCORE:2` is set, expect very verbose output.

D_PRIV
: This flag provides log messages about the privilege state switching that the daemons do. See *User Accounts in HTCondor on Unix Platforms* on UIDs in HTCondor for details.

D_COMMAND
: With this flag set, any daemon that uses DaemonCore will print out a log message whenever a command comes in. The name and integer of the command, whether the command was sent via UDP or TCP, and where the command was sent from are all logged. Because the messages about the command used by *condor_kbdd* to communicate with the *condor_startd* whenever there is activity on the X server, and the command used for keep-alives are both only printed with `D_FULLDEBUG` enabled, it is best if this setting is used for all daemons.

D_LOAD
: The *condor_startd* keeps track of the load average on the machine where it is running. Both the general system load average, and the load average being generated by HTCondor's activity there are determined. With this flag set, the *condor_startd* will log a message with the current state of both of these load averages whenever it computes them. This flag only affects the *condor_startd*.

D_KEYBOARD
: With this flag set, the *condor_startd* will print out a log message with the current values for remote and local keyboard idle time. This flag affects only the *condor_startd*.

D_JOB
: When this flag is set, the *condor_startd* will send to its log file the contents of any job ClassAd that the *condor_schedd* sends to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

D_MACHINE
: When this flag is set, the *condor_startd* will send to its log file the contents of its resource ClassAd when the *condor_schedd* tries to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

D_SYSCALLS
: This flag is used to make the *condor_shadow* log remote syscall requests and return values. This can help track down problems a user is having with a particular job by providing the system calls the job is performing. If any are failing, the reason for the failure is given. The *condor_schedd* also uses this flag for the server portion of the queue management code. With `D_SYSCALLS` defined in `SCHEDD_DEBUG` there will be verbose logging of all queue management operations the *condor_schedd* performs.

D_MATCH
When this flag is set, the *condor_negotiator* logs a message for every match.

D_NETWORK
When this flag is set, all HTCondor daemons will log a message on every TCP accept, connect, and close, and on every UDP send and receive. This flag is not yet fully supported in the *condor_shadow*.

D_HOSTNAME
When this flag is set, the HTCondor daemons and/or tools will print verbose messages explaining how they resolve host names, domain names, and IP addresses. This is useful for sites that are having trouble getting HTCondor to work because of problems with DNS, NIS or other host name resolving systems in use.

D_SECURITY
This flag will enable debug messages pertaining to the setup of secure network communication, including messages for the negotiation of a socket authentication mechanism, the management of a session key cache. and messages about the authentication process itself. See *HTCondor's Security Model* for more information about secure communication configuration. `D_SECURITY:2` logging is highly verbose and should be used only when actively debugging security configuration problems.

D_PROCFAMILY
HTCondor often times needs to manage an entire family of processes, (that is, a process and all descendants of that process). This debug flag will turn on debugging output for the management of families of processes.

D_ACCOUNTANT
When this flag is set, the *condor_negotiator* will output debug messages relating to the computation of user priorities (see *User Priorities and Negotiation*).

D_PROTOCOL
Enable debug messages relating to the protocol for HTCondor's matchmaking and resource claiming framework.

D_STATS
Enable debug messages relating to the TCP statistics for file transfers. Note that the shadow and starter, by default, log these statistics to special log files (see and . Note that, as of version 8.5.6, `C_GAHP_DEBUG` defaults to `D_STATS`.

D_PID
This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If `D_PID` is set, HTCondor will always print out the process identifier (PID) of the process writing each line to the log file. This is especially helpful for HTCondor daemons that can fork multiple helper-processes (such as the *condor_schedd* or *condor_collector*) so the log file will clearly show which thread of execution is generating each log message.

D_FDS
This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If `D_FDS` is set, HTCondor will always print out the file descriptor that the open of the log file was allocated by the operating system. This can be helpful in debugging HTCondor's use of system file descriptors as it will generally track the number of file descriptors that HTCondor has open.

D_CAT or D_CATEGORY
This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If `D_CAT` or `D_CATEGORY` is set, Condor will include the debugging level flags that were in effect for each line of output. This may be used to filter log output by the level or tag it, for example, identifying all logging output at level `D_SECURITY`, or `D_ACCOUNTANT`.

D_TIMESTAMP

This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If D_TIMESTAMP is set, the time at the beginning of each line in the log file with be a number of seconds since the start of the Unix era. This form of timestamp can be more convenient for tools to process.

D_SUB_SECOND
This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If D_SUB_SECOND is set, the time at the beginning of each line in the log file will contain a fractional part to the seconds field that is accurate to the millisecond.

List of possible subsystems to set <SUBSYS> can be found at .

ALL_DEBUG
Used to make all subsystems share a debug flag. Set the parameter ALL_DEBUG instead of changing all of the individual parameters. For example, to turn on all debugging in all subsystems, set ALL_DEBUG = D_ALL.

TOOL_DEBUG
Uses the same values (debugging levels) as <SUBSYS>_DEBUG to describe the amount of debugging information sent to `stderr` for HTCondor tools.

Log files may optionally be specified per debug level as follows:

<SUBSYS>_<LEVEL>_LOG
The name of a log file for messages at a specific debug level for a specific subsystem. <LEVEL> is defined by any debug level, but without the D_ prefix. See for the list of debug levels. If the debug level is included in $(<SUBSYS>_DEBUG), then all messages of this debug level will be written both to the log file defined by <SUBSYS>_LOG and the the log file defined by <SUBSYS>_<LEVEL>_LOG. As examples, SHADOW_SYSCALLS_LOG specifies a log file for all remote system call debug messages, and NEGOTIATOR_MATCH_LOG specifies a log file that only captures *condor_negotiator* debug events occurring with matches.

List of possible subsystems to set <SUBSYS> can be found at .

MAX_<SUBSYS>_<LEVEL>_LOG
See .

TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN
See .

The following macros control where and what is written to the event log, a file that receives job events, but across all users and user's jobs.

EVENT_LOG
The full path and file name of the event log. There is no default value for this variable, so no event log will be written, if not defined.

EVENT_LOG_MAX_SIZE
Controls the maximum length in bytes to which the event log will be allowed to grow. The log file will grow to the specified length, then be saved to a file with the suffix .old. The .old files are overwritten each time the log is saved. A value of 0 specifies that the file may grow without bounds (and disables rotation). The default is 1 MiB. For backwards compatibility, MAX_EVENT_LOG will be used if EVENT_LOG_MAX_SIZE is not defined. If EVENT_LOG is not defined, this parameter has no effect.

MAX_EVENT_LOG
See .

EVENT_LOG_MAX_ROTATIONS
Controls the maximum number of rotations of the event log that will be stored. If this value is 1 (the default), the event log will be rotated to a ".old" file as described above. However, if this is greater than 1, then multiple rotation files will be stores, up to EVENT_LOG_MAX_ROTATIONS of them. These files will be named, instead of the ".old" suffix, ".1", ".2", with the ".1" being the most recent rotation. This is an integer parameter with a

default value of 1. If EVENT_LOG is not defined, or if EVENT_LOG_MAX_SIZE has a value of 0 (which disables event log rotation), this parameter has no effect.

EVENT_LOG_ROTATION_LOCK
Specifies the lock file that will be used to ensure that, when rotating files, the rotation is done by a single process. This is a string parameter; its default value is $(LOCK)/EventLogLock. If an empty value is set, then the file that is used is the file path of the event log itself, with the string .lock appended. If EVENT_LOG is not defined, or if EVENT_LOG_MAX_SIZE has a value of 0 (which disables event log rotation), this configuration variable has no effect.

EVENT_LOG_FSYNC
A boolean value that controls whether HTCondor will perform an fsync() after writing each event to the event log. When True, an fsync() operation is performed after each event. This fsync() operation forces the operating system to synchronize the updates to the event log to the disk, but can negatively affect the performance of the system. Defaults to False.

EVENT_LOG_LOCKING
A boolean value that defaults to False on Unix platforms and True on Windows platforms. When True, the event log (as specified by EVENT_LOG) will be locked before being written to. When False, HTCondor does not lock the file before writing.

EVENT_LOG_COUNT_EVENTS
A boolean value that is False by default. When True, upon rotation of the user's job event log, a count of the number of job events is taken by scanning the log, such that the newly created, post-rotation user job event log will have this count in its header. This configuration variable is relevant when rotation of the user's job event log is enabled.

EVENT_LOG_FORMAT_OPTIONS
A list of case-insensitive keywords that control formatting of the log events and of timestamps for the log specified by EVENT_LOG. Use zero or one of the following formatting options:

XML
Log events in XML format. This has the same effect EVENT_LOG_USE_XML below

JSON
Log events in JSON format. This conflicts with EVENT_LOG_USE_XML below

And zero or more of the following option flags:

UTC
Log event timestamps as Universal Coordinated Time. The time value will be printed with a timezone value of Z to indicate that times are UTC.

ISO_DATE
Log event timestamps in ISO 8601 format. This format includes a 4 digit year and is printed in a way that makes sorting by date easier.

SUB_SECOND
Include fractional seconds in event timestamps.

LEGACY
Set all time formatting flags to be compatible with older versions of HTCondor.

All of the above options are case-insensitive, and can be preceded by a ! to invert their meaning, so configuring !UTC, !ISO_DATE, !SUB_SECOND gives the same result as configuring LEGACY.

EVENT_LOG_USE_XML
A boolean value that defaults to False. When True, events are logged in XML format. If EVENT_LOG is not defined, this parameter has no effect.

EVENT_LOG_JOB_AD_INFORMATION_ATTRS
A comma separated list of job ClassAd attributes, whose evaluated values form a new event, the `JobAdInformationEvent`, given Event Number 028. This new event is placed in the event log in addition to each logged event. If `EVENT_LOG` is not defined, this configuration variable has no effect. This configuration variable is the same as the job ClassAd attribute `JobAdInformationAttrs` (see *Job ClassAd Attributes*), but it applies to the system Event Log rather than the user job log.

DEFAULT_USERLOG_FORMAT_OPTIONS
A list of case-insensitive keywords that control formatting of the events and of timestamps for the log specified by a job's `UserLog` or `DAGManNodesLog` attributes. see `EVENT_LOG_FORMAT_OPTIONS` above for the permitted options.

5.5.3 DaemonCore Configuration File Entries

Please read *DaemonCore* for details on DaemonCore. There are certain configuration file settings that DaemonCore uses which affect all HTCondor daemons.

ALLOW...
All macros that begin with either `ALLOW` or `DENY` are settings for HTCondor's security. See *Authorization* on Setting up security in HTCondor for details on these macros and how to configure them.

ENABLE_RUNTIME_CONFIG
The *condor_config_val* tool has an option **-rset** for dynamically setting run time configuration values, and which only affect the in-memory configuration variables. Because of the potential security implications of this feature, by default, HTCondor daemons will not honor these requests. To use this functionality, HTCondor administrators must specifically enable it by setting `ENABLE_RUNTIME_CONFIG` to `True`, and specify what configuration variables can be changed using the `SETTABLE_ATTRS...` family of configuration options. Defaults to `False`.

ENABLE_PERSISTENT_CONFIG
The *condor_config_val* tool has a **-set** option for dynamically setting persistent configuration values. These values override options in the normal HTCondor configuration files. Because of the potential security implications of this feature, by default, HTCondor daemons will not honor these requests. To use this functionality, HTCondor administrators must specifically enable it by setting `ENABLE_PERSISTENT_CONFIG` to `True`, creating a directory where the HTCondor daemons will hold these dynamically-generated persistent configuration files (declared using `PERSISTENT_CONFIG_DIR`, described below) and specify what configuration variables can be changed using the `SETTABLE_ATTRS...` family of configuration options. Defaults to `False`.

PERSISTENT_CONFIG_DIR
Directory where daemons should store dynamically-generated persistent configuration files (used to support *condor_config_val* **-set**) This directory should **only** be writable by root, or the user the HTCondor daemons are running as (if non-root). There is no default, administrators that wish to use this functionality must create this directory and define this setting. This directory must not be shared by multiple HTCondor installations, though it can be shared by all HTCondor daemons on the same host. Keep in mind that this directory should not be placed on an NFS mount where "root-squashing" is in effect, or else HTCondor daemons running as root will not be able to write to them. A directory (only writable by root) on the local file system is usually the best location for this directory.

SETTABLE_ATTRS_<PERMISSION-LEVEL>
All macros that begin with `SETTABLE_ATTRS` or `<SUBSYS>.SETTABLE_ATTRS` are settings used to restrict the configuration values that can be changed using the *condor_config_val* command. See *Authorization* on Setting up Security in HTCondor for details on these macros and how to configure them. In particular, *Authorization* contains details specific to these macros.

SHUTDOWN_GRACEFUL_TIMEOUT
Determines how long HTCondor will allow daemons try their graceful shutdown methods before they do a hard

shutdown. It is defined in terms of seconds. The default is 1800 (30 minutes).

<SUBSYS>_ADDRESS_FILE
A complete path to a file that is to contain an IP address and port number for a daemon. Every HTCondor daemon that uses DaemonCore has a command port where commands are sent. The IP/port of the daemon is put in that daemon's ClassAd, so that other machines in the pool can query the *condor_collector* (which listens on a well-known port) to find the address of a given daemon on a given machine. When tools and daemons are all executing on the same single machine, communications do not require a query of the *condor_collector* daemon. Instead, they look in a file on the local disk to find the IP/port. This macro causes daemons to write the IP/port of their command socket to a specified file. In this way, local tools will continue to operate, even if the machine running the *condor_collector* crashes. Using this file will also generate slightly less network traffic in the pool, since tools including *condor_q* and *condor_rm* do not need to send any messages over the network to locate the *condor_schedd* daemon. This macro is not necessary for the *condor_collector* daemon, since its command socket is at a well-known port.

List of possible subsystems to set <SUBSYS> can be found at .

<SUBSYS>_SUPER_ADDRESS_FILE
A complete path to a file that is to contain an IP address and port number for a command port that is serviced with priority for a daemon. Every HTCondor daemon that uses DaemonCore may have a higher priority command port where commands are sent. Any command that goes through *condor_sos*, and any command issued by the super user (root or local system) for a daemon on the local machine will have the command sent to this port. Default values are provided for the *condor_schedd* daemon at `$(SPOOL)/.schedd_address.super` and the *condor_collector* daemon at `$(LOG)/.collector_address.super`. When not defined for other DaemonCore daemons, there will be no higher priority command port.

List of possible subsystems to set <SUBSYS> can be found at .

<SUBSYS>_DAEMON_AD_FILE
A complete path to a file that is to contain the ClassAd for a daemon. When the daemon sends a ClassAd describing itself to the *condor_collector*, it will also place a copy of the ClassAd in this file. Currently, this setting only works for the *condor_schedd*.

List of possible subsystems to set <SUBSYS> can be found at .

<SUBSYS>_ATTRS
Allows any DaemonCore daemon to advertise arbitrary expressions from the configuration file in its ClassAd. Give the list of entries from the configuration file you want in the given daemon's ClassAd. Frequently used to add attributes to machines so that the machines can discriminate between other machines in a job's **rank** and **requirements**.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined by .

Note: The *condor_kbdd* does not send ClassAds now, so this entry does not affect it. The *condor_startd*, *condor_schedd*, *condor_master*, and *condor_collector* do send ClassAds, so those would be valid subsystems to set this entry for.

SUBMIT_ATTRS not part of the <SUBSYS>_ATTRS, it is documented in .

Because of the different syntax of the configuration file and ClassAds, a little extra work is required to get a given entry into a ClassAd. In particular, ClassAds require quote marks (") around strings. Numeric values and boolean expressions can go in directly. For example, if the *condor_startd* is to advertise a string macro, a numeric macro, and a boolean expression, do something similar to:

```
STRING = This is a string
NUMBER = 666
BOOL1 = True
```

(continues on next page)

(continued from previous page)

```
BOOL2 = time() >= $(NUMBER) || $(BOOL1)
MY_STRING = "$(STRING)"
STARTD_ATTRS = MY_STRING, NUMBER, BOOL1, BOOL2
```

List of possible subsystems to set <SUBSYS> can be found at .

DAEMON_SHUTDOWN
Starting with HTCondor version 6.9.3, whenever a daemon is about to publish a ClassAd update to the *condor_collector*, it will evaluate this expression. If it evaluates to `True`, the daemon will gracefully shut itself down, exit with the exit code 99, and will not be restarted by the *condor_master* (as if it sent itself a *condor_off* command). The expression is evaluated in the context of the ClassAd that is being sent to the *condor_collector*, so it can reference any attributes that can be seen with condor_status -long [-daemon_type] (for example, condor_status -long [-master] for the *condor_master*). Since each daemon's ClassAd will contain different attributes, administrators should define these shutdown expressions specific to each daemon, for example:

```
STARTD.DAEMON_SHUTDOWN = when to shutdown the startd
MASTER.DAEMON_SHUTDOWN = when to shutdown the master
```

Normally, these expressions would not be necessary, so if not defined, they default to FALSE.

Note: This functionality does not work in conjunction with HTCondor's high-availability support (see *The High Availability of Daemons* for more information). If you enable high-availability for a particular daemon, you should not define this expression.

DAEMON_SHUTDOWN_FAST
Identical to `DAEMON_SHUTDOWN` (defined above), except the daemon will use the fast shutdown mode (as if it sent itself a *condor_off* command using the **-fast** option).

USE_CLONE_TO_CREATE_PROCESSES
A boolean value that controls how an HTCondor daemon creates a new process on Linux platforms. If set to the default value of `True`, the `clone` system call is used. Otherwise, the `fork` system call is used. `clone` provides scalability improvements for daemons using a large amount of memory, for example, a *condor_schedd* with a lot of jobs in the queue. Currently, the use of `clone` is available on Linux systems. If HTCondor detects that it is running under the *valgrind* analysis tools, this setting is ignored and treated as `False`, to work around incompatibilities.

MAX_TIME_SKIP
When an HTCondor daemon notices the system clock skip forwards or backwards more than the number of seconds specified by this parameter, it may take special action. For instance, the *condor_master* will restart HTCondor in the event of a clock skip. Defaults to a value of 1200, which in effect means that HTCondor will restart if the system clock jumps by more than 20 minutes.

NOT_RESPONDING_TIMEOUT
When an HTCondor daemon's parent process is another HTCondor daemon, the child daemon will periodically send a short message to its parent stating that it is alive and well. If the parent does not hear from the child for a while, the parent assumes that the child is hung, kills the child, and restarts the child. This parameter controls how long the parent waits before killing the child. It is defined in terms of seconds and defaults to 3600 (1 hour). The child sends its alive and well messages at an interval of one third of this value.

<SUBSYS>_NOT_RESPONDING_TIMEOUT
Identical to `NOT_RESPONDING_TIMEOUT`, but controls the timeout for a specific type of daemon. For example, `SCHEDD_NOT_RESPONDING_TIMEOUT` controls how long the *condor_schedd* 's parent daemon will wait without receiving an alive and well message from the *condor_schedd* before killing it.

List of possible subsystems to set <SUBSYS> can be found at .

NOT_RESPONDING_WANT_CORE
A boolean value with a default value of `False`. This parameter is for debugging purposes on Unix systems, and it controls the behavior of the parent process when the parent process determines that a child process is not responding. If `NOT_RESPONDING_WANT_CORE` is `True`, the parent will send a SIGABRT instead of SIGKILL to the child process. If the child process is configured with the configuration variable `CREATE_CORE_FILES` enabled, the child process will then generate a core dump. See and for more details.

LOCK_FILE_UPDATE_INTERVAL
An integer value representing seconds, controlling how often valid lock files should have their on disk timestamps updated. Updating the timestamps prevents administrative programs, such as *tmpwatch*, from deleting long lived lock files. If set to a value less than 60, the update time will be 60 seconds. The default value is 28800, which is 8 hours. This variable only takes effect at the start or restart of a daemon.

SOCKET_LISTEN_BACKLOG
An integer value that defaults to 4096, which defines the backlog value for the listen() network call when a daemon creates a socket for incoming connections. It limits the number of new incoming network connections the operating system will accept for a daemon that the daemon has not yet serviced.

MAX_ACCEPTS_PER_CYCLE
An integer value that defaults to 8. It is a rarely changed performance tuning parameter to limit the number of accepts of new, incoming, socket connect requests per DaemonCore event cycle. A value of zero or less means no limit. It has the most noticeable effect on the *condor_schedd*, and would be given a higher integer value for tuning purposes when there is a high number of jobs starting and exiting per second.

MAX_TIMER_EVENTS_PER_CYCLE
An integer value that defaults to 3. It is a rarely changed performance tuning parameter to set the max number of internal timer events will be dispatched per DaemonCore event cycle. A value of zero means no limit, so that all timers that are due at the start of the event cycle should be dispatched.

MAX_UDP_MSGS_PER_CYCLE
An integer value that defaults to 1. It is a rarely changed performance tuning parameter to set the number of incoming UDP messages a daemon will read per DaemonCore event cycle. A value of zero means no limit. It has the most noticeable effect on the *condor_schedd* and *condor_collector* daemons, which can receive a large number of UDP messages when under heavy load.

MAX_REAPS_PER_CYCLE
An integer value that defaults to 0. It is a rarely changed performance tuning parameter that places a limit on the number of child process exits to process per DaemonCore event cycle. A value of zero or less means no limit.

CORE_FILE_NAME
Defines the name of the core file created on Windows platforms. Defaults to `core.$(SUBSYSTEM).WIN32`.

PIPE_BUFFER_MAX
The maximum number of bytes read from a `stdout` or `stdout` pipe. The default value is 10240. A rare example in which the value would need to increase from its default value is when a hook must output an entire ClassAd, and the ClassAd may be larger than the default.

5.5.4 Network-Related Configuration File Entries

More information about networking in HTCondor can be found in *Networking (includes sections on Port Usage and CCB)*.

BIND_ALL_INTERFACES
For systems with multiple network interfaces, if this configuration setting is `False`, HTCondor will only bind network sockets to the IP address specified with `NETWORK_INTERFACE` (described below). If set to `True`, the default value, HTCondor will listen on all interfaces. However, currently HTCondor is still only able to advertise

a single IP address, even if it is listening on multiple interfaces. By default, it will advertise the IP address of the network interface used to contact the collector, since this is the most likely to be accessible to other processes which query information from the same collector. More information about using this setting can be found in *Configuring HTCondor for Machines With Multiple Network Interfaces*.

CCB_ADDRESS
This is the address of a *condor_collector* that will serve as this daemon's HTCondor Connection Broker (CCB). Multiple addresses may be listed (separated by commas and/or spaces) for redundancy. The CCB server must authorize this daemon at DAEMON level for this configuration to succeed. It is highly recommended to also configure `PRIVATE_NETWORK_NAME` if you configure `CCB_ADDRESS` so communications originating within the same private network do not need to go through CCB. For more information about CCB, see *HTCondor Connection Brokering (CCB)*.

CCB_HEARTBEAT_INTERVAL
This is the maximum number of seconds of silence on a daemon's connection to the CCB server after which it will ping the server to verify that the connection still works. The default is 5 minutes. This feature serves to both speed up detection of dead connections and to generate a guaranteed minimum frequency of activity to attempt to prevent the connection from being dropped. The special value 0 disables the heartbeat. The heartbeat is automatically disabled if the CCB server is older than HTCondor version 7.5.0. Having the heartbeat interval greater than the job ClassAd attribute `JobLeaseDuration` may cause unnecessary job disconnects in pools with network issues.

CCB_POLLING_INTERVAL
In seconds, the smallest amount of time that could go by before CCB would begin another round of polling to check on already connected clients. While the value of this variable does not change, the actual interval used may be exceeded if the measured amount of time previously taken to poll to check on already connected clients exceeded the amount of time desired, as expressed with `CCB_POLLING_TIMESLICE`. The default value is 20 seconds.

CCB_POLLING_MAX_INTERVAL
In seconds, the interval of time after which polling to check on already connected clients must occur, independent of any other factors. The default value is 600 seconds.

CCB_POLLING_TIMESLICE
A floating point fraction representing the fractional amount of the total run time of CCB to set as a target for the maximum amount of CCB running time used on polling to check on already connected clients. The default value is 0.05.

CCB_READ_BUFFER
The size of the kernel TCP read buffer in bytes for all sockets used by CCB. The default value is 2 KiB.

CCB_REQUIRED_TO_START
If true, and is false, and is set, but HTCondor fails to register with any broker, HTCondor will exit rather then continue to retry indefinitely.

CCB_TIMEOUT
The length, in seconds, that we wait for any CCB operation to complete. The default value is 300.

CCB_WRITE_BUFFER
The size of the kernel TCP write buffer in bytes for all sockets used by CCB. The default value is 2 KiB.

CCB_SWEEP_INTERVAL
The interval, in seconds, between times when the CCB server writes its information about open TCP connections to a file. Crash recovery is accomplished using the information. The default value is 1200 seconds (20 minutes).

CCB_RECONNECT_FILE
The full path and file name of the file that the CCB server writes its information about open TCP connections to a file. Crash recovery is accomplished using the information. The default value is `$(SPOOL)/<ip address>-<shared port ID or port number>.ccb_reconnect`.

COLLECTOR_USES_SHARED_PORT
A boolean value that specifies whether the *condor_collector* uses the *condor_shared_port* daemon. When true, the *condor_shared_port* will transparently proxy queries to the *condor_collector* so users do not need to be aware of the presence of the *condor_shared_port* when querying the collector and configuring other daemons. The default is `True`

SHARED_PORT_DEFAULT_ID
When `COLLECTOR_USES_SHARED_PORT` is set to `True`, this is the shared port ID used by the *condor_collector*. This defaults to `collector` and will not need to be changed by most sites.

AUTO_INCLUDE_SHARED_PORT_IN_DAEMON_LIST
A boolean value that specifies whether `SHARED_PORT` should be automatically inserted into *condor_master* 's `DAEMON_LIST` when `USE_SHARED_PORT` is `True`. The default for this setting is `True`.

<SUBSYS>_MAX_FILE_DESCRIPTORS
This setting is identical to `MAX_FILE_DESCRIPTORS`, but it only applies to a specific subsystem. If the subsystem-specific setting is unspecified, `MAX_FILE_DESCRIPTORS` is used. For the *condor_collector* daemon, the value defaults to 10240, and for the *condor_schedd* daemon, the value defaults to 4096. If the *condor_shared_port* daemon is in use, its value for this parameter should match the largest value set for the other daemons.

List of possible subsystems to set <SUBSYS> can be found at .

MAX_FILE_DESCRIPTORS
Under Unix, this specifies the maximum number of file descriptors to allow the HTCondor daemon to use. File descriptors are a system resource used for open files and for network connections. HTCondor daemons that make many simultaneous network connections may require an increased number of file descriptors. For example, see *HTCondor Connection Brokering (CCB)* for information on file descriptor requirements of CCB. Changes to this configuration variable require a restart of HTCondor in order to take effect. Also note that only if HTCondor is running as root will it be able to increase the limit above the hard limit (on maximum open files) that it inherits.

NETWORK_HOSTNAME
The name HTCondor should use as the host name of the local machine, overriding the value returned by gethostname(). Among other things, the host name is used to identify daemons in an HTCondor pool, via the `Machine` and `Name` attributes of daemon ClassAds. This variable can be used when a machine has multiple network interfaces with different host names, to use a host name that is not the primary one. It should be set to a fully-qualified host name that will resolve to an IP address of the local machine.

NETWORK_INTERFACE
An IP address of the form `123.123.123.123` or the name of a network device, as in the example `eth0`. The wild card character (*) may be used within either. For example, `123.123.*` would match a network interface with an IP address of `123.123.123.123` or `123.123.100.100`. The default value is *, which matches all network interfaces.

The effect of this variable depends on the value of `BIND_ALL_INTERFACES`. There are two cases:

If `BIND_ALL_INTERFACES` is `True` (the default), `NETWORK_INTERFACE` controls what IP address will be advertised as the public address of the daemon. If multiple network interfaces match the value, the IP address that is chosen to be advertised will be the one associated with the first device (in system-defined order) that is in a public address space, or a private address space, or a loopback address, in that order of preference. If it is desired to advertise an IP address that is not associated with any local network interface, for example, when TCP forwarding is being used, then `TCP_FORWARDING_HOST` should be used instead of `NETWORK_INTERFACE`.

If `BIND_ALL_INTERFACES` is `False`, then `NETWORK_INTERFACE` specifies which IP address HTCondor should use for all incoming and outgoing communication. If more than one IP address matches the value, then the IP address that is chosen will be the one associated with the first device (in system-defined order) that is in a public address space, or a private address space, or a loopback address, in that order of preference.

More information about configuring HTCondor on machines with multiple network interfaces can be found in *Configuring HTCondor for Machines With Multiple Network Interfaces*.

PRIVATE_NETWORK_NAME
If two HTCondor daemons are trying to communicate with each other, and they both belong to the same private network, this setting will allow them to communicate directly using the private network interface, instead of having to use CCB or to go through a public IP address. Each private network should be assigned a unique network name. This string can have any form, but it must be unique for a particular private network. If another HTCondor daemon or tool is configured with the same PRIVATE_NETWORK_NAME, it will attempt to contact this daemon using its private network address. Even for sites using CCB, this is an important optimization, since it means that two daemons on the same network can communicate directly, without having to go through the broker. If CCB is enabled, and the PRIVATE_NETWORK_NAME is defined, the daemon's private address will be defined automatically. Otherwise, you can specify a particular private IP address to use by defining the PRIVATE_NETWORK_INTERFACE setting (described below). The default is $(FULL_HOSTNAME). After changing this setting and running *condor_reconfig*, it may take up to one *condor_collector* update interval before the change becomes visible.

PRIVATE_NETWORK_INTERFACE
For systems with multiple network interfaces, if this configuration setting and PRIVATE_NETWORK_NAME are both defined, HTCondor daemons will advertise some additional attributes in their ClassAds to help other HTCondor daemons and tools in the same private network to communicate directly.

PRIVATE_NETWORK_INTERFACE defines what IP address of the form 123.123.123.123 or name of a network device (as in the example eth0) a given multi-homed machine should use for the private network. The asterisk (*) may be used as a wild card character within either the IP address or the device name. If another HTCondor daemon or tool is configured with the same PRIVATE_NETWORK_NAME, it will attempt to contact this daemon using the IP address specified here. The syntax for specifying an IP address is identical to NETWORK_INTERFACE. Sites using CCB only need to define the PRIVATE_NETWORK_NAME, and the PRIVATE_NETWORK_INTERFACE will be defined automatically. Unless CCB is enabled, there is no default value for this variable. After changing this variable and running *condor_reconfig*, it may take up to one *condor_collector* update interval before the change becomes visible.

TCP_FORWARDING_HOST
This specifies the host or IP address that should be used as the public address of this daemon. If a host name is specified, be aware that it will be resolved to an IP address by this daemon, not by the clients wishing to connect to it. It is the IP address that is advertised, not the host name. This setting is useful if HTCondor on this host may be reached through a NAT or firewall by connecting to an IP address that forwards connections to this host. It is assumed that the port number on the TCP_FORWARDING_HOST that forwards to this host is the same port number assigned to HTCondor on this host. This option could also be used when ssh port forwarding is being used. In this case, the incoming addresses of connections to this daemon will appear as though they are coming from the forwarding host rather than from the real remote host, so any authorization settings that rely on host addresses should be considered accordingly.

HIGHPORT
Specifies an upper limit of given port numbers for HTCondor to use, such that HTCondor is restricted to a range of port numbers. If this macro is not explicitly specified, then HTCondor will not restrict the port numbers that it uses. HTCondor will use system-assigned port numbers. For this macro to work, both HIGHPORT and LOWPORT (given below) must be defined.

LOWPORT
Specifies a lower limit of given port numbers for HTCondor to use, such that HTCondor is restricted to a range of port numbers. If this macro is not explicitly specified, then HTCondor will not restrict the port numbers that it uses. HTCondor will use system-assigned port numbers. For this macro to work, both HIGHPORT (given above) and LOWPORT must be defined.

IN_LOWPORT
An integer value that specifies a lower limit of given port numbers for HTCondor to use on incoming connections (ports for listening), such that HTCondor is restricted to a range of port numbers. This range implies the use of both IN_LOWPORT and IN_HIGHPORT. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify IN_LOWPORT in combination with IN_HIGHPORT such that the range crosses the port

1024 boundary. Applies only to Unix machine configuration. Use of IN_LOWPORT and IN_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

IN_HIGHPORT
An integer value that specifies an upper limit of given port numbers for HTCondor to use on incoming connections (ports for listening), such that HTCondor is restricted to a range of port numbers. This range implies the use of both IN_LOWPORT and IN_HIGHPORT. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify IN_LOWPORT in combination with IN_HIGHPORT such that the range crosses the port 1024 boundary. Applies only to Unix machine configuration. Use of IN_LOWPORT and IN_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

OUT_LOWPORT
An integer value that specifies a lower limit of given port numbers for HTCondor to use on outgoing connections, such that HTCondor is restricted to a range of port numbers. This range implies the use of both OUT_LOWPORT and OUT_HIGHPORT. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of OUT_LOWPORT and OUT_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

OUT_HIGHPORT
An integer value that specifies an upper limit of given port numbers for HTCondor to use on outgoing connections, such that HTCondor is restricted to a range of port numbers. This range implies the use of both OUT_LOWPORT and OUT_HIGHPORT. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of OUT_LOWPORT and OUT_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

UPDATE_COLLECTOR_WITH_TCP
This boolean value controls whether TCP or UDP is used by daemons to send ClassAd updates to the *condor_collector*. Please read *Using TCP to Send Updates to the condor_collector* for more details and a discussion of when this functionality is needed. When using TCP in large pools, it is also necessary to ensure that the *condor_collector* has a large enough file descriptor limit using COLLECTOR_MAX_FILE_DESCRIPTORS . The default value is True.

UPDATE_VIEW_COLLECTOR_WITH_TCP
This boolean value controls whether TCP or UDP is used by the *condor_collector* to forward ClassAd updates to the *condor_collector* daemons specified by CONDOR_VIEW_HOST . Please read *Using TCP to Send Updates to the condor_collector* for more details and a discussion of when this functionality is needed. The default value is False.

TCP_UPDATE_COLLECTORS
The list of *condor_collector* daemons which will be updated with TCP instead of UDP when UPDATE_COLLECTOR_WITH_TCP or UPDATE_VIEW_COLLECTOR_WITH_TCP is False. Please read *Using TCP to Send Updates to the condor_collector* for more details and a discussion of when a site needs this functionality.

<SUBSYS>_TIMEOUT_MULTIPLIER
An integer value that defaults to 1. This value multiplies configured timeout values for all targeted subsystem communications, thereby increasing the time until a timeout occurs. This configuration variable is intended for use by developers for debugging purposes, where communication timeouts interfere.

List of possible subsystems to set <SUBSYS> can be found at .

NONBLOCKING_COLLECTOR_UPDATE
A boolean value that defaults to True. When True, the establishment of TCP connections to the *condor_collector* daemon for a security-enabled pool are done in a nonblocking manner.

NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT
A boolean value that defaults to True. When True, the establishment of TCP connections from the *condor_negotiator* daemon to the *condor_startd* daemon for a security-enabled pool are done in a nonblocking manner.

UDP_NETWORK_FRAGMENT_SIZE
An integer value that defaults to 1000 and represents the maximum size in bytes of an outgoing UDP packet. If the outgoing message is larger than `$(UDP_NETWORK_FRAGMENT_SIZE)`, then the message will be split (fragmented) into multiple packets no larger than `$(UDP_NETWORK_FRAGMENT_SIZE)`. If the destination of the message is the loopback network interface, see `UDP_LOOPBACK_FRAGMENT_SIZE` below. For instance, the maximum payload size of a UDP packet over Ethernet is typically 1472 bytes, and thus if a UDP payload exceeds 1472 bytes the IP network stack on either hosts or forwarding devices (such as network routers) will have to perform message fragmentation on transmission and reassembly on receipt. Experimentation has shown that such devices are more likely to simply drop a UDP message under high-traffic scenarios if the message requires reassembly. HTCondor avoids this situation via the capability to perform UDP fragmentation and reassembly on its own.

UDP_LOOPBACK_FRAGMENT_SIZE
An integer value that defaults to 60000 and represents the maximum size in bytes of an outgoing UDP packet that is being sent to the loopback network interface (e.g. 127.0.0.1). If the outgoing message is larger than `$(UDP_LOOPBACK_FRAGMENT_SIZE)`, then the message will be split (fragmented) into multiple packets no larger than `$(UDP_LOOPBACK_FRAGMENT_SIZE)`. If the destination of the message is not the loopback interface, see `UDP_NETWORK_FRAGMENT_SIZE` above.

ALWAYS_REUSEADDR
A boolean value that, when `True`, tells HTCondor to set `SO_REUSEADDR` socket option, so that the schedd can run large numbers of very short jobs without exhausting the number of local ports needed for shadows. The default value is `True`. (Note that this represents a change in behavior compared to versions of HTCondor older than 8.6.0, which did not include this configuration macro. To restore the previous behavior, set this value to `False`.)

5.5.5 Shared File System Configuration File Macros

These macros control how HTCondor interacts with various shared and network file systems. If you are using AFS as your shared file system, be sure to read *Using HTCondor with AFS*. For information on submitting jobs under shared file systems, see *Submitting Jobs Using a Shared File System*.

UID_DOMAIN
The `UID_DOMAIN` macro is used to decide under which user to run jobs. If the `$(UID_DOMAIN)` on the submitting machine is different than the `$(UID_DOMAIN)` on the machine that runs a job, then HTCondor runs the job as the user nobody. For example, if the access point has a `$(UID_DOMAIN)` of flippy.cs.wisc.edu, and the machine where the job will execute has a `$(UID_DOMAIN)` of cs.wisc.edu, the job will run as user nobody, because the two `$(UID_DOMAIN)`s are not the same. If the `$(UID_DOMAIN)` is the same on both the submit and execute machines, then HTCondor will run the job as the user that submitted the job.

A further check attempts to assure that the submitting machine can not lie about its `UID_DOMAIN`. HTCondor compares the submit machine's claimed value for `UID_DOMAIN` to its fully qualified name. If the two do not end the same, then the access point is presumed to be lying about its `UID_DOMAIN`. In this case, HTCondor will run the job as user nobody. For example, a job submission to the HTCondor pool at the UW Madison from flippy.example.com, claiming a `UID_DOMAIN` of of cs.wisc.edu, will run the job as the user nobody.

Because of this verification, `$(UID_DOMAIN)` must be a real domain name. At the Computer Sciences department at the UW Madison, we set the `$(UID_DOMAIN)` to be cs.wisc.edu to indicate that whenever someone submits from a department machine, we will run the job as the user who submits it.

Also see `SOFT_UID_DOMAIN` below for information about one more check that HTCondor performs before running a job as a given user.

A few details:

An administrator could set `UID_DOMAIN` to *. This will match all domains, but it is a gaping security hole. It is not recommended.

An administrator can also leave UID_DOMAIN undefined. This will force HTCondor to always run jobs as user nobody. If vanilla jobs are run as user nobody, then files that need to be accessed by the job will need to be marked as world readable/writable so the user nobody can access them.

When HTCondor sends e-mail about a job, HTCondor sends the e-mail to user@$(UID_DOMAIN). If UID_DOMAIN is undefined, the e-mail is sent to user@submitmachinename.

TRUST_UID_DOMAIN
As an added security precaution when HTCondor is about to spawn a job, it ensures that the UID_DOMAIN of a given access point is a substring of that machine's fully-qualified host name. However, at some sites, there may be multiple UID spaces that do not clearly correspond to Internet domain names. In these cases, administrators may wish to use names to describe the UID domains which are not substrings of the host names of the machines. For this to work, HTCondor must not do this regular security check. If the TRUST_UID_DOMAIN setting is defined to True, HTCondor will not perform this test, and will trust whatever UID_DOMAIN is presented by the access point when trying to spawn a job, instead of making sure the access point's host name matches the UID_DOMAIN. When not defined, the default is False, since it is more secure to perform this test.

TRUST_LOCAL_UID_DOMAIN
This parameter works like TRUST_UID_DOMAIN, but is only applied when the *condor_starter* and *condor_shadow* are on the same machine. If this parameter is set to True, then the *condor_shadow* 's UID_DOMAIN doesn't have to be a substring its hostname. If this parameter is set to False, then UID_DOMAIN controls whether this substring requirement is enforced by the *condor_starter*. The default is True.

SOFT_UID_DOMAIN
A boolean variable that defaults to False when not defined. When HTCondor is about to run a job as a particular user (instead of as user nobody), it verifies that the UID given for the user is in the password file and actually matches the given user name. However, under installations that do not have every user in every machine's password file, this check will fail and the execution attempt will be aborted. To cause HTCondor not to do this check, set this configuration variable to True. HTCondor will then run the job under the user's UID.

SLOT<N>_USER
The name of a user for HTCondor to use instead of user nobody, as part of a solution that plugs a security hole whereby a lurker process can prey on a subsequent job run as user name nobody. <N> is an integer associated with slots. On non Windows platforms you can use NOBODY_SLOT_USER instead of this configuration variable. On Windows, SLOT<N>_USER will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. See *User Accounts in HTCondor on Unix Platforms* for more information.

NOBODY_SLOT_USER
The name of a user for HTCondor to use instead of user nobody when The SLOT<N>_USER for this slot is not configured. Configure this to the value $(STARTER_SLOT_NAME) to use the name of the slot as the user name. This configuration macro is ignored on Windows, where the Starter will automatically create a unique temporary user for each slot as needed. See *User Accounts in HTCondor on Unix Platforms* for more information.

STARTER_ALLOW_RUNAS_OWNER
A boolean expression evaluated with the job ad as the target, that determines whether the job may run under the job owner's account (True) or whether it will run as SLOT<N>_USER or nobody (False). On Unix, this defaults to True. On Windows, it defaults to False. The job ClassAd may also contain the attribute RunAsOwner which is logically ANDed with the *condor_starter* daemon's boolean value. Under Unix, if the job does not specify it, this attribute defaults to True. Under Windows, the attribute defaults to False. In Unix, if the UidDomain of the machine and job do not match, then there is no possibility to run the job as the owner anyway, so, in that case, this setting has no effect. See *User Accounts in HTCondor on Unix Platforms* for more information.

DEDICATED_EXECUTE_ACCOUNT_REGEXP
This is a regular expression (i.e. a string matching pattern) that matches the account name(s) that are dedicated to running condor jobs on the execute machine and which will never be used for more than one job at a time. The default matches no account name. If you have configured SLOT<N>_USER to be a different account for each HTCondor slot, and no non-condor processes will ever be run by these accounts, then this pattern should match the names of all SLOT<N>_USER accounts. Jobs run under a dedicated execute account are reliably tracked by

HTCondor, whereas other jobs, may spawn processes that HTCondor fails to detect. Therefore, a dedicated execution account provides more reliable tracking of CPU usage by the job and it also guarantees that when the job exits, no "lurker" processes are left behind. When the job exits, condor will attempt to kill all processes owned by the dedicated execution account. Example:

```
SLOT1_USER = cndrusr1
SLOT2_USER = cndrusr2
STARTER_ALLOW_RUNAS_OWNER = False
DEDICATED_EXECUTE_ACCOUNT_REGEXP = cndrusr[0-9]+
```

You can tell if the starter is in fact treating the account as a dedicated account, because it will print a line such as the following in its log file:

```
Tracking process family by login "cndrusr1"
```

EXECUTE_LOGIN_IS_DEDICATED
This configuration setting is deprecated because it cannot handle the case where some jobs run as dedicated accounts and some do not. Use `DEDICATED_EXECUTE_ACCOUNT_REGEXP` instead.

A boolean value that defaults to `False`. When `True`, HTCondor knows that all jobs are being run by dedicated execution accounts (whether they are running as the job owner or as nobody or as `SLOT<N>_USER`). Therefore, when the job exits, all processes running under the same account will be killed.

FILESYSTEM_DOMAIN
An arbitrary string that is used to decide if the two machines, a access point and an execute machine, share a file system. Although this configuration variable name contains the word "DOMAIN", its value is not required to be a domain name. It often is a domain name.

Note that this implementation is not ideal: machines may share some file systems but not others. HTCondor currently has no way to express this automatically. A job can express the need to use a particular file system where machines advertise an additional ClassAd attribute and the job requires machines with the attribute, as described on the question within the https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToAdminRecipes page for how to run jobs on a subset of machines that have required software installed.

Note that if you do not set `$(FILESYSTEM_DOMAIN)`, the value defaults to the fully qualified host name of the local machine. Since each machine will have a different `$(FILESYSTEM_DOMAIN)`, they will not be considered to have shared file systems.

USE_NFS
This configuration variable changes the semantics of Chirp file I/O when running in the vanilla, java or parallel universe. If this variable is set in those universes, Chirp will not send I/O requests over the network as requested, but perform them directly to the locally mounted file system.

IGNORE_NFS_LOCK_ERRORS
When set to `True`, all errors related to file locking errors from NFS are ignored. Defaults to `False`, not ignoring errors.

5.5.6 condor_master Configuration File Macros

These macros control the *condor_master*.

DAEMON_LIST
This macro determines what daemons the *condor_master* will start and keep its watchful eyes on. The list is a comma or space separated list of subsystem names (listed in *Pre-Defined Macros*). For example,

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

Note: The *condor_shared_port* daemon will be included in this list automatically when USE_SHARED_PORT is configured to True. While adding SHARED_PORT to the DAEMON_LIST without setting USE_SHARED_PORT to True will start the *condor_shared_port* daemon, but it will not be used. So there is generally no point in adding SHARED_PORT to the daemon list.

Note: On your central manager, your $(DAEMON_LIST) will be different from your regular pool, since it will include entries for the *condor_collector* and *condor_negotiator*.

DC_DAEMON_LIST
A list delimited by commas and/or spaces that lists the daemons in DAEMON_LIST which use the HTCondor DaemonCore library. The *condor_master* must differentiate between daemons that use DaemonCore and those that do not, so it uses the appropriate inter-process communication mechanisms. This list currently includes all HTCondor daemons.

As of HTCondor version 7.2.1, a daemon may be appended to the default DC_DAEMON_LIST value by placing the plus character (+) before the first entry in the DC_DAEMON_LIST definition. For example:

```
DC_DAEMON_LIST = +NEW_DAEMON
```

<SUBSYS>
Once you have defined which subsystems you want the *condor_master* to start, you must provide it with the full path to each of these binaries. For example:

```
MASTER          = $(SBIN)/condor_master
STARTD          = $(SBIN)/condor_startd
SCHEDD          = $(SBIN)/condor_schedd
```

These are most often defined relative to the $(SBIN) macro.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined by .

<DaemonName>_ENVIRONMENT
<DaemonName> is the name of a daemon listed in DAEMON_LIST. Defines changes to the environment that the daemon is invoked with. It should use the same syntax for specifying the environment as the environment specification in a submit description file. For example, to redefine the TMP and CONDOR_CONFIG environment variables seen by the *condor_schedd*, place the following in the configuration:

```
SCHEDD_ENVIRONMENT = "TMP=/new/value CONDOR_CONFIG=/special/config"
```

When the *condor_schedd* daemon is started by the *condor_master*, it would see the specified values of TMP and CONDOR_CONFIG.

<SUBSYS>_ARGS
This macro allows the specification of additional command line arguments for any process spawned by the *condor_master*. List the desired arguments using the same syntax as the arguments specification in a *condor_submit* submit file (see *condor_submit*), with one exception: do not escape double-quotes when using the old-style syntax (this is for backward compatibility). Set the arguments for a specific daemon with this macro, and the macro will affect only that daemon. Define one of these for each daemon the *condor_master* is controlling. For example, set `$(STARTD_ARGS)` to specify any extra command line arguments to the *condor_startd*.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined by .

<SUBSYS>_USERID
The account name that should be used to run the `SUBSYS` process spawned by the *condor_master*. When not defined, the process is spawned as the same user that is running *condor_master*. When defined, the real user id of the spawned process will be set to the specified account, so if this account is not root, the process will not have root privileges. The *condor_master* must be running as root in order to start processes as other users. Example configuration:

```
COLLECTOR_USERID = condor
NEGOTIATOR_USERID = condor
```

The above example runs the *condor_collector* and *condor_negotiator* as the condor user with no root privileges. If we specified some account other than the condor user, as set by the (`CONDOR_IDS`) configuration variable, then we would need to configure the log files for these daemons to be in a directory that they can write to. When using a security method in which the daemon credential is owned by root, it is also necessary to make a copy of the credential, make it be owned by the account the daemons are using, and configure the daemons to use that copy.

List of possible subsystems to set <SUBSYS> can be found at .

PREEN
In addition to the daemons defined in `$(DAEMON_LIST)`, the *condor_master* also starts up a special process, *condor_preen* to clean out junk files that have been left laying around by HTCondor. This macro determines where the *condor_master* finds the *condor_preen* binary. If this macro is set to nothing, *condor_preen* will not run.

PREEN_ARGS
Controls how *condor_preen* behaves by allowing the specification of command-line arguments. This macro works as `$(<SUBSYS>_ARGS)` does. The difference is that you must specify this macro for *condor_preen* if you want it to do anything. *condor_preen* takes action only because of command line arguments. **-m** means you want e-mail about files *condor_preen* finds that it thinks it should remove. **-r** means you want *condor_preen* to actually remove these files.

PREEN_INTERVAL
This macro determines how often *condor_preen* should be started. It is defined in terms of seconds and defaults to 86400 (once a day).

PUBLISH_OBITUARIES
When a daemon crashes, the *condor_master* can send e-mail to the address specified by `$(CONDOR_ADMIN)` with an obituary letting the administrator know that the daemon died, the cause of death (which signal or exit status it exited with), and (optionally) the last few entries from that daemon's log file. If you want obituaries, set this macro to `True`.

OBITUARY_LOG_LENGTH
This macro controls how many lines of the log file are part of obituaries. This macro has a default value of 20 lines.

START_MASTER
If this setting is defined and set to `False` the *condor_master* will immediately exit upon startup. This appears strange, but perhaps you do not want HTCondor to run on certain machines in your pool, yet the boot scripts for your entire pool are handled by a centralized set of files - setting `START_MASTER` to `False` for those machines

would allow this. Note that START_MASTER is an entry you would most likely find in a local configuration file, not a global configuration file. If not defined, START_MASTER defaults to True.

START_DAEMONS
This macro is similar to the $(START_MASTER) macro described above. However, the *condor_master* does not exit; it does not start any of the daemons listed in the $(DAEMON_LIST). The daemons may be started at a later time with a *condor_on* command.

MASTER_UPDATE_INTERVAL
This macro determines how often the *condor_master* sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_CHECK_NEW_EXEC_INTERVAL
This macro controls how often the *condor_master* checks the timestamps of the running daemons. If any daemons have been modified, the master restarts them. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_NEW_BINARY_RESTART
Defines a mode of operation for the restart of the *condor_master*, when it notices that the *condor_master* binary has changed. Valid values are GRACEFUL, PEACEFUL, and NEVER, with a default value of GRACEFUL. On a GRACEFUL restart of the master, child processes are told to exit, but if they do not before a timer expires, then they are killed. On a PEACEFUL restart, child processes are told to exit, after which the *condor_master* waits until they do so.

MASTER_NEW_BINARY_DELAY
Once the *condor_master* has discovered a new binary, this macro controls how long it waits before attempting to execute the new binary. This delay exists because the *condor_master* might notice a new binary while it is in the process of being copied, in which case trying to execute it yields unpredictable results. The entry is defined in seconds and defaults to 120 (2 minutes).

SHUTDOWN_FAST_TIMEOUT
This macro determines the maximum amount of time daemons are given to perform their fast shutdown procedure before the *condor_master* kills them outright. It is defined in seconds and defaults to 300 (5 minutes).

DEFAULT_MASTER_SHUTDOWN_SCRIPT
A full path and file name of a program that the *condor_master* is to execute via the Unix execl() call, or the similar Win32 _execl() call, instead of the normal call to exit(). This allows the admin to specify a program to execute as root when the *condor_master* exits. Note that a successful call to the *condor_set_shutdown* program will override this setting; see the documentation for config knob MASTER_SHUTDOWN_<Name> below.

MASTER_SHUTDOWN_<Name>
A full path and file name of a program that the *condor_master* is to execute via the Unix execl() call, or the similar Win32 _execl() call, instead of the normal call to exit(). Multiple programs to execute may be defined with multiple entries, each with a unique Name. These macros have no effect on a *condor_master* unless *condor_set_shutdown* is run, or the *-exec* argument is used with *condor_off* or *condor_restart*. The Name specified as an argument to the *condor_set_shutdown* program or *-exec* arg must match the Name portion of one of these MASTER_SHUTDOWN_<Name> macros; if not, the *condor_master* will log an error and ignore the command. If a match is found, the *condor_master* will attempt to verify the program, and it will store the path and program name. When the *condor_master* shuts down (that is, just before it exits), the program is then executed as described above. The manual page for *condor_set_shutdown* contains details on the use of this program.

NOTE: This program will be run with root privileges under Unix or administrator privileges under Windows. The administrator must ensure that this cannot be used in such a way as to violate system integrity.

MASTER_BACKOFF_CONSTANT and MASTER_<name>_BACKOFF_CONSTANT
When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. These settings define the constant value of the expression used to determine how long to wait before starting the daemon again (and, effectively becomes the initial backoff time). It is an integer in units of seconds, and defaults to 9 seconds.

$(MASTER_<name>_BACKOFF_CONSTANT) is the daemon-specific form of MASTER_BACKOFF_CONSTANT; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

MASTER_BACKOFF_FACTOR and MASTER_<name>_BACKOFF_FACTOR
When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This setting is the base of the exponent used to determine how long to wait before starting the daemon again. It defaults to 2 seconds.

$(MASTER_<name>_BACKOFF_FACTOR) is the daemon-specific form of MASTER_BACKOFF_FACTOR; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

MASTER_BACKOFF_CEILING and MASTER_<name>_BACKOFF_CEILING
When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This entry determines the maximum amount of time you want the master to wait between attempts to start a given daemon. (With 2.0 as the $(MASTER_BACKOFF_FACTOR), 1 hour is obtained in 12 restarts). It is defined in terms of seconds and defaults to 3600 (1 hour).

$(MASTER_<name>_BACKOFF_CEILING) is the daemon-specific form of MASTER_BACKOFF_CEILING; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

MASTER_RECOVER_FACTOR and MASTER_<name>_RECOVER_FACTOR
A macro to set how long a daemon needs to run without crashing before it is considered recovered. Once a daemon has recovered, the number of restarts is reset, so the exponential back off returns to its initial state. The macro is defined in terms of seconds and defaults to 300 (5 minutes).

$(MASTER_<name>_RECOVER_FACTOR) is the daemon-specific form of MASTER_RECOVER_FACTOR; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

When a daemon crashes, *condor_master* will restart the daemon after a delay (a back off). The length of this delay is based on how many times it has been restarted, and gets larger after each crashes. The equation for calculating this backoff time is given by:

$$t = c + k^n$$

where t is the calculated time, c is the constant defined by $(MASTER_BACKOFF_CONSTANT), k is the "factor" defined by $(MASTER_BACKOFF_FACTOR), and n is the number of restarts already attempted (0 for the first restart, 1 for the next, etc.).

With default values, after the first crash, the delay would be t = 9 + 2.0^0, giving 10 seconds (remember, n = 0). If the daemon keeps crashing, the delay increases.

For example, take the $(MASTER_BACKOFF_FACTOR) (which defaults to 2.0) to the power the number of times the daemon has restarted, and add $(MASTER_BACKOFF_CONSTANT) (which defaults to 9). Thus:

1^{st} crash: n = 0, so: t = 9 + 2^0 = 9 + 1 = 10 seconds

2^{nd} crash: n = 1, so: t = 9 + 2^1 = 9 + 2 = 11 seconds

3^{rd} crash: n = 2, so: t = 9 + 2^2 = 9 + 4 = 13 seconds

...

6^{th} crash: n = 5, so: t = 9 + 2^5 = 9 + 32 = 41 seconds

...

9^{th} crash: n = 8, so: t = 9 + 2^8 = 9 + 256 = 265 seconds

And, after the 13 crashes, it would be:

13^{th} crash: n = 12, so: t = 9 + 2^{12} = 9 + 4096 = 4105 seconds

This is bigger than the $(MASTER_BACKOFF_CEILING), which defaults to 3600, so the daemon would really be restarted after only 3600 seconds, not 4105. The *condor_master* tries again every hour (since the numbers would get larger and would always be capped by the ceiling). Eventually, imagine that daemon finally started and did not crash. This might happen if, for example, an administrator reinstalled an accidentally deleted binary after receiving e-mail about the daemon crashing. If it stayed alive for $(MASTER_RECOVER_FACTOR) seconds (defaults to 5 minutes), the count of how many restarts this daemon has performed is reset to 0.

The moral of the example is that the defaults work quite well, and you probably will not want to change them for any reason.

MASTER_NAME
Defines a unique name given for a *condor_master* daemon on a machine. For a *condor_master* running as root, it defaults to the fully qualified host name. When not running as root, it defaults to the user that instantiates the *condor_master*, concatenated with an at symbol (@), concatenated with the fully qualified host name. If more than one *condor_master* is running on the same host, then the MASTER_NAME for each *condor_master* must be defined to uniquely identify the separate daemons.

A defined MASTER_NAME is presumed to be of the form identifying-string@full.host.name. If the string does not include an @ sign, HTCondor appends one, followed by the fully qualified host name of the local machine. The identifying-string portion may contain any alphanumeric ASCII characters or punctuation marks, except the @ sign. We recommend that the string does not contain the : (colon) character, since that might cause problems with certain tools. Previous to HTCondor 7.1.1, when the string included an @ sign, HTCondor replaced whatever followed the @ sign with the fully qualified host name of the local machine. HTCondor does not modify any portion of the string, if it contains an @ sign. This is useful for remote job submissions under the high availability of the job queue.

If the MASTER_NAME setting is used, and the *condor_master* is configured to spawn a *condor_schedd*, the name defined with MASTER_NAME takes precedence over the setting. Since HTCondor makes the assumption that there is only one instance of the *condor_startd* running on a machine, the MASTER_NAME is not automatically propagated to the *condor_startd*. However, in situations where multiple *condor_startd* daemons are running on the same host, the STARTD_NAME should be set to uniquely identify the *condor_startd* daemons.

If an HTCondor daemon (master, schedd or startd) has been given a unique name, all HTCondor tools that need to contact that daemon can be told what name to use via the **-name** command-line option.

MASTER_ATTRS
This macro is described in .

MASTER_DEBUG
This macro is described in .

MASTER_ADDRESS_FILE
This macro is described in .

ALLOW_ADMIN_COMMANDS
If set to NO for a given host, this macro disables administrative commands, such as *condor_restart*, *condor_on*, and *condor_off*, to that host.

MASTER_INSTANCE_LOCK
Defines the name of a file for the *condor_master* daemon to lock in order to prevent multiple *condor_master* s from starting. This is useful when using shared file systems like NFS which do not technically support locking in the case where the lock files reside on a local disk. If this macro is not defined, the default file name will be $(LOCK)/InstanceLock. $(LOCK) can instead be defined to specify the location of all lock files, not just the *condor_master* 's InstanceLock. If $(LOCK) is undefined, then the master log itself is locked.

ADD_WINDOWS_FIREWALL_EXCEPTION
When set to False, the *condor_master* will not automatically add HTCondor to the Windows Firewall list of trusted applications. Such trusted applications can accept incoming connections without interference from the firewall. This only affects machines running Windows XP SP2 or higher. The default is True.

WINDOWS_FIREWALL_FAILURE_RETRY
An integer value (default value is 2) that represents the number of times the *condor_master* will retry to add firewall exceptions. When a Windows machine boots up, HTCondor starts up by default as well. Under certain conditions, the *condor_master* may have difficulty adding exceptions to the Windows Firewall because of a delay in other services starting up. Examples of services that may possibly be slow are the SharedAccess service, the Netman service, or the Workstation service. This configuration variable allows administrators to set the number of times (once every 5 seconds) that the *condor_master* will retry to add firewall exceptions. A value of 0 means that HTCondor will retry indefinitely.

USE_PROCESS_GROUPS
A boolean value that defaults to `True`. When `False`, HTCondor daemons on Unix machines will not create new sessions or process groups. HTCondor uses processes groups to help it track the descendants of processes it creates. This can cause problems when HTCondor is run under another job execution system.

DISCARD_SESSION_KEYRING_ON_STARTUP
A boolean value that defaults to `True`. When `True`, the *condor_master* daemon will replace the kernel session keyring it was invoked with with a new keyring named `htcondor`. Various Linux system services, such as OpenAFS and eCryptFS, use the kernel session keyring to hold passwords and authentication tokens. By replacing the keyring on start up, the *condor_master* ensures these keys cannot be unintentionally obtained by user jobs.

ENABLE_KERNEL_TUNING
Relevant only to Linux platforms, a boolean value that defaults to `True`. When `True`, the *condor_master* daemon invokes the kernel tuning script specified by configuration variable `LINUX_KERNEL_TUNING_SCRIPT` once as root when the *condor_master* daemon starts up.

KERNEL_TUNING_LOG
A string value that defaults to `$(LOG)/KernelTuningLog`. If the kernel tuning script runs, its output will be logged to this file.

LINUX_KERNEL_TUNING_SCRIPT
A string value that defaults to `$(LIBEXEC)/linux_kernel_tuning`. This is the script that the *condor_master* runs to tune the kernel when `ENABLE_KERNEL_TUNING` is `True`.

5.5.7 condor_startd Configuration File Macros

Note: If you are running HTCondor on a multi-CPU machine, be sure to also read *condor_startd Policy Configuration* which describes how to set up and configure HTCondor on multi-core machines.

These settings control general operation of the *condor_startd*. Examples using these configuration macros, as well as further explanation is found in the *Policy Configuration for Execution Points and for Access Points* section.

START
A boolean expression that, when `True`, indicates that the machine is willing to start running an HTCondor job. `START` is considered when the *condor_negotiator* daemon is considering evicting the job to replace it with one that will generate a better rank for the *condor_startd* daemon, or a user with a higher priority.

DEFAULT_DRAINING_START_EXPR
An alternate `START` expression to use while draining when the drain command is sent without a `-start` argument. When this configuration parameter is not set and the drain command does not specify a `-start` argument, `START` will have the value `undefined` and `Requirements` will be `false` while draining. This will prevent new jobs from matching. To allow evictable jobs to match while draining, set this to an expression that matches only those jobs.

SUSPEND
A boolean expression that, when `True`, causes HTCondor to suspend running an HTCondor job. The machine may still be claimed, but the job makes no further progress, and HTCondor does not generate a load on the machine.

PREEMPT
A boolean expression that, when `True`, causes HTCondor to stop a currently running job once `MAXJOBRETIREMENTTIME` has expired. This expression is not evaluated if `WANT_SUSPEND` is `True`. The default value is `False`, such that preemption is disabled.

WANT_HOLD
A boolean expression that defaults to `False`. When `True` and the value of `PREEMPT` becomes `True` and `WANT_SUSPEND` is `False` and `MAXJOBRETIREMENTTIME` has expired, the job is put on hold for the reason (optionally) specified by the variables `WANT_HOLD_REASON` and `WANT_HOLD_SUBCODE`. As usual, the job owner may specify **periodic_release** and/or **periodic_remove** expressions to react to specific hold states automatically. The attribute `HoldReasonCode` in the job ClassAd is set to the value 21 when `WANT_HOLD` is responsible for putting the job on hold.

Here is an example policy that puts jobs on hold that use too much virtual memory:

```
VIRTUAL_MEMORY_AVAILABLE_MB = (VirtualMemory*0.9)
MEMORY_EXCEEDED = ImageSize/1024 > $(VIRTUAL_MEMORY_AVAILABLE_MB)
PREEMPT = ($(PREEMPT)) || ($(MEMORY_EXCEEDED))
WANT_SUSPEND = ($(WANT_SUSPEND)) && ($(MEMORY_EXCEEDED)) =!= TRUE
WANT_HOLD = ($(MEMORY_EXCEEDED))
WANT_HOLD_REASON = \
   ifThenElse( $(MEMORY_EXCEEDED), \
               "Your job used too much virtual memory.", \
               undefined )
```

WANT_HOLD_REASON
An expression that defines a string utilized to set the job ClassAd attribute `HoldReason` when a job is put on hold due to `WANT_HOLD`. If not defined or if the expression evaluates to `Undefined`, a default hold reason is provided.

WANT_HOLD_SUBCODE
An expression that defines an integer value utilized to set the job ClassAd attribute `HoldReasonSubCode` when a job is put on hold due to `WANT_HOLD`. If not defined or if the expression evaluates to `Undefined`, the value is set to 0. Note that `HoldReasonCode` is always set to 21.

CONTINUE
A boolean expression that, when `True`, causes HTCondor to continue the execution of a suspended job.

KILL
A boolean expression that, when `True`, causes HTCondor to immediately stop the execution of a vacating job, without delay. The job is hard-killed, so any attempt by the job to clean up will be aborted. This expression should normally be `False`. When desired, it may be used to abort the graceful shutdown of a job earlier than the limit imposed by `MachineMaxVacateTime` .

PERIODIC_CHECKPOINT
A boolean expression that, when `True`, causes HTCondor to initiate a checkpoint of the currently running job. This setting applies to vm universe jobs that have set **vm_checkpoint** to `True` in the submit description file.

RANK
A floating point value that HTCondor uses to compare potential jobs. A larger value for a specific job ranks that job above others with lower values for `RANK`.

ADVERTISE_PSLOT_ROLLUP_INFORMATION
A boolean value that defaults to `True`, causing the *condor_startd* to advertise ClassAd attributes that may be used in partitionable slot preemption. The attributes are

- ChildAccountingGroup
- ChildActivity
- ChildCPUs
- ChildCurrentRank
- ChildEnteredCurrentState
- ChildMemory
- ChildName
- ChildRemoteOwner
- ChildRemoteUser
- ChildRetirementTimeRemaining
- ChildState
- PslotRollupInformation

STARTD_PARTITIONABLE_SLOT_ATTRS
A list of additional from the above default attributes from dynamic slots that will be rolled up into a list attribute in their parent partitionable slot, prefixed with the name Child.

WANT_SUSPEND
A boolean expression that, when `True`, tells HTCondor to evaluate the `SUSPEND` expression to decide whether to suspend a running job. When `True`, the `PREEMPT` expression is not evaluated. When not explicitly set, the *condor_startd* exits with an error. When explicitly set, but the evaluated value is anything other than `True`, the value is utilized as if it were `False`.

WANT_VACATE
A boolean expression that, when `True`, defines that a preempted HTCondor job is to be vacated, instead of killed. This means the job will be soft-killed and given time to clean up. The amount of time given depends on `MachineMaxVacateTime` and `KILL` . The default value is `True`.

IS_OWNER
A boolean expression that determines when a machine ad should enter the `Owner` state. While in the `Owner` state, the machine ad will not be matched to any jobs. The default value is `False` (never enter `Owner` state). Job ClassAd attributes should not be used in defining `IS_OWNER`, as they would be `Undefined`.

STARTD_HISTORY
A file name where the *condor_startd* daemon will maintain a job history file in an analogous way to that of the history file defined by the configuration variable `HISTORY`. It will be rotated in the same way, and the same parameters that apply to the `HISTORY` file rotation apply to the *condor_startd* daemon history as well. This can be read with the *condor_history* command by passing the name of the file to the -file option of *condor_history*.

```
$ condor_history -file `condor_config_val LOG`/startd_history
```

STARTER
This macro holds the full path to the *condor_starter* binary that the *condor_startd* should spawn. It is normally defined relative to `$(SBIN)`.

KILLING_TIMEOUT
The amount of time in seconds that the *condor_startd* should wait after sending a fast shutdown request to *condor_starter* before forcibly killing the job and *condor_starter*. The default value is 30 seconds.

POLLING_INTERVAL
When a *condor_startd* enters the claimed state, this macro determines how often the state of the machine is

polled to check the need to suspend, resume, vacate or kill the job. It is defined in terms of seconds and defaults to 5.

UPDATE_INTERVAL
Determines how often the *condor_startd* should send a ClassAd update to the *condor_collector*. The *condor_startd* also sends update on any state or activity change, or if the value of its START expression changes. See *condor_startd Policy Configuration* on *condor_startd* states, *condor_startd* Activities, and *condor_startd* START expression for details on states, activities, and the START expression. This macro is defined in terms of seconds and defaults to 300 (5 minutes).

UPDATE_OFFSET
An integer value representing the number of seconds of delay that the *condor_startd* should wait before sending its initial update, and the first update after a *condor_reconfig* command is sent to the *condor_collector*. The time of all other updates sent after this initial update is determined by $(UPDATE_INTERVAL). Thus, the first update will be sent after $(UPDATE_OFFSET) seconds, and the second update will be sent after $(UPDATE_OFFSET) + $(UPDATE_INTERVAL). This is useful when used in conjunction with the $RANDOM_INTEGER() macro for large pools, to spread out the updates sent by a large number of *condor_startd* daemons. Defaults to zero. The example configuration

```
startd.UPDATE_INTERVAL = 300
startd.UPDATE_OFFSET   = $RANDOM_INTEGER(0,300)
```

causes the initial update to occur at a random number of seconds falling between 0 and 300, with all further updates occurring at fixed 300 second intervals following the initial update.

MachineMaxVacateTime
An integer expression representing the number of seconds the machine is willing to wait for a job that has been soft-killed to gracefully shut down. The default value is 600 seconds (10 minutes). This expression is evaluated when the job starts running. The job may adjust the wait time by setting JobMaxVacateTime. If the job's setting is less than the machine's, the job's specification is used. If the job's setting is larger than the machine's, the result depends on whether the job has any excess retirement time. If the job has more retirement time left than the machine's maximum vacate time setting, then retirement time will be converted into vacating time, up to the amount of JobMaxVacateTime. The KILL expression may be used to abort the graceful shutdown of the job at any time. At the time when the job is preempted, the WANT_VACATE expression may be used to skip the graceful shutdown of the job.

MAXJOBRETIREMENTTIME
When the *condor_startd* wants to evict a job, a job which has run for less than the number of seconds specified by this expression will not be hard-killed. The *condor_startd* will wait for the job to finish or to exceed this amount of time, whichever comes sooner. Time spent in suspension does not count against the job. The default value of 0 (when the configuration variable is not present) means that the job gets no retirement time. If the job vacating policy grants the job X seconds of vacating time, a preempted job will be soft-killed X seconds before the end of its retirement time, so that hard-killing of the job will not happen until the end of the retirement time if the job does not finish shutting down before then. Note that in peaceful shutdown mode of the *condor_startd*, retirement time is treated as though infinite. In graceful shutdown mode, the job will not be preempted until the configured retirement time expires or SHUTDOWN_GRACEFUL_TIMEOUT expires. In fast shutdown mode, retirement time is ignored. See MAXJOBRETIREMENTTIME in *condor_startd Policy Configuration* for further explanation.

By default the *condor_negotiator* will not match jobs to a slot with retirement time remaining. This behavior is controlled by NEGOTIATOR_CONSIDER_EARLY_PREEMPTION .

There is no default value for this configuration variable.

CLAIM_WORKLIFE
This expression specifies the number of seconds after which a claim will stop accepting additional jobs. The default is 1200, which is 20 minutes. Once the *condor_negotiator* gives a *condor_schedd* a claim to a slot, the *condor_schedd* will keep running jobs on that slot as long as it has more jobs with matching requirements, and CLAIM_WORKLIFE has not expired, and it is not preempted. Once CLAIM_WORKLIFE expires, any existing job

may continue to run as usual, but once it finishes or is preempted, the claim is closed. When `CLAIM_WORKLIFE` is -1, this is treated as an infinite claim work life, so claims may be held indefinitely (as long as they are not preempted and the user does not run out of jobs, of course). A value of 0 has the effect of not allowing more than one job to run per claim, since it immediately expires after the first job starts running.

MAX_CLAIM_ALIVES_MISSED
The *condor_schedd* sends periodic updates to each *condor_startd* as a keep alive (see the description of . If the *condor_startd* does not receive any keep alive messages, it assumes that something has gone wrong with the *condor_schedd* and that the resource is not being effectively used. Once this happens, the *condor_startd* considers the claim to have timed out, it releases the claim, and starts advertising itself as available for other jobs. Because these keep alive messages are sent via UDP, they are sometimes dropped by the network. Therefore, the *condor_startd* has some tolerance for missed keep alive messages, so that in case a few keep alives are lost, the *condor_startd* will not immediately release the claim. This setting controls how many keep alive messages can be missed before the *condor_startd* considers the claim no longer valid. The default is 6.

STARTD_HAS_BAD_UTMP
When the *condor_startd* is computing the idle time of all the users of the machine (both local and remote), it checks the `utmp` file to find all the currently active ttys, and only checks access time of the devices associated with active logins. Unfortunately, on some systems, `utmp` is unreliable, and the *condor_startd* might miss keyboard activity by doing this. So, if your `utmp` is unreliable, set this macro to `True` and the *condor_startd* will check the access time on all tty and pty devices.

CONSOLE_DEVICES
This macro allows the *condor_startd* to monitor console (keyboard and mouse) activity by checking the access times on special files in `/dev`. Activity on these files shows up as `ConsoleIdle` time in the *condor_startd* 's ClassAd. Give a comma-separated list of the names of devices considered the console, without the `/dev/` portion of the path name. The defaults vary from platform to platform, and are usually correct.

One possible exception to this is on Linux, where we use "mouse" as one of the entries. Most Linux installations put in a soft link from `/dev/mouse` that points to the appropriate device (for example, `/dev/psaux` for a PS/2 bus mouse, or `/dev/tty00` for a serial mouse connected to com1). However, if your installation does not have this soft link, you will either need to put it in (you will be glad you did), or change this macro to point to the right device.

Unfortunately, modern versions of Linux do not update the access time of device files for USB devices. Thus, these files cannot be be used to determine when the console is in use. Instead, use the *condor_kbdd* daemon, which gets this information by connecting to the X server.

KBDD_BUMP_CHECK_SIZE
The number of pixels that the mouse can move in the X and/or Y direction, while still being considered a bump, and not keyboard activity. If the movement is greater than this bump size then the move is not a transient one, and it will register as activity. The default is 16, and units are pixels. Setting the value to 0 effectively disables bump testing.

KBDD_BUMP_CHECK_AFTER_IDLE_TIME
The number of seconds of keyboard idle time that will pass before bump testing begins. The default is 15 minutes.

STARTD_JOB_ATTRS
When the machine is claimed by a remote user, the *condor_startd* can also advertise arbitrary attributes from the job ClassAd in the machine ClassAd. List the attribute names to be advertised.

Note: Since these are already ClassAd expressions, do not do anything unusual with strings. By default, the job ClassAd attributes JobUniverse, NiceUser, ExecutableSize and ImageSize are advertised into the machine ClassAd.

STARTD_LATCH_EXPRS
Each time a slot is created, activated, or when periodic STARTD policy is evaluated HTCondor will evaluate

expressions whose names are listed in this configuration variable. If the evaluated value can be converted to an integer, and the value of the integer changes, the time of the change will be published.

This macro should be a list of the names of configuration variables that contain an expression to be evaluated, the name of the configuration variable will be treated as the base name of attributes published for the macro. Thus expressions listed behave like with the additional behavior the most recent evaluated value will be advertised as <name>Value and the time the value changed will be advertised as <name>Time. Entries in this list can also be the names of standard slot attributes like NumDynamicSlots, in which case the change time will be advertised but the evaluated value will not be advertised, since that would be redundant.

It is not an error when the result of evaluation is undefined, in that case the STARTD will remember the time that the value became undefined but not advertise the time. If the evaluated value becomes defined again, the time that it changed from undefined to the new value will again be advertised.

Example:

```
STARTD_LATCH_EXPRS = HalfFull NumDynamicSlots
HalfFull = Cpus < (TotalSlotCPUs/2) || Memory < (TotalSlotMemory/2)
```

For the configuration fragment above, the STARTD will advertise HalfFull as an expression, along with the last evaluated value of that expression as HalfFullValue, and the time it changed to that value as HalfFullTime. It will also advertise the time that the number of dynamic slots changed to its current value as NumDynamicSlotsTime. It will not advertise a NumDynamicSlotsValue because the <name>Value attribute is only advertised if <name> is an expression in the configuration that is not simple literal value.

STARTD_ATTRS
This macro is described in .

STARTD_DEBUG
This macro (and other settings related to debug logging in the *condor_startd*) is described in .

STARTD_ADDRESS_FILE
This macro is described in

STARTD_SHOULD_WRITE_CLAIM_ID_FILE
The *condor_startd* can be configured to write out the ClaimId for the next available claim on all slots to separate files. This boolean attribute controls whether the *condor_startd* should write these files. The default value is True.

STARTD_CLAIM_ID_FILE
This macro controls what file names are used if the above STARTD_SHOULD_WRITE_CLAIM_ID_FILE is true. By default, HTCondor will write the ClaimId into a file in the $(LOG) directory called .startd_claim_id.slotX, where X is the value of SlotID, the integer that identifies a given slot on the system, or 1 on a single-slot machine. If you define your own value for this setting, you should provide a full path, and HTCondor will automatically append the .slotX portion of the file name.

STARTD_PRINT_ADS_ON_SHUTDOWN
The *condor_startd* can be configured to write out the slot ads into the daemon's log file as it is shutting down. This is a boolean and the default value is False.

STARTD_PRINT_ADS_FILTER
When STARTD_PRINT_ADS_ON_SHUTDOWN above is set to True, this macro can list which specific types of ads will get written to the log. The possible values are static`, partitionable, and dynamic. The list is comma separated and the default is to print all three types of ads.

NUM_CPUS
An integer value, which can be used to lie to the *condor_startd* daemon about how many CPUs a machine has. When set, it overrides the value determined with HTCondor's automatic computation of the number of CPUs in the machine. Lying in this way can allow multiple HTCondor jobs to run on a single-CPU machine, by having that machine treated like a multi-core machine with multiple CPUs, which could have different HTCondor jobs

running on each one. Or, a multi-core machine may advertise more slots than it has CPUs. However, lying in this manner will hurt the performance of the jobs, since now multiple jobs will run on the same CPU, and the jobs will compete with each other. The option is only meant for people who specifically want this behavior and know what they are doing. It is disabled by default.

The default value is `$(DETECTED_CPUS_LIMIT)` .

The *condor_startd* only takes note of the value of this configuration variable on start up, therefore it cannot be changed with a simple reconfigure. To change this, restart the *condor_startd* daemon for the change to take effect. The command will be

```
$ condor_restart -startd
```

MAX_NUM_CPUS
An integer value used as a ceiling for the number of CPUs detected by HTCondor on a machine. This value is ignored if `NUM_CPUS` is set. If set to zero, there is no ceiling. If not defined, the default value is zero, and thus there is no ceiling.

Note that this setting cannot be changed with a simple reconfigure, either by sending a SIGHUP or by using the *condor_reconfig* command. To change this, restart the *condor_startd* daemon for the change to take effect. The command will be

```
$ condor_restart -startd
```

COUNT_HYPERTHREAD_CPUS
This configuration variable controls how HTCondor sees hyper-threaded processors. When set to the default value of `True`, it includes virtual CPUs in the default value of `DETECTED_CPUS`. On dedicated cluster nodes, counting virtual CPUs can sometimes improve total throughput at the expense of individual job speed. However, counting them on desktop workstations can interfere with interactive job performance.

MEMORY
Normally, HTCondor will automatically detect the amount of physical memory available on your machine. Define `MEMORY` to tell HTCondor how much physical memory (in MB) your machine has, overriding the value HTCondor computes automatically. The actual amount of memory detected by HTCondor is always available in the pre-defined configuration macro `DETECTED_MEMORY` .

RESERVED_MEMORY
How much memory would you like reserved from HTCondor? By default, HTCondor considers all the physical memory of your machine as available to be used by HTCondor jobs. If `RESERVED_MEMORY` is defined, HTCondor subtracts it from the amount of memory it advertises as available.

STARTD_NAME
Used to give an alternative value to the `Name` attribute in the *condor_startd* 's ClassAd. This esoteric configuration macro might be used in the situation where there are two *condor_startd* daemons running on one machine, and each reports to the same *condor_collector*. Different names will distinguish the two daemons. See the description of for defaults and composition of valid HTCondor daemon names.

RUNBENCHMARKS
A boolean expression that specifies whether to run benchmarks. When the machine is in the Unclaimed state and this expression evaluates to `True`, benchmarks will be run. If `RUNBENCHMARKS` is specified and set to anything other than `False`, additional benchmarks will be run once, when the *condor_startd* starts. To disable start up benchmarks, set `RunBenchmarks` to `False`.

DedicatedScheduler
A string that identifies the dedicated scheduler this machine is managed by. *HTCondor's Dedicated Scheduling* details the use of a dedicated scheduler.

STARTD_NOCLAIM_SHUTDOWN
The number of seconds to run without receiving a claim before shutting HTCondor down on this machine.

Defaults to unset, which means to never shut down. This is primarily intended to facilitate glidein; use in other situations is not recommended.

STARTD_PUBLISH_WINREG

A string containing a semicolon-separated list of Windows registry key names. For each registry key, the contents of the registry key are published in the machine ClassAd. All attribute names are prefixed with `WINREG_`. The remainder of the attribute name is formed in one of two ways. The first way explicitly specifies the name within the list with the syntax

```
STARTD_PUBLISH_WINREG = AttrName1 = KeyName1; AttrName2 = KeyName2
```

The second way of forming the attribute name derives the attribute names from the key names in the list. The derivation uses the last three path elements in the key name and changes each illegal character to an underscore character. Illegal characters are essentially any non-alphanumeric character. In addition, the percent character (%) is replaced by the string `Percent`, and the string `/sec` is replaced by the string `_Per_Sec`.

HTCondor expects that the hive identifier, which is the first element in the full path given by a key name, will be the valid abbreviation. Here is a list of abbreviations:

- `HKLM` is the abbreviation for `HKEY_LOCAL_MACHINE`
- `HKCR` is the abbreviation for `HKEY_CLASSES_ROOT`
- `HKCU` is the abbreviation for `HKEY_CURRENT_USER`
- `HKPD` is the abbreviation for `HKEY_PERFORMANCE_DATA`
- `HKCC` is the abbreviation for `HKEY_CURRENT_CONFIG`
- `HKU` is the abbreviation for `HKEY_USERS`

The `HKPD` key names are unusual, as they are not shown in *regedit*. Their values are periodically updated at the interval defined by `UPDATE_INTERVAL`. The others are not updated until *condor_reconfig* is issued.

Here is a complete example of the configuration variable definition,

```
STARTD_PUBLISH_WINREG = HKLM\Software\Perl\BinDir; \
 BATFile_RunAs_Command = HKCR\batFile\shell\RunAs\command; \
 HKPD\Memory\Available MBytes; \
 BytesAvail = HKPD\Memory\Available Bytes; \
 HKPD\Terminal Services\Total Sessions; \
 HKPD\Processor\% Idle Time; \
 HKPD\System\Processes
```

which generates the following portion of a machine ClassAd:

```
WINREG_Software_Perl_BinDir = "C:\Perl\bin\perl.exe"
WINREG_BATFile_RunAs_Command = "%SystemRoot%\System32\cmd.exe /C \"%1\" %*"
WINREG_Memory_Available_MBytes = 5331
WINREG_BytesAvail = 5590536192.000000
WINREG_Terminal_Services_Total_Sessions = 2
WINREG_Processor_Percent_Idle_Time = 72.350384
WINREG_System_Processes = 166
```

MOUNT_UNDER_SCRATCH

A ClassAd expression, which when evaluated in the context of the job ClassAd, evaluates to a string that contains a comma separated list of directories. For each directory in the list, HTCondor creates a directory in the job's temporary scratch directory with that name, and makes it available at the given name using bind mounts. This is available on Linux systems which provide bind mounts and per-process tree mount tables, such as Red Hat

Enterprise Linux 5. A bind mount is like a symbolic link, but is not globally visible to all processes. It is only visible to the job and the job's child processes. As an example:

```
MOUNT_UNDER_SCRATCH = ifThenElse(TARGET.UtsnameSysname ? "Linux", "/tmp,/var/tmp", "
↪")
```

If the job is running on a Linux system, it will see the usual `/tmp` and `/var/tmp` directories, but when accessing files via these paths, the system will redirect the access. The resultant files will actually end up in directories named `tmp` or `var/tmp` under the the job's temporary scratch directory. This is useful, because the job's scratch directory will be cleaned up after the job completes, two concurrent jobs will not interfere with each other, and because jobs will not be able to fill up the real `/tmp` directory. Another use case might be for home directories, which some jobs might want to write to, but that should be cleaned up after each job run. The default value is `"/tmp,/var/tmp"`.

If the job's execute directory is encrypted, `/tmp` and `/var/tmp` are automatically added to `MOUNT_UNDER_SCRATCH` when the job is run (they will not show up if `MOUNT_UNDER_SCRATCH` is examined with *condor_config_val*).

Note: The MOUNT_UNDER_SCRATCH mounts do not take place until the PreCmd of the job, if any, completes. (See *Job ClassAd Attributes* for information on PreCmd.)

Also note that, if `MOUNT_UNDER_SCRATCH` is defined, it must either be a ClassAd string (with double-quotes) or an expression that evaluates to a string.

For Docker Universe jobs, any directories that are mounted under scratch are also volume mounted on the same paths inside the container. That is, any reads or writes to files in those directories goes to the host filesystem under the scratch directory. This is useful if a container has limited space to grow a filesystem.

MOUNT_PRIVATE_DEV_SHM
This boolean value, which defaults to `True` tells the *condor_starter* to make /dev/shm on Linux private to each job. When private, the starter removes any files from the private /dev/shm at job exit time.

Warning: The per job filesystem feature is a work in progress and not currently supported.

The following macros control if the *condor_startd* daemon should create a custom filesystem for the job's scratch directory. This allows HTCondor to prevent the job from using more scratch space than provisioned.

STARTD_ENFORCE_DISK_LIMITS
This boolean value, which is only evaluated on Linux systems, tells the *condor_startd* whether to make an ephemeral filesystem for the scratch execute directory for jobs. The default is `False`. This should only be set to true on HTCondor installations that have root privilege. When `true`, you must set and , or alternatively set .

THINPOOL_NAME
This string-valued parameter has no default, and should be set to the Linux LVM logical volume to be used for ephemeral execute directories. `"htcondor_lv"` might be a good choice. This setting only matters when is `True`, and HTCondor has root privilege.

THINPOOL_VOLUME_GROUP_NAME
This string-valued parameter has no default, and should be set to the name of the Linux LVM volume group to be used for logical volumes for ephemeral execute directories. `"htcondor_vg"` might be a good choice. This setting only matters when is True, and HTCondor has root privilege.

THINPOOL_BACKING_FILE
This string-valued parameter has no default. If a rootly HTCondor does not have a Linux LVM configured, a

single large file can be used as the backing store for ephemeral file systems for execute directories. This parameter should be set to the path of a large, pre-created file to hold the blocks these filesystems are created from.

THINPOOL_HIDE_MOUNT
A boolean value that defaults to `false`. When thinpool ephemeral filesystems are enabled (as described above), if this knob is set to `true`, the mount will only be visible to the job and the starter. Any process in any other process tree will not be able to see the mount. Setting this to true breaks Docker universe.

The following macros control if the *condor_startd* daemon should perform backfill computations whenever resources would otherwise be idle. See *Configuring HTCondor for Running Backfill Jobs* for details.

ENABLE_BACKFILL
A boolean value that, when `True`, indicates that the machine is willing to perform backfill computations when it would otherwise be idle. This is not a policy expression that is evaluated, it is a simple `True` or `False`. This setting controls if any of the other backfill-related expressions should be evaluated. The default is `False`.

BACKFILL_SYSTEM
A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported value for this is `"BOINC"`, which stands for the Berkeley Open Infrastructure for Network Computing. See http://boinc.berkeley.edu for more information about BOINC. There is no default value, administrators must define this.

START_BACKFILL
A boolean expression that is evaluated whenever an HTCondor resource is in the Unclaimed/Idle state and the `ENABLE_BACKFILL` expression is `True`. If `START_BACKFILL` evaluates to `True`, the machine will enter the Backfill state and attempt to spawn a backfill computation. This expression is analogous to the `START` expression that controls when an HTCondor resource is available to run normal HTCondor jobs. The default value is `False` (which means do not spawn a backfill job even if the machine is idle and `ENABLE_BACKFILL` expression is `True`). For more information about policy expressions and the Backfill state, see *Policy Configuration for Execution Points and for Access Points*, especially the *condor_startd Policy Configuration* section.

EVICT_BACKFILL
A boolean expression that is evaluated whenever an HTCondor resource is in the Backfill state which, when `True`, indicates the machine should immediately kill the currently running backfill computation and return to the Owner state. This expression is a way for administrators to define a policy where interactive users on a machine will cause backfill jobs to be removed. The default value is `False`. For more information about policy expressions and the Backfill state, see *Policy Configuration for Execution Points and for Access Points*, especially the *condor_startd Policy Configuration* section.

The following macros only apply to the *condor_startd* daemon when it is running on a multi-core machine. See the *condor_startd Policy Configuration* section for details.

STARTD_RESOURCE_PREFIX
A string which specifies what prefix to give the unique HTCondor resources that are advertised on multi-core machines. Previously, HTCondor used the term virtual machine to describe these resources, so the default value for this setting was `vm`. However, to avoid confusion with other kinds of virtual machines, such as the ones created using tools like VMware or Xen, the old virtual machine terminology has been changed, and has become the term slot. Therefore, the default value of this prefix is now `slot`. If sites want to continue using `vm`, or prefer something other `slot`, this setting enables sites to define what string the *condor_startd* will use to name the individual resources on a multi-core machine.

SLOTS_CONNECTED_TO_CONSOLE
An integer which indicates how many of the machine slots the *condor_startd* is representing should be "connected" to the console. This allows the *condor_startd* to notice console activity. Defaults to the number of slots in the machine, which is `$(NUM_CPUS)`.

SLOTS_CONNECTED_TO_KEYBOARD
An integer which indicates how many of the machine slots the *condor_startd* is representing should be "connected" to the keyboard (for remote tty activity, as well as console activity). This defaults to all slots (N in a

machine with N CPUs).

DISCONNECTED_KEYBOARD_IDLE_BOOST
If there are slots not connected to either the keyboard or the console, the corresponding idle time reported will be the time since the *condor_startd* was spawned, plus the value of this macro. It defaults to 1200 seconds (20 minutes). We do this because if the slot is configured not to care about keyboard activity, we want it to be available to HTCondor jobs as soon as the *condor_startd* starts up, instead of having to wait for 15 minutes or more (which is the default time a machine must be idle before HTCondor will start a job). If you do not want this boost, set the value to 0. If you change your START expression to require more than 15 minutes before a job starts, but you still want jobs to start right away on some of your multi-core nodes, increase this macro's value.

STARTD_SLOT_ATTRS
The list of ClassAd attribute names that should be shared across all slots on the same machine. This setting was formerly know as `STARTD_VM_ATTRS` For each attribute in the list, the attribute's value is taken from each slot's machine ClassAd and placed into the machine ClassAd of all the other slots within the machine. For example, if the configuration file for a 2-slot machine contains

```
STARTD_SLOT_ATTRS = State, Activity, EnteredCurrentActivity
```

then the machine ClassAd for both slots will contain attributes that will be of the form:

```
slot1_State = "Claimed"
slot1_Activity = "Busy"
slot1_EnteredCurrentActivity = 1075249233
slot2_State = "Unclaimed"
slot2_Activity = "Idle"
slot2_EnteredCurrentActivity = 1075240035
```

The following settings control the number of slots reported for a given multi-core host, and what attributes each one has. They are only needed if you do not want to have a multi-core machine report to HTCondor with a separate slot for each CPU, with all shared system resources evenly divided among them. Please read *condor_startd Policy Configuration* for details on how to properly configure these settings to suit your needs.

Note: You cannot change the number or definition of the different slot types with a reconfig. If you change anything related to slot provisioning, you must restart the *condor_startd* for the change to take effect (for example, using `condor_restart -startd`).

Note: Prior to version 6.9.3, any settings that included the term `slot` used to use virtual machine or `vm`. If searching for information about one of these older settings, search for the corresponding attribute names using `slot`, instead.

MAX_SLOT_TYPES
The maximum number of different slot types. Note: this is the maximum number of different types, not of actual slots. Defaults to 10. (You should only need to change this setting if you define more than 10 separate slot types, which would be pretty rare.)

SLOT_TYPE_<N>
This setting defines a given slot type, by specifying what part of each shared system resource (like RAM, swap space, etc) this kind of slot gets. This setting has no effect unless you also define `NUM_SLOTS_TYPE_<N>`. N can be any integer from 1 to the value of `$(MAX_SLOT_TYPES)`, such as `SLOT_TYPE_1`. The format of this entry can be somewhat complex, so please refer to *condor_startd Policy Configuration* for details on the different possibilities.

SLOT_TYPE_<N>_PARTITIONABLE
A boolean variable that defaults to `False`. When `True`, this slot permits dynamic provisioning, as specified in

condor_startd Policy Configuration.

CLAIM_PARTITIONABLE_LEFTOVERS
A boolean variable that defaults to `True`. When `True` within the configuration for both the *condor_schedd* and the *condor_startd*, and the *condor_schedd* claims a partitionable slot, the *condor_startd* returns the slot's ClassAd and a claim id for leftover resources. In doing so, the *condor_schedd* can claim multiple dynamic slots without waiting for a negotiation cycle.

ENABLE_CLAIMABLE_PARTITIONABLE_SLOTS
A boolean variable that defaults to `False`. When set to `True` in the configuration of both the *condor_startd* and the *condor_schedd*, and the *condor_schedd* claims a partitionable slot, the partitionable slot's `State` will change to `Claimed` in addition to the creation of a `Claimed` dynamic slot. While the slot is `Claimed`, no other *condor_schedd* is able to create new dynamic slots to run jobs.

MAX_PARTITIONABLE_SLOT_CLAIM_TIME
An integer that indicates the maximum amount of time that a partitionable slot can be in the `Claimed` state before returning to the Unclaimed state, expressed in seconds. The default value is 3600.

MACHINE_RESOURCE_NAMES
A comma and/or space separated list of resource names that represent custom resources specific to a machine. These resources are further intended to be statically divided or partitioned, and these resource names identify the configuration variables that define the partitioning. If used, custom resources without names in the list are ignored.

MACHINE_RESOURCE_<name>
An integer that specifies the quantity of or list of identifiers for the customized local machine resource available for an SMP machine. The portion of this configuration variable's name identified with `<name>` will be used to label quantities of the resource allocated to a slot. If a quantity is specified, the resource is presumed to be fungible and slots will be allocated a quantity of the resource but specific instances will not be identified. If a list of identifiers is specified the quantity is the number of identifiers and slots will be allocated both a quantity of the resource and assigned specific resource identifiers.

OFFLINE_MACHINE_RESOURCE_<name>
A comma and/or space separated list of resource identifiers for any customized local machine resources that are currently offline, and therefore should not be allocated to a slot. The identifiers specified here must match those specified by value of configuration variables `MACHINE_RESOURCE_<name>` or `MACHINE_RESOURCE_INVENTORY_<name>` , or the identifiers will be ignored. The `<name>` identifies the type of resource, as specified by the value of configuration variable `MACHINE_RESOURCE_NAMES`. This configuration variable is used to have resources that are detected and reported to exist by HTCondor, but not assigned to slots. A restart of the *condor_startd* is required for changes to resources assigned to slots to take effect. If this variable is changed and *condor_reconfig* command is sent to the Startd, the list of Offline resources will be updated, and the count of resources of that type will be updated, but newly offline resources will still be assigned to slots. If an offline resource is assigned to a Partitionable slot, it will never be assigned to a new dynamic slot but it will not be removed from the `Assigned<name>` attribute of an existing dynamic slot.

MACHINE_RESOURCE_INVENTORY_<name>
Specifies a command line that is executed upon start up of the *condor_startd* daemon. The script is expected to output an attribute definition of the form

```
Detected<xxx>=y
```

or of the form

```
Detected<xxx>="y, z, a, ..."
```

where `<xxx>` is the name of a resource that exists on the machine, and `y` is the quantity of the resource or `"y, z, a, ..."` is a comma and/or space separated list of identifiers of the resource that exist on the machine. This attribute is added to the machine ClassAd, such that these resources may be statically divided or partitioned. A

script may be a convenient way to specify a calculated or detected quantity of the resource, instead of specifying a fixed quantity or list of the resource in the the configuration when set by MACHINE_RESOURCE_<name> .

The script may also output an attribute of the form

```
Offline<xxx>="y, z"
```

where <xxx> is the name of the resource, and "y, z" is a comma and/or space separated list of resource identifiers that are also in the Detected<xxx> list. This attribute is added to the machine ClassAd, and resources y and z will not be assigned to any slot and will not be included in the count of resources of this type. This will override the configuration variable OFFLINE_MACHINE_RESOURCE_<xxx> on startup. But OFFLINE_MACHINE_RESOURCE_<xxx> can still be used to take additional resources offline without restarting.

ENVIRONMENT_FOR_Assigned<name>
A space separated list of environment variables to set for the job. Each environment variable will be set to the list of assigned resources defined by the slot ClassAd attribute Assigned<name>. Each environment variable name may be followed by an equals sign and a Perl style regular expression that defines how to modify each resource ID before using it as the value of the environment variable. As a special case for CUDA GPUs, if the environment variable name is CUDA_VISIBLE_DEVICES, then the correct Perl style regular expression is applied automatically.

For example, with the configuration

```
ENVIRONMENT_FOR_AssignedGPUs = VISIBLE_GPUS=/^/gpuid:/
```

and with the machine ClassAd attribute AssignedGPUs = "CUDA1, CUDA2", the job's environment will contain

```
VISIBLE_GPUS = gpuid:CUDA1, gpuid:CUDA2
```

ENVIRONMENT_VALUE_FOR_UnAssigned<name>
Defines the value to set for environment variables specified in by configuration variable ENVIRONMENT_FOR_Assigned<name> when there is no machine ClassAd attribute Assigned<name> for the slot. This configuration variable exists to deal with the situation where jobs will use a resource that they have not been assigned because there is no explicit assignment. The CUDA runtime library (for GPUs) has this problem.

For example, where configuration is

```
ENVIRONMENT_FOR_AssignedGPUs = VISIBLE_GPUS
ENVIRONMENT_VALUE_FOR_UnAssignedGPUs = none
```

and there is no machine ClassAd attribute AssignedGPUs, the job's environment will contain

```
VISIBLE_GPUS = none
```

MUST_MODIFY_REQUEST_EXPRS
A boolean value that defaults to False. When False, configuration variables whose names begin with MODIFY_REQUEST_EXPR are only applied if the job claim still matches the partitionable slot after modification. If True, the modifications always take place, and if the modifications cause the claim to no longer match, then the *condor_startd* will simply refuse the claim.

MODIFY_REQUEST_EXPR_REQUESTMEMORY
An integer expression used by the *condor_startd* daemon to modify the evaluated value of the RequestMemory job ClassAd attribute, before it used to provision a dynamic slot. The default value is given by

```
quantize(RequestMemory,{128})
```

MODIFY_REQUEST_EXPR_REQUESTDISK
An integer expression used by the *condor_startd* daemon to modify the evaluated value of the `RequestDisk` job ClassAd attribute, before it used to provision a dynamic slot. The default value is given by

```
quantize(RequestDisk,{1024})
```

MODIFY_REQUEST_EXPR_REQUESTCPUS
An integer expression used by the *condor_startd* daemon to modify the evaluated value of the `RequestCpus` job ClassAd attribute, before it used to provision a dynamic slot. The default value is given by

```
quantize(RequestCpus,{1})
```

NUM_SLOTS_TYPE_<N>
This macro controls how many of a given slot type are actually reported to HTCondor. There is no default.

NUM_SLOTS
An integer value representing the number of slots reported when the multi-core machine is being evenly divided, and the slot type settings described above are not being used. The default is one slot for each CPU. This setting can be used to reserve some CPUs on a multi-core machine, which would not be reported to the HTCondor pool. This value cannot be used to make HTCondor advertise more slots than there are CPUs on the machine. To do that, use `NUM_CPUS` .

The following variables set consumption policies for partitionable slots. The *condor_startd Policy Configuration* section details consumption policies.

CONSUMPTION_POLICY
A boolean value that defaults to `False`. When `True`, consumption policies are enabled for partitionable slots within the *condor_startd* daemon. Any definition of the form `SLOT_TYPE_<N>_CONSUMPTION_POLICY` overrides this global definition for the given slot type.

CONSUMPTION_<Resource>
An expression that specifies a consumption policy for a particular resource within a partitionable slot. To support a consumption policy, each resource advertised by the slot must have such a policy configured. Custom resources may be specified, substituting the resource name for `<Resource>`. Any definition of the form `SLOT_TYPE_<N>_CONSUMPTION_<Resource>` overrides this global definition for the given slot type. CPUs, memory, and disk resources are always advertised by *condor_startd*, and have the default values:

```
CONSUMPTION_CPUS = quantize(target.RequestCpus,{1})
CONSUMPTION_MEMORY = quantize(target.RequestMemory,{128})
CONSUMPTION_DISK = quantize(target.RequestDisk,{1024})
```

Custom resources have no default consumption policy.

SLOT_WEIGHT
An expression that specifies a slot's weight, used as a multiplier the *condor_negotiator* daemon during matchmaking to assess user usage of a slot, which affects user priority. Defaults to `Cpus`.

In the case of slots with consumption policies, the cost of each match is is assessed as the difference in the slot weight expression before and after the resources consumed by the match are deducted from the slot. Only Memory, Cpus and Disk are valid attributes for this parameter.

NUM_CLAIMS
Specifies the number of claims a partitionable slot will advertise for use by the *condor_negotiator* daemon. In the case of slots with a defined consumption policy, the *condor_negotiator* may match more than one job to the slot in a single negotiation cycle. For partitionable slots with a consumption policy, `NUM_CLAIMS` defaults to the number of CPUs owned by the slot. Otherwise, it defaults to 1.

The following configuration variables support java universe jobs.

JAVA
The full path to the Java interpreter (the Java Virtual Machine).

JAVA_CLASSPATH_ARGUMENT
The command line argument to the Java interpreter (the Java Virtual Machine) that specifies the Java Classpath. Classpath is a Java-specific term that denotes the list of locations (`.jar` files and/or directories) where the Java interpreter can look for the Java class files that a Java program requires.

JAVA_CLASSPATH_SEPARATOR
The single character used to delimit constructed entries in the Classpath for the given operating system and Java Virtual Machine. If not defined, the operating system is queried for its default Classpath separator.

JAVA_CLASSPATH_DEFAULT
A list of path names to `.jar` files to be added to the Java Classpath by default. The comma and/or space character delimits list entries.

JAVA_EXTRA_ARGUMENTS
A list of additional arguments to be passed to the Java executable.

The following configuration variables control .NET version advertisement.

STARTD_PUBLISH_DOTNET
A boolean value that controls the advertising of the .NET framework on Windows platforms. When `True`, the *condor_startd* will advertise all installed versions of the .NET framework within the `DotNetVersions` attribute in the *condor_startd* machine ClassAd. The default value is `True`. Set the value to `false` to turn off .NET version advertising.

DOT_NET_VERSIONS
A string expression that administrators can use to override the way that .NET versions are advertised. If the administrator wishes to advertise .NET installations, but wishes to do so in a format different than what the *condor_startd* publishes in its ClassAds, setting a string in this expression will result in the *condor_startd* publishing the string when `STARTD_PUBLISH_DOTNET` is `True`. No value is set by default.

These macros control the power management capabilities of the *condor_startd* to optionally put the machine in to a low power state and wake it up later. See *Power Management* for more details.

HIBERNATE_CHECK_INTERVAL
An integer number of seconds that determines how often the *condor_startd* checks to see if the machine is ready to enter a low power state. The default value is 0, which disables the check. If not 0, the `HIBERNATE` expression is evaluated within the context of each slot at the given interval. If used, a value 300 (5 minutes) is recommended.

As a special case, the interval is ignored when the machine has just returned from a low power state, excluding `"SHUTDOWN"`. In order to avoid machines from volleying between a running state and a low power state, an hour of uptime is enforced after a machine has been woken. After the hour has passed, regular checks resume.

HIBERNATE
A string expression that represents lower power state. When this state name evaluates to a valid state other than `"NONE"`, causes HTCondor to put the machine into the specified low power state. The following names are supported (and are not case sensitive):

- `"NONE"`, `"0"`: No-op; do not enter a low power state
- `"S1"`, `"1"`, `"STANDBY"`, `"SLEEP"`: On Windows, this is Sleep (standby)
- `"S2"`, `"2"`: On Windows, this is Sleep (standby)
- `"S3"`, `"3"`, `"RAM"`, `"MEM"`, `"SUSPEND"`: On Windows, this is Sleep (standby)
- `"S4"`, `"4"`, `"DISK"`, `"HIBERNATE"`: Hibernate
- `"S5"`, `"5"`, `"SHUTDOWN"`, `"OFF"`: Shutdown (soft-off)

The HIBERNATE expression is written in terms of the S-states as defined in the Advanced Configuration and Power Interface (ACPI) specification. The S-states take the form S<n>, where <n> is an integer in the range 0 to 5, inclusive. The number that results from evaluating the expression determines which S-state to enter. The notation was adopted because it appears to be the standard naming scheme for power states on several popular operating systems, including various flavors of Windows and Linux distributions. The other strings, such as "RAM" and "DISK", are provided for ease of configuration.

Since this expression is evaluated in the context of each slot on the machine, any one slot has veto power over the other slots. If the evaluation of HIBERNATE in one slot evaluates to "NONE" or "0", then the machine will not be placed into a low power state. On the other hand, if all slots evaluate to a non-zero value, but differ in value, then the largest value is used as the representative power state.

Strings that do not match any in the table above are treated as "NONE".

UNHIBERNATE
A boolean expression that specifies when an offline machine should be woken up. The default value is MachineLastMatchTime =!= UNDEFINED. This expression does not do anything, unless there is an instance of *condor_rooster* running, or another program that evaluates the Unhibernate expression of offline machine ClassAds. In addition, the collecting of offline machine ClassAds must be enabled for this expression to work. The variable explains this. The special attribute MachineLastMatchTime is updated in the ClassAds of offline machines when a job would have been matched to the machine if it had been online. For multi-slot machines, the offline ClassAd for slot1 will also contain the attributes slot<X>_MachineLastMatchTime, where X is replaced by the slot id of the other slots that would have been matched while offline. This allows the slot1 UNHIBERNATE expression to refer to all of the slots on the machine, in case that is necessary. By default, *condor_rooster* will wake up a machine if any slot on the machine has its UNHIBERNATE expression evaluate to True.

HIBERNATION_PLUGIN
A string which specifies the path and executable name of the hibernation plug-in that the *condor_startd* should use in the detection of low power states and switching to the low power states. The default value is $(LIBEXEC)/power_state. A default executable in that location which meets these specifications is shipped with HTCondor.

The *condor_startd* initially invokes this plug-in with both the value defined for HIBERNATION_PLUGIN_ARGS and the argument *ad*, and expects the plug-in to output a ClassAd to its standard output stream. The *condor_startd* will use this ClassAd to determine what low power setting to use on further invocations of the plug-in. To that end, the ClassAd must contain the attribute HibernationSupportedStates, a comma separated list of low power modes that are available. The recognized mode strings are the same as those in the table for the configuration variable HIBERNATE. The optional attribute HibernationMethod specifies a string which describes the mechanism used by the plug-in. The default Linux plug-in shipped with HTCondor will produce one of the strings NONE, /sys, /proc, or pm-utils. The optional attribute HibernationRawMask is an integer which represents the bit mask of the modes detected.

Subsequent *condor_startd* invocations of the plug-in have command line arguments defined by HIBERNATION_PLUGIN_ARGS plus the argument **set** <*power-mode*>, where <*power-mode*> is one of the supported states as given in the attribute HibernationSupportedStates.

HIBERNATION_PLUGIN_ARGS
Command line arguments appended to the command that invokes the plug-in. The additional argument *ad* is appended when the *condor_startd* initially invokes the plug-in.

HIBERNATION_OVERRIDE_WOL
A boolean value that defaults to False. When True, it causes the *condor_startd* daemon's detection of the whether or not the network interface handles WOL packets to be ignored. When False, hibernation is disabled if the network interface does not use WOL packets to wake from hibernation. Therefore, when True hibernation can be enabled despite the fact that WOL packets are not used to wake machines.

LINUX_HIBERNATION_METHOD
A string that can be used to override the default search used by HTCondor on Linux platforms to detect the hibernation method to use. This is used by the default hibernation plug-in executable that is shipped with HTCondor.

The default behavior orders its search with:

1. Detect and use the *pm-utils* command line tools. The corresponding string is defined with "pm-utils".
2. Detect and use the directory in the virtual file system `/sys/power`. The corresponding string is defined with "/sys".
3. Detect and use the directory in the virtual file system `/proc/ACPI`. The corresponding string is defined with "/proc".

To override this ordered search behavior, and force the use of one particular method, set `LINUX_HIBERNATION_METHOD` to one of the defined strings.

OFFLINE_EXPIRE_ADS_AFTER
An integer number of seconds specifying the lifetime of the persistent machine ClassAd representing a hibernating machine. Defaults to the largest 32-bit integer.

DOCKER
Defines the path and executable name of the Docker CLI. The default value is /usr/bin/docker. Remember that the condor user must also be in the docker group for Docker Universe to work. See the Docker universe manual section for more details (admin-manual/setting-up-vm-docker-universes:setting up the vm and docker universes). An example of the configuration for running the Docker CLI:

```
DOCKER = /usr/bin/docker
```

DOCKER_VOLUMES
A list of directories on the host execute machine to be volume mounted within the container. See the Docker Universe section for full details (admin-manual/setting-up-vm-docker-universes:setting up the vm and docker universes).

DOCKER_IMAGE_CACHE_SIZE
The number of most recently used Docker images that will be kept on the local machine. The default value is 8.

DOCKER_DROP_ALL_CAPABILITIES
A class ad expression, which defaults to true. Evaluated in the context of the job ad and the machine ad, when true, runs the Docker container with the command line option -drop-all-capabilities. Admins should be very careful with this setting, and only allow trusted users to run with full Linux capabilities within the container.

DOCKER_PERFORM_TEST
When the *condor_startd* starts up, it runs a simple Docker container to verify that Docker completely works. If DOCKER_PERFORM_TEST is false, this test is skipped.

DOCKER_RUN_UNDER_INIT
A boolean value which defaults to true, which tells the worker node to run Docker universe jobs with the –init option.

DOCKER_EXTRA_ARGUMENTS
Any additional command line options the administrator wants to be added to the Docker container create command line can be set with this parameter. Note that the admin should be careful setting this, it is intended for newer Docker options that HTCondor doesn't support directly. Arbitrary Docker options may break Docker universe, for example don't pass the –rm flag in DOCKER_EXTRA_ARGUMENTS, because then HTCondor cannot get the final exit status from a Docker job.

DOCKER_NETWORKS
An optional, comma-separated list of admin-defined networks that a job may request with the `docker_network_type` submit file command. Advertised into the slot attribute DockerNetworks.

DOCKER_SHM_SIZE
An optional knob that can be configured to adapt the `--shm-size` Docker container create argument. Allowed values are integers in bytes. If not set, `--shm-size` will not be specified by HTCondor and Docker's default is used. This is used to configure the size of the container's `/dev/shm` size adapting to the job's requested memory.

DOCKER_CACHE_ADVERTISE_INTERVAL
The *condor_startd* periodically advertises how much disk space the docker daemon is using to store images into the slot attribute DockerCachedImageSize. This knob, which defaults to 1200 (seconds), controls how often the start polls the docker daemon for this information.

OPENMPI_INSTALL_PATH
The location of the Open MPI installation on the local machine. Referenced by `examples/openmpiscript`, which is used for running Open MPI jobs in the parallel universe. The Open MPI bin and lib directories should exist under this path. The default value is `/usr/lib64/openmpi`.

OPENMPI_EXCLUDE_NETWORK_INTERFACES
A comma-delimited list of network interfaces that Open MPI should not use for MPI communications. Referenced by `examples/openmpiscript`, which is used for running Open MPI jobs in the parallel universe.

The list should contain any interfaces that your job could potentially see from any execute machine. The list may contain undefined interfaces without generating errors. Open MPI should exclusively use low latency/high speed networks it finds (e.g. InfiniBand) regardless of this setting. The default value is `docker0,virbr0`.

These macros control the startds (and starters) capability to create a private filesystem for the scratch directory for each job.

THINPOOL_VOLUME_GROUP_NAME
A string that names the Linux LVM volume group the administrator has configured as the storage for per-job scratch directories.

THINPOOL_NAME
A string that names the Linux LVM logical volume for storage for per-job scratch directories.

STARTD_ENFORCE_DISK_LIMITS
A boolean that defaults to false that controls whether the starter puts a job on hold that fills the per-job filesystem.

5.5.8 condor_schedd Configuration File Entries

These macros control the *condor_schedd*.

SHADOW
This macro determines the full path of the *condor_shadow* binary that the *condor_schedd* spawns. It is normally defined in terms of `$(SBIN)`.

START_LOCAL_UNIVERSE
A boolean value that defaults to `TotalLocalJobsRunning < 200`. The *condor_schedd* uses this macro to determine whether to start a **local** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **local** universe job that it has. For each job, if the `START_LOCAL_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **local** universe jobs to execute concurrently. The attribute `TotalLocalJobsRunning` is supplied by *condor_schedd* 's ClassAd:

```
START_LOCAL_UNIVERSE = TotalLocalJobsRunning < 10
```

STARTER_LOCAL
The complete path and executable name of the *condor_starter* to run for **local** universe jobs. This variable's value is defined in the initial configuration provided with HTCondor as

```
STARTER_LOCAL = $(SBIN)/condor_starter
```

This variable would only be modified or hand added into the configuration for a pool to be upgraded from one running a version of HTCondor that existed before the **local** universe to one that includes the **local** universe, but without utilizing the newer, provided configuration files.

LOCAL_UNIV_EXECUTE
A string value specifying the execute location for local universe jobs. Each running local universe job will receive a uniquely named subdirectory within this directory. If not specified, it defaults to `$(SPOOL)/local_univ_execute`.

START_SCHEDULER_UNIVERSE
A boolean value that defaults to `TotalSchedulerJobsRunning < 500`. The *condor_schedd* uses this macro to determine whether to start a **scheduler** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **scheduler** universe job that it has. For each job, if the `START_SCHEDULER_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **scheduler** universe jobs to execute concurrently. The attribute `TotalSchedulerJobsRunning` is supplied by *condor_schedd* 's ClassAd:

```
START_SCHEDULER_UNIVERSE = TotalSchedulerJobsRunning < 10
```

START_VANILLA_UNIVERSE
A boolean expression that defaults to nothing. When this macro is defined the *condor_schedd* uses it to determine whether to start a **vanilla** universe job. The *condor_schedd* uses the expression when matching a job with a slot in addition to the `Requirements` expression of the job and the slot ClassAds. The expression can refer to job attributes by using the prefix `JOB`, slot attributes by using the prefix `SLOT`, and job owner attributes by using the prefix `OWNER`.

The following example prevents jobs owned by a user from starting when that user has more than 25 held jobs

```
START_VANILLA_UNIVERSE = OWNER.JobsHeld <= 25
```

SCHEDD_USES_STARTD_FOR_LOCAL_UNIVERSE
A boolean value that defaults to false. When true, the *condor_schedd* will spawn a special startd process to run local universe jobs. This allows local universe jobs to run with both a condor_shadow and a condor_starter, which means that file transfer will work with local universe jobs.

MAX_JOBS_RUNNING
An integer representing a limit on the number of *condor_shadow* processes spawned by a given *condor_schedd* daemon, for all job universes except grid, scheduler, and local universe. Limiting the number of running scheduler and local universe jobs can be done using `START_LOCAL_UNIVERSE` and `START_SCHEDULER_UNIVERSE`. The actual number of allowed *condor_shadow* daemons may be reduced, if the amount of memory defined by `RESERVED_SWAP` limits the number of *condor_shadow* daemons. A value for `MAX_JOBS_RUNNING` that is less than or equal to 0 prevents any new job from starting. Changing this setting to be below the current number of jobs that are running will cause running jobs to be aborted until the number running is within the limit.

Like all integer configuration variables, `MAX_JOBS_RUNNING` may be a ClassAd expression that evaluates to an integer, and which refers to constants either directly or via macro substitution. The default value is an expression that depends on the total amount of memory and the operating system. The default expression requires 1MByte of RAM per running job on the access point. In some environments and configurations, this is overly generous and can be cut by as much as 50%. On Windows platforms, the number of running jobs is capped at 2000. A 64-bit version of Windows is recommended in order to raise the value above the default. Under Unix, the maximum default is now 10,000. To scale higher, we recommend that the system ephemeral port range is extended such that there are at least 2.1 ports per running job.

Here are example configurations:

```
## Example 1:
MAX_JOBS_RUNNING = 10000

## Example 2:
## This is more complicated, but it produces the same limit as the default.
## First define some expressions to use in our calculation.
## Assume we can use up to 80% of memory and estimate shadow private data
## size of 800k.
MAX_SHADOWS_MEM = ceiling($(DETECTED_MEMORY)*0.8*1024/800)
## Assume we can use ~21,000 ephemeral ports (avg ~2.1 per shadow).
## Under Linux, the range is set in /proc/sys/net/ipv4/ip_local_port_range.
MAX_SHADOWS_PORTS = 10000
## Under windows, things are much less scalable, currently.
## Note that this can probably be safely increased a bit under 64-bit windows.
MAX_SHADOWS_OPSYS = ifThenElse(regexp("WIN.*","$(OPSYS)"),2000,100000)
## Now build up the expression for MAX_JOBS_RUNNING.  This is complicated
## due to lack of a min() function.
MAX_JOBS_RUNNING = $(MAX_SHADOWS_MEM)
MAX_JOBS_RUNNING = \
  ifThenElse( $(MAX_SHADOWS_PORTS) < $(MAX_JOBS_RUNNING), \
              $(MAX_SHADOWS_PORTS), \
              $(MAX_JOBS_RUNNING) )
MAX_JOBS_RUNNING = \
  ifThenElse( $(MAX_SHADOWS_OPSYS) < $(MAX_JOBS_RUNNING), \
              $(MAX_SHADOWS_OPSYS), \
              $(MAX_JOBS_RUNNING) )
```

MAX_JOBS_SUBMITTED
This integer value limits the number of jobs permitted in a *condor_schedd* daemon's queue. Submission of a new cluster of jobs fails, if the total number of jobs would exceed this limit. The default value for this variable is the largest positive integer value.

MAX_JOBS_PER_OWNER
This integer value limits the number of jobs any given owner (user) is permitted to have within a *condor_schedd* daemon's queue. A job submission fails if it would cause this limit on the number of jobs to be exceeded. The default value is 100000.

This configuration variable may be most useful in conjunction with `MAX_JOBS_SUBMITTED`, to ensure that no one user can dominate the queue.

ALLOW_SUBMIT_FROM_KNOWN_USERS_ONLY
This boolean value determines if a User record will be created automatically when an unknown user submits a job. When true, only daemons or users that have a User record in the *condor_schedd* can submit jobs. When false, a User record will be added during job submission for users that have never submitted a job before. The default value is false which is consistent with the behavior of the *condor_schedd* before User records were added.

MAX_RUNNING_SCHEDULER_JOBS_PER_OWNER
This integer value limits the number of scheduler universe jobs that any given owner (user) can have running at one time. This limit will affect the number of running Dagman jobs, but not the number of nodes within a DAG. The default value is 200

MAX_JOBS_PER_SUBMISSION
This integer value limits the number of jobs any single submission is permitted to add to a *condor_schedd* daemon's queue. The whole submission fails if the number of jobs would exceed this limit. The default value is 20000.

This configuration variable may be useful for catching user error, and for protecting a busy *condor_schedd* daemon from the excessively lengthy interruption required to accept a very large number of jobs at one time.

MAX_SHADOW_EXCEPTIONS
This macro controls the maximum number of times that *condor_shadow* processes can have a fatal error (exception) before the *condor_schedd* will relinquish the match associated with the dying shadow. Defaults to 5.

MAX_PENDING_STARTD_CONTACTS
An integer value that limits the number of simultaneous connection attempts by the *condor_schedd* when it is requesting claims from one or more *condor_startd* daemons. The intention is to protect the *condor_schedd* from being overloaded by authentication operations. The default value is 0. The special value 0 indicates no limit.

CURB_MATCHMAKING
A ClassAd expression evaluated by the *condor_schedd* in the context of the *condor_schedd* daemon's own ClassAd. While this expression evaluates to `True`, the *condor_schedd* will refrain from requesting more resources from a *condor_negotiator*. Defaults to `RecentDaemonCoreDutyCycle > 0.98`.

MAX_CONCURRENT_DOWNLOADS
This specifies the maximum number of simultaneous transfers of output files from execute machines to the access point. The limit applies to all jobs submitted from the same *condor_schedd*. The default is 100. A setting of 0 means unlimited transfers. This limit currently does not apply to grid universe jobs, and it also does not apply to streaming output files. When the limit is reached, additional transfers will queue up and wait before proceeding.

MAX_CONCURRENT_UPLOADS
This specifies the maximum number of simultaneous transfers of input files from the access point to execute machines. The limit applies to all jobs submitted from the same *condor_schedd*. The default is 100. A setting of 0 means unlimited transfers. This limit currently does not apply to grid universe jobs. When the limit is reached, additional transfers will queue up and wait before proceeding.

FILE_TRANSFER_DISK_LOAD_THROTTLE
This configures throttling of file transfers based on the disk load generated by file transfers. The maximum number of concurrent file transfers is specified by `MAX_CONCURRENT_UPLOADS` and `MAX_CONCURRENT_DOWNLOADS` . Throttling will dynamically reduce the level of concurrency further to attempt to prevent disk load from exceeding the specified level. Disk load is computed as the average number of file transfer processes conducting read/write operations at the same time. The throttle may be specified as a single floating point number or as a range. Syntax for the range is the smaller number followed by 1 or more spaces or tabs, the string `"to"`, 1 or more spaces or tabs, and then the larger number. Example:

```
FILE_TRANSFER_DISK_LOAD_THROTTLE = 5 to 6.5
```

If only a single number is provided, this serves as the upper limit, and the lower limit is set to 90% of the upper limit. When the disk load is above the upper limit, no new transfers will be started. When between the lower and upper limits, new transfers will only be started to replace ones that finish. The default value is 2.0.

FILE_TRANSFER_DISK_LOAD_THROTTLE_WAIT_BETWEEN_INCREMENTS
This rarely configured variable sets the waiting period between increments to the concurrency level set by `FILE_TRANSFER_DISK_LOAD_THROTTLE`. The default is 1 minute. A value that is too short risks starting too many transfers before their effect on the disk load becomes apparent.

FILE_TRANSFER_DISK_LOAD_THROTTLE_SHORT_HORIZON
This rarely configured variable specifies the string name of the short monitoring time span to use for throttling. The named time span must exist in `TRANSFER_IO_REPORT_TIMESPANS` . The default is `1m`, which is 1 minute.

FILE_TRANSFER_DISK_LOAD_THROTTLE_LONG_HORIZON
This rarely configured variable specifies the string name of the long monitoring time span to use for throttling. The named time span must exist in `TRANSFER_IO_REPORT_TIMESPANS` . The default is `5m`, which is 5 minutes.

TRANSFER_QUEUE_USER_EXPR
This rarely configured expression specifies the user name to be used for scheduling purposes in the file transfer

queue. The scheduler attempts to give equal weight to each user when there are multiple jobs waiting to transfer files within the limits set by MAX_CONCURRENT_UPLOADS and/or MAX_CONCURRENT_DOWNLOADS . When choosing a new job to allow to transfer, the first job belonging to the transfer queue user who has least number of active transfers will be selected. In case of a tie, the user who has least recently been given an opportunity to start a transfer will be selected. By default, a transfer queue user is identified as the job owner. A different user name may be specified by configuring TRANSFER_QUEUE_USER_EXPR to a string expression that is evaluated in the context of the job ad. For example, if this expression were set to a name that is the same for all jobs, file transfers would be scheduled in first-in-first-out order rather than equal share order. Note that the string produced by this expression is used as a prefix in the ClassAd attributes for per-user file transfer I/O statistics that are published in the *condor_schedd* ClassAd.

MAX_TRANSFER_INPUT_MB
This integer expression specifies the maximum allowed total size in MiB of the input files that are transferred for a job. This expression does not apply to grid universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. The default value is -1. The job may override the system setting by specifying its own limit using the MaxTransferInputMB attribute. If the observed size of all input files at submit time is larger than the limit, the job will be immediately placed on hold with a HoldReasonCode value of 32. If the job passes this initial test, but the size of the input files increases or the limit decreases so that the limit is violated, the job will be placed on hold at the time when the file transfer is attempted.

MAX_TRANSFER_OUTPUT_MB
This integer expression specifies the maximum allowed total size in MiB of the output files that are transferred for a job. This expression does not apply to grid universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. The default value is -1. The job may override the system setting by specifying its own limit using the MaxTransferOutputMB attribute. If the total size of the job's output files to be transferred is larger than the limit, the job will be placed on hold with a HoldReasonCode value of 33. The output will be transferred up to the point when the limit is hit, so some files may be fully transferred, some partially, and some not at all.

MAX_TRANSFER_QUEUE_AGE
The number of seconds after which an aged and queued transfer may be dequeued from the transfer queue, as it is presumably hung. Defaults to 7200 seconds, which is 120 minutes.

TRANSFER_IO_REPORT_INTERVAL
The sampling interval in seconds for collecting I/O statistics for file transfer. The default is 10 seconds. To provide sufficient resolution, the sampling interval should be small compared to the smallest time span that is configured in TRANSFER_IO_REPORT_TIMESPANS. The shorter the sampling interval, the more overhead of data collection, which may slow down the *condor_schedd*. See *Scheduler ClassAd Attributes* for a description of the published attributes.

TRANSFER_IO_REPORT_TIMESPANS
A string that specifies a list of time spans over which I/O statistics are reported, using exponential moving averages (like the 1m, 5m, and 15m load averages in Unix). Each entry in the list consists of a label followed by a colon followed by the number of seconds over which the named time span should extend. The default is 1m:60 5m:300 1h:3600 1d:86400. To provide sufficient resolution, the smallest reported time span should be large compared to the sampling interval, which is configured by TRANSFER_IO_REPORT_INTERVAL. See *Scheduler ClassAd Attributes* for a description of the published attributes.

SCHEDD_QUERY_WORKERS
This specifies the maximum number of concurrent sub-processes that the *condor_schedd* will spawn to handle queries. The setting is ignored in Windows. In Unix, the default is 8. If the limit is reached, the next query will be handled in the *condor_schedd* 's main process.

CONDOR_Q_USE_V3_PROTOCOL
A boolean value that, when True, causes the *condor_schedd* to use an algorithm that responds to *condor_q* requests by not forking itself to handle each request. It instead handles the requests in a non-blocking way. The

default value is `True`.

CONDOR_Q_DASH_BATCH_IS_DEFAULT
A boolean value that, when `True`, causes *condor_q* to print the **-batch** output unless the **-nobatch** option is used or the other arguments to *condor_q* are incompatible with batch mode. For instance **-long** is incompatible with **-batch**. The default value is `True`.

CONDOR_Q_ONLY_MY_JOBS
A boolean value that, when `True`, causes *condor_q* to request that only the current user's jobs be queried unless the current user is a queue superuser. It also causes the *condor_schedd* to honor that request. The default value is `True`. A value of `False` in either *condor_q* or the *condor_schedd* will result in the old behavior of querying all jobs.

CONDOR_Q_SHOW_OLD_SUMMARY
A boolean value that, when `True`, causes *condor_q* to show the old single line summary totals. When `False` *condor_q* will show the new multi-line summary totals.

SCHEDD_INTERVAL
This macro determines the maximum interval for both how often the *condor_schedd* sends a ClassAd update to the *condor_collector* and how often the *condor_schedd* daemon evaluates jobs. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

ABSENT_SUBMITTER_LIFETIME
This macro determines the maximum time that the *condor_schedd* will remember a submitter after the last job for that submitter leaves the queue. It is defined in terms of seconds and defaults to 1 week.

ABSENT_SUBMITTER_UPDATE_RATE
This macro can be used to set the maximum rate at which the *condor_schedd* sends updates to the *condor_collector* for submitters that have no jobs in the queue. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

WINDOWED_STAT_WIDTH
The number of seconds that forms a time window within which performance statistics of the *condor_schedd* daemon are calculated. Defaults to 300 seconds.

SCHEDD_INTERVAL_TIMESLICE
The bookkeeping done by the *condor_schedd* takes more time when there are large numbers of jobs in the job queue. However, when it is not too expensive to do this bookkeeping, it is best to keep the collector up to date with the latest state of the job queue. Therefore, this macro is used to adjust the bookkeeping interval so that it is done more frequently when the cost of doing so is relatively small, and less frequently when the cost is high. The default is 0.05, which means the schedd will adapt its bookkeeping interval to consume no more than 5% of the total time available to the schedd. The lower bound is configured by `SCHEDD_MIN_INTERVAL` (default 5 seconds), and the upper bound is configured by `SCHEDD_INTERVAL` (default 300 seconds).

JOB_START_COUNT
This macro works together with the `JOB_START_DELAY` macro to throttle job starts. The default and minimum values for this integer configuration variable are both 1.

JOB_START_DELAY
This integer-valued macro works together with the `JOB_START_COUNT` macro to throttle job starts. The *condor_schedd* daemon starts `$(JOB_START_COUNT)` jobs at a time, then delays for `$(JOB_START_DELAY)` seconds before starting the next set of jobs. This delay prevents a sudden, large load on resources required by the jobs during their start up phase. The resulting job start rate averages as fast as (`$(JOB_START_COUNT)/$(JOB_START_DELAY)`) jobs/second. This setting is defined in terms of seconds and defaults to 0, which means jobs will be started as fast as possible. If you wish to throttle the rate of specific types of jobs, you can use the job attribute `NextJobStartDelay`.

MAX_NEXT_JOB_START_DELAY
An integer number of seconds representing the maximum allowed value of the job ClassAd attribute

NextJobStartDelay. It defaults to 600, which is 10 minutes.

JOB_STOP_COUNT
An integer value representing the number of jobs operated on at one time by the *condor_schedd* daemon, when throttling the rate at which jobs are stopped via *condor_rm*, *condor_hold*, or *condor_vacate_job*. The default and minimum values are both 1. This variable is ignored for grid and scheduler universe jobs.

JOB_STOP_DELAY
An integer value representing the number of seconds delay utilized by the *condor_schedd* daemon, when throttling the rate at which jobs are stopped via *condor_rm*, *condor_hold*, or *condor_vacate_job*. The *condor_schedd* daemon stops $(JOB_STOP_COUNT) jobs at a time, then delays for $(JOB_STOP_DELAY) seconds before stopping the next set of jobs. This delay prevents a sudden, large load on resources required by the jobs when they are terminating. The resulting job stop rate averages as fast as JOB_STOP_COUNT/JOB_STOP_DELAY jobs per second. This configuration variable is also used during the graceful shutdown of the *condor_schedd* daemon. During graceful shutdown, this macro determines the wait time in between requesting each *condor_shadow* daemon to gracefully shut down. The default value is 0, which means jobs will be stopped as fast as possible. This variable is ignored for grid and scheduler universe jobs.

JOB_IS_FINISHED_COUNT
An integer value representing the number of jobs that the *condor_schedd* will let permanently leave the job queue each time that it examines the jobs that are ready to do so. The default value is 1.

JOB_IS_FINISHED_INTERVAL
The *condor_schedd* maintains a list of jobs that are ready to permanently leave the job queue, for example, when they have completed or been removed. This integer-valued macro specifies a delay in seconds between instances of taking jobs permanently out of the queue. The default value is 0, which tells the *condor_schedd* to not impose any delay.

ALIVE_INTERVAL
An initial value for an integer number of seconds defining how often the *condor_schedd* sends a UDP keep alive message to any *condor_startd* it has claimed. When the *condor_schedd* claims a *condor_startd*, the *condor_schedd* tells the *condor_startd* how often it is going to send these messages. The utilized interval for sending keep alive messages is the smallest of the two values ALIVE_INTERVAL and the expression JobLeaseDuration/3, formed with the job ClassAd attribute JobLeaseDuration. The value of the interval is further constrained by the floor value of 10 seconds. If the *condor_startd* does not receive any of these keep alive messages during a certain period of time (defined via) the *condor_startd* releases the claim, and the *condor_schedd* no longer pays for the resource (in terms of user priority in the system). The macro is defined in terms of seconds and defaults to 300, which is 5 minutes.

STARTD_SENDS_ALIVES
Note: This setting is deprecated, and may go away in a future version of HTCondor. This setting is mainly useful when running mixing very old *condor_schedd* daemons with newer pools. A boolean value that defaults to True, causing keep alive messages to be sent from the *condor_startd* to the *condor_schedd* by TCP during a claim. When False, the *condor_schedd* daemon sends keep alive signals to the *condor_startd*, reversing the direction. If both *condor_startd* and *condor_schedd* daemons are HTCondor version 7.5.4 or more recent, this variable is only used by the *condor_schedd* daemon. For earlier HTCondor versions, the variable must be set to the same value, and it must be set for both daemons.

REQUEST_CLAIM_TIMEOUT
This macro sets the time (in seconds) that the *condor_schedd* will wait for a claim to be granted by the *condor_startd*. The default is 30 minutes. This is only likely to matter if NEGOTIATOR_CONSIDER_EARLY_PREEMPTION is True, and the *condor_startd* has an existing claim, and it takes a long time for the existing claim to be preempted due to MaxJobRetirementTime. Once a request times out, the *condor_schedd* will simply begin the process of finding a machine for the job all over again.

Normally, it is not a good idea to set this to be very small, where a small value is a few minutes. Doing so can lead to failure to preempt, because the preempting job will spend a significant fraction of its time waiting to be

re-matched. During that time, it would miss out on any opportunity to run if the job it is trying to preempt gets out of the way.

SHADOW_SIZE_ESTIMATE
The estimated private virtual memory size of each *condor_shadow* process in KiB. This value is only used if `RESERVED_SWAP` is non-zero. The default value is 800.

SHADOW_RENICE_INCREMENT
When the *condor_schedd* spawns a new *condor_shadow*, it can do so with a nice-level. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority so that the processes run with less priority than other tasks on the machine. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 0.

SCHED_UNIV_RENICE_INCREMENT
Analogous to `JOB_RENICE_INCREMENT` and `SHADOW_RENICE_INCREMENT`, scheduler universe jobs can be given a nice-level. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 0.

QUEUE_CLEAN_INTERVAL
The *condor_schedd* maintains the job queue on a given machine. It does so in a persistent way such that if the *condor_schedd* crashes, it can recover a valid state of the job queue. The mechanism it uses is a transaction-based log file (the `job_queue.log` file, not the `SchedLog` file). This file contains an initial state of the job queue, and a series of transactions that were performed on the queue (such as new jobs submitted or jobs completing). Periodically, the *condor_schedd* will go through this log, truncate all the transactions and create a new file with containing only the new initial state of the log. This is a somewhat expensive operation, but it speeds up when the *condor_schedd* restarts since there are fewer transactions it has to play to figure out what state the job queue is really in. This macro determines how often the *condor_schedd* should rework this queue to cleaning it up. It is defined in terms of seconds and defaults to 86400 (once a day).

WALL_CLOCK_CKPT_INTERVAL
The job queue contains a counter for each job's "wall clock" run time, i.e., how long each job has executed so far. This counter is displayed by *condor_q*. The counter is updated when the job is evicted or when the job completes. When the *condor_schedd* crashes, the run time for jobs that are currently running will not be added to the counter (and so, the run time counter may become smaller than the CPU time counter). The *condor_schedd* saves run time "checkpoints" periodically for running jobs so if the *condor_schedd* crashes, only run time since the last checkpoint is lost. This macro controls how often the *condor_schedd* saves run time checkpoints. It is defined in terms of seconds and defaults to 3600 (one hour). A value of 0 will disable wall clock checkpoints.

QUEUE_ALL_USERS_TRUSTED
Defaults to False. If set to True, then unauthenticated users are allowed to write to the queue, and also we always trust whatever the `Owner` value is set to be by the client in the job ad. This was added so users can continue to use the SOAP web-services interface over HTTP (w/o authenticating) to submit jobs in a secure, controlled environment - for instance, in a portal setting.

QUEUE_SUPER_USERS
A comma and/or space separated list of user names on a given machine that are given super-user access to the job queue, meaning that they can modify or delete the job ClassAds of other users. These should be of form `USER@DOMAIN`; if the domain is not present in the username, HTCondor will assume the default `UID_DOMAIN`. When not on this list, users can only modify or delete their own ClassAds from the job queue. Whatever user name corresponds with the UID that HTCondor is running as - usually user condor - will automatically be included in this list, because that is needed for HTCondor's proper functioning. See *User Accounts in HTCondor on Unix Platforms* on UIDs in HTCondor for more details on this. By default, the Unix user root and the Windows user administrator are given the ability to remove other user's jobs, in addition to user condor. In addition to a single user, Unix user groups may be specified by using a special syntax defined for this configuration variable; the syntax is the percent character (`%`) followed by the user group name. All members of the user group are given super-user access.

QUEUE_SUPER_USER_MAY_IMPERSONATE

A regular expression that matches the operating system user names (that is, job owners in the form USER) that the queue super user may impersonate when managing jobs. This allows the admin to limit the operating system users a super user can launch jobs as. When not set, the default behavior is to allow impersonation of any user who has had a job in the queue during the life of the *condor_schedd*. For proper functioning of the *condor_shadow*, the *condor_gridmanager*, and the *condor_job_router*, this expression, if set, must match the owner names of all jobs that these daemons will manage. Note that a regular expression that matches only part of the user name is still considered a match. If acceptance of partial matches is not desired, the regular expression should begin with ^ and end with $.

SYSTEM_JOB_MACHINE_ATTRS
This macro specifies a space and/or comma separated list of machine attributes that should be recorded in the job ClassAd. The default attributes are Cpus and SlotWeight. When there are multiple run attempts, history of machine attributes from previous run attempts may be kept. The number of run attempts to store is specified by the configuration variable SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH . A machine attribute named X will be inserted into the job ClassAd as an attribute named MachineAttrX0. The previous value of this attribute will be named MachineAttrX1, the previous to that will be named MachineAttrX2, and so on, up to the specified history length. A history of length 1 means that only MachineAttrX0 will be recorded. Additional attributes to record may be specified on a per-job basis by using the **job_machine_attrs** submit file command. The value recorded in the job ClassAd is the evaluation of the machine attribute in the context of the job ClassAd when the *condor_schedd* daemon initiates the start up of the job. If the evaluation results in an Undefined or Error result, the value recorded in the job ClassAd will be Undefined or Error respectively.

SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH
The integer number of run attempts to store in the job ClassAd when recording the values of machine attributes listed in SYSTEM_JOB_MACHINE_ATTRS . The default is 1. The history length may also be extended on a per-job basis by using the submit file command **job_machine_attrs_history_length** . The larger of the system and per-job history lengths will be used. A history length of 0 disables recording of machine attributes.

SCHEDD_LOCK
This macro specifies what lock file should be used for access to the SchedLog file. It must be a separate file from the SchedLog, since the SchedLog may be rotated and synchronization across log file rotations is desired. This macro is defined relative to the $(LOCK) macro.

SCHEDD_NAME
Used to give an alternative value to the Name attribute in the *condor_schedd* 's ClassAd.

See the description of for defaults and composition of valid HTCondor daemon names.

SCHEDD_ATTRS
This macro is described in .

SCHEDD_DEBUG
This macro (and other settings related to debug logging in the *condor_schedd*) is described in .

SCHEDD_ADDRESS_FILE
This macro is described in .

SCHEDD_EXECUTE
A directory to use as a temporary sandbox for local universe jobs. Defaults to $(SPOOL)/execute.

FLOCK_NEGOTIATOR_HOSTS
Defines a comma and/or space separated list of *condor_negotiator* host names for pools in which the *condor_schedd* should attempt to run jobs. If not set, the *condor_schedd* will query the *condor_collector* daemons for the addresses of the *condor_negotiator* daemons. If set, then the *condor_negotiator* daemons must be specified in order, corresponding to the list set by FLOCK_COLLECTOR_HOSTS. In the typical case, where each pool has the *condor_collector* and *condor_negotiator* running on the same machine, $(FLOCK_NEGOTIATOR_HOSTS) should have the same definition as $(FLOCK_COLLECTOR_HOSTS). This configuration value is also typically used as a macro for adding the *condor_negotiator* to the relevant authorization lists.

FLOCK_COLLECTOR_HOSTS
This macro defines a list of collector host names (not including the local $(COLLECTOR_HOST) machine) for pools in which the *condor_schedd* should attempt to run jobs. Hosts in the list should be in order of preference. The *condor_schedd* will only send a request to a central manager in the list if the local pool and pools earlier in the list are not satisfying all the job requests. must also be configured to allow negotiators from all of the pools to contact the *condor_schedd* at the NEGOTIATOR authorization level. Similarly, the central managers of the remote pools must be configured to allow this *condor_schedd* to join the pool (this requires ADVERTISE_SCHEDD authorization level, which defaults to WRITE).

FLOCK_INCREMENT
This integer value controls how quickly flocking to various pools will occur. It defaults to 1, meaning that pools will be considered for flocking slowly. The first *condor_collector* daemon listed in FLOCK_COLLECTOR_HOSTS will be considered for flocking, and then the second, and so on. A larger value increases the number of *condor_collector* daemons to be considered for flocking. For example, a value of 2 will partition the FLOCK_COLLECTOR_HOSTS into sets of 2 *condor_collector* daemons, and each set will be considered for flocking.

MIN_FLOCK_LEVEL
This integer value specifies a number of remote pools that the *condor_schedd* should always flock to. It defaults to 0, meaning that none of the pools listed in FLOCK_COLLECTOR_HOSTS will be considered for flocking when there are no idle jobs in need of match-making. Setting a larger value N means the *condor_schedd* will always flock to (i.e. look for matches in) the first N pools listed in FLOCK_COLLECTOR_HOSTS.

NEGOTIATE_ALL_JOBS_IN_CLUSTER
If this macro is set to False (the default), when the *condor_schedd* fails to start an idle job, it will not try to start any other idle jobs in the same cluster during that negotiation cycle. This makes negotiation much more efficient for large job clusters. However, in some cases other jobs in the cluster can be started even though an earlier job can't. For example, the jobs' requirements may differ, because of different disk space, memory, or operating system requirements. Or, machines may be willing to run only some jobs in the cluster, because their requirements reference the jobs' virtual memory size or other attribute. Setting this macro to True will force the *condor_schedd* to try to start all idle jobs in each negotiation cycle. This will make negotiation cycles last longer, but it will ensure that all jobs that can be started will be started.

PERIODIC_EXPR_INTERVAL
This macro determines the minimum period, in seconds, between evaluation of periodic job control expressions, such as periodic_hold, periodic_release, and periodic_remove, given by the user in an HTCondor submit file. By default, this value is 60 seconds. A value of 0 prevents the *condor_schedd* from performing the periodic evaluations.

MAX_PERIODIC_EXPR_INTERVAL
This macro determines the maximum period, in seconds, between evaluation of periodic job control expressions, such as periodic_hold, periodic_release, and periodic_remove, given by the user in an HTCondor submit file. By default, this value is 1200 seconds. If HTCondor is behind on processing events, the actual period between evaluations may be higher than specified.

PERIODIC_EXPR_TIMESLICE
This macro is used to adapt the frequency with which the *condor_schedd* evaluates periodic job control expressions. When the job queue is very large, the cost of evaluating all of the ClassAds is high, so in order for the *condor_schedd* to continue to perform well, it makes sense to evaluate these expressions less frequently. The default time slice is 0.01, so the *condor_schedd* will set the interval between evaluations so that it spends only 1% of its time in this activity. The lower bound for the interval is configured by PERIODIC_EXPR_INTERVAL (default 60 seconds) and the upper bound is configured with MAX_PERIODIC_EXPR_INTERVAL (default 1200 seconds).

SYSTEM_PERIODIC_HOLD_NAMES
A comma and/or space separated list of unique names, where each is used in the formation of a configuration variable name that will contain an expression that will be periodically evaluated for each job that is not in the HELD, COMPLETED, or REMOVED state. Each name in the list will be used in the name of configuration vari-

able SYSTEM_PERIODIC_HOLD_<Name>. The named expressions are evaluated in the order in which names appear in this list. Names are not case-sensitive. After all of the named expressions are evaluated, the nameless SYSTEM_PERIODIC_HOLD expression will be evaluated. If any of these expression evaluates to True the job will be held. See also SYSTEM_PERIODIC_HOLD There is no default value.

SYSTEM_PERIODIC_HOLD and SYSTEM_PERIODIC_HOLD_<Name>
This expression behaves identically to the job expression periodic_hold, but it is evaluated for every job in the queue. It defaults to False. When True, it causes the job to stop running and go on hold. Here is an example that puts jobs on hold if they have been restarted too many times, have an unreasonably large virtual memory ImageSize, or have unreasonably large disk usage for an invented environment.

```
if version > 9.5
   # use hold names if the version supports it
   SYSTEM_PERIODIC_HOLD_NAMES = Mem Disk
   SYSTEM_PERIODIC_HOLD_Mem = ImageSize > 3000000
   SYSTEM_PERIODIC_HOLD_Disk = JobStatus == 2 && DiskUsage > 10000000
   SYSTEM_PERIODIC_HOLD = JobStatus == 1 && JobRunCount > 10
else
   SYSTEM_PERIODIC_HOLD = \
  (JobStatus == 1 || JobStatus == 2) && \
  (JobRunCount > 10 || ImageSize > 3000000 || DiskUsage > 10000000)
endif
```

SYSTEM_PERIODIC_HOLD_REASON and SYSTEM_PERIODIC_HOLD_<Name>_REASON
This string expression is evaluated when the job is placed on hold due to SYSTEM_PERIODIC_HOLD or SYSTEM_PERIODIC_HOLD_<Name> evaluating to True. If it evaluates to a non-empty string, this value is used to set the job attribute HoldReason. Otherwise, a default description is used.

SYSTEM_PERIODIC_HOLD_SUBCODE and SYSTEM_PERIODIC_HOLD_<Name>_SUBCODE
This integer expression is evaluated when the job is placed on hold due to SYSTEM_PERIODIC_HOLD or SYSTEM_PERIODIC_HOLD_<Name> evaluating to True. If it evaluates to a valid integer, this value is used to set the job attribute HoldReasonSubCode. Otherwise, a default of 0 is used. The attribute HoldReasonCode is set to 26, which indicates that the job went on hold due to a system job policy expression.

SYSTEM_PERIODIC_RELEASE_NAMES
A comma and/or space separated list of unique names, where each is used in the formation of a configuration variable name that will contain an expression that will be periodically evaluated for each job that is in the HELD state (jobs with a HoldReasonCode value of 1 are ignored). Each name in the list will be used in the name of configuration variable SYSTEM_PERIODIC_RELEASE_<Name>. The named expressions are evaluated in the order in which names appear in this list. Names are not case-sensitive. After all of the named expressions are evaluated, the nameless SYSTEM_PERIODIC_RELEASE expression will be evaluated. If any of these expressions evaluates to True the job will be released. See also SYSTEM_PERIODIC_RELEASE There is no default value.

SYSTEM_PERIODIC_RELEASE and SYSTEM_PERIODIC_RELEASE_<Name>
This expression behaves identically to a job's definition of a **periodic_release** expression in a submit description file, but it is evaluated for every job in the queue. It defaults to False. When True, it causes a Held job to return to the Idle state. Here is an example that releases jobs from hold if they have tried to run less than 20 times, have most recently been on hold for over 20 minutes, and have gone on hold due to Connection timed out when trying to execute the job, because the file system containing the job's executable is temporarily unavailable.

```
SYSTEM_PERIODIC_RELEASE = \
  (JobRunCount < 20 && (time() - EnteredCurrentStatus) > 1200 ) &&  \
    (HoldReasonCode == 6 && HoldReasonSubCode == 110)
```

SYSTEM_PERIODIC_REMOVE_NAMES
A comma and/or space separated list of unique names, where each is used in the formation of a configuration

variable name that will contain an expression that will be periodically evaluated for each job in the queue. Each name in the list will be used in the name of configuration variable SYSTEM_PERIODIC_REMOVE_<Name>. The named expressions are evaluated in the order in which names appear in this list. Names are not case-sensitive. After all of the named expressions are evaluated, the nameless SYSTEM_PERIODIC_REMOVE expression will be evaluated. If any of these expressions evaluates to True the job will be removed from the queue. See also SYSTEM_PERIODIC_REMOVE There is no default value.

SYSTEM_PERIODIC_REMOVE and SYSTEM_PERIODIC_REMOVE_<Name>
This expression behaves identically to the job expression periodic_remove, but it is evaluated for every job in the queue. As it is in the configuration file, it is easy for an administrator to set a remove policy that applies to all jobs. It defaults to False. When True, it causes the job to be removed from the queue. Here is an example that removes jobs which have been on hold for 30 days:

```
SYSTEM_PERIODIC_REMOVE = \
  (JobStatus == 5 && time() - EnteredCurrentStatus > 3600*24*30)
```

SCHEDD_ASSUME_NEGOTIATOR_GONE
This macro determines the period, in seconds, that the *condor_schedd* will wait for the *condor_negotiator* to initiate a negotiation cycle before the schedd will simply try to claim any local *condor_startd*. This allows for a machine that is acting as both a submit and execute node to run jobs locally if it cannot communicate with the central manager. The default value, if not specified, is 2,000,000 seconds (effectively never). If this feature is desired, we recommend setting it to some small multiple of the negotiation cycle, say, 1200 seconds, or 20 minutes.

GRACEFULLY_REMOVE_JOBS
A boolean value defaulting to True. If True, jobs will be given a chance to shut down cleanly when removed. In the vanilla universe, this means that the job will be sent the signal set in its SoftKillSig attribute, or SIGTERM if undefined; if the job hasn't exited after its max vacate time, it will be hard-killed (sent SIGKILL). Signals are different on Windows, and other details differ between universes.

The submit command *want_graceful_removal* overrides this configuration variable.

See for details on how HTCondor computes the job's max vacate time.

SCHEDD_ROUND_ATTR_<xxxx>
This is used to round off attributes in the job ClassAd so that similar jobs may be grouped together for negotiation purposes. There are two cases. One is that a percentage such as 25% is specified. In this case, the value of the attribute named <xxxx>\ in the job ClassAd will be rounded up to the next multiple of the specified percentage of the values order of magnitude. For example, a setting of 25% will cause a value near 100 to be rounded up to the next multiple of 25 and a value near 1000 will be rounded up to the next multiple of 250. The other case is that an integer, such as 4, is specified instead of a percentage. In this case, the job attribute is rounded up to the specified number of decimal places. Replace <xxxx> with the name of the attribute to round, and set this macro equal to the number of decimal places to round up. For example, to round the value of job ClassAd attribute foo up to the nearest 100, set

```
SCHEDD_ROUND_ATTR_foo = 2
```

When the schedd rounds up an attribute value, it will save the raw (un-rounded) actual value in an attribute with the same name appended with "_RAW". So in the above example, the raw value will be stored in attribute foo_RAW in the job ClassAd. The following are set by default:

```
SCHEDD_ROUND_ATTR_ResidentSetSize = 25%
SCHEDD_ROUND_ATTR_ProportionalSetSizeKb = 25%
SCHEDD_ROUND_ATTR_ImageSize = 25%
SCHEDD_ROUND_ATTR_ExecutableSize = 25%
SCHEDD_ROUND_ATTR_DiskUsage = 25%
SCHEDD_ROUND_ATTR_NumCkpts = 4
```

Thus, an ImageSize near 100MB will be rounded up to the next multiple of 25MB. If your batch slots have less memory or disk than the rounded values, it may be necessary to reduce the amount of rounding, because the job requirements will not be met.

SCHEDD_BACKUP_SPOOL
A boolean value that, when `True`, causes the *condor_schedd* to make a backup of the job queue as it starts. When `True`, the *condor_schedd* creates a host-specific backup of the current spool file to the spool directory. This backup file will be overwritten each time the *condor_schedd* starts. Defaults to `False`.

SCHEDD_PREEMPTION_REQUIREMENTS
This boolean expression is utilized only for machines allocated by a dedicated scheduler. When `True`, a machine becomes a candidate for job preemption. This configuration variable has no default; when not defined, preemption will never be considered.

SCHEDD_PREEMPTION_RANK
This floating point value is utilized only for machines allocated by a dedicated scheduler. It is evaluated in context of a job ClassAd, and it represents a machine's preference for running a job. This configuration variable has no default; when not defined, preemption will never be considered.

ParallelSchedulingGroup
For parallel jobs which must be assigned within a group of machines (and not cross group boundaries), this configuration variable is a string which identifies a group of which this machine is a member. Each machine within a group sets this configuration variable with a string that identifies the group.

PER_JOB_HISTORY_DIR
If set to a directory writable by the HTCondor user, when a job leaves the *condor_schedd* 's queue, a copy of the job's ClassAd will be written in that directory. The files are named `history`, with the job's cluster and process number appended. For example, job 35.2 will result in a file named `history.35.2`. HTCondor does not rotate or delete the files, so without an external entity to clean the directory, it can grow very large. This option defaults to being unset. When not set, no files are written.

DEDICATED_SCHEDULER_USE_FIFO
When this parameter is set to true (the default), parallel universe jobs will be scheduled in a first-in, first-out manner. When set to false, parallel jobs are scheduled using a best-fit algorithm. Using the best-fit algorithm is not recommended, as it can cause starvation.

DEDICATED_SCHEDULER_WAIT_FOR_SPOOLER
A boolean value that when `True`, causes the dedicated scheduler to schedule parallel universe jobs in a very strict first-in, first-out manner. When the default value of `False`, parallel jobs that are being remotely submitted to a scheduler and are on hold, waiting for spooled input files to arrive at the scheduler, will not block jobs that arrived later, but whose input files have finished spooling. When `True`, jobs with larger cluster IDs, but that are in the Idle state will not be scheduled to run until all earlier jobs have finished spooling in their input files and have been scheduled.

SCHEDD_SEND_VACATE_VIA_TCP
A boolean value that defaults to `True`. When `True`, the *condor_schedd* daemon sends vacate signals via TCP, instead of the default UDP.

SCHEDD_CLUSTER_INITIAL_VALUE
An integer that specifies the initial cluster number value to use within a job id when a job is first submitted. If the job cluster number reaches the value set by `SCHEDD_CLUSTER_MAXIMUM_VALUE` and wraps, it will be re-set to the value given by this variable. The default value is 1.

SCHEDD_CLUSTER_INCREMENT_VALUE
A positive integer that defaults to 1, representing a stride used for the assignment of cluster numbers within a job id. When a job is submitted, the job will be assigned a job id. The cluster number of the job id will be equal to the previous cluster number used plus the value of this variable.

SCHEDD_CLUSTER_MAXIMUM_VALUE

An integer that specifies an upper bound on assigned job cluster id values. For value M, the maximum job cluster id assigned to any job will be M - 1. When the maximum id is reached, cluster ids will continue assignment using SCHEDD_CLUSTER_INITIAL_VALUE. The default value of this variable is zero, which represents the behavior of having no maximum cluster id value.

Note that HTCondor does not check for nor take responsibility for duplicate cluster ids for queued jobs. If SCHEDD_CLUSTER_MAXIMUM_VALUE is set to a non-zero value, the system administrator is responsible for ensuring that older jobs do not stay in the queue long enough for cluster ids of new jobs to wrap around and reuse the same id. With a low enough value, it is possible for jobs to be erroneously assigned duplicate cluster ids, which will result in a corrupt job queue.

SCHEDD_JOB_QUEUE_LOG_FLUSH_DELAY
An integer which specifies an upper bound in seconds on how long it takes for changes to the job ClassAd to be visible to the HTCondor Job Router. The default is 5 seconds.

ROTATE_HISTORY_DAILY
A boolean value that defaults to False. When True, the history file will be rotated daily, in addition to the rotations that occur due to the definition of MAX_HISTORY_LOG that rotate due to size.

ROTATE_HISTORY_MONTHLY
A boolean value that defaults to False. When True, the history file will be rotated monthly, in addition to the rotations that occur due to the definition of MAX_HISTORY_LOG that rotate due to size.

SCHEDD_COLLECT_STATS_FOR_<Name>
A boolean expression that when True creates a set of *condor_schedd* ClassAd attributes of statistics collected for a particular set. These attributes are named using the prefix of <Name>. The set includes each entity for which this expression is True. As an example, assume that *condor_schedd* statistics attributes are to be created for only user Einstein's jobs. Defining

```
SCHEDD_COLLECT_STATS_FOR_Einstein = (Owner=="einstein")
```

causes the creation of the set of statistics attributes with names such as EinsteinJobsCompleted and EinsteinJobsCoredumped.

SCHEDD_COLLECT_STATS_BY_<Name>
Defines a string expression. The evaluated string is used in the naming of a set of *condor_schedd* statistics ClassAd attributes. The naming begins with <Name>, an underscore character, and the evaluated string. Each character not permitted in an attribute name will be converted to the underscore character. For example,

```
SCHEDD_COLLECT_STATS_BY_Host = splitSlotName(RemoteHost)[1]
```

a set of statistics attributes will be created and kept. If the string expression were to evaluate to "storm.04.cs.wisc.edu", the names of two of these attributes will be Host_storm_04_cs_wisc_edu_JobsCompleted and Host_storm_04_cs_wisc_edu_JobsCoredumped.

SCHEDD_EXPIRE_STATS_BY_<Name>
The number of seconds after which the *condor_schedd* daemon will stop collecting and discard the statistics for a subset identified by <Name>, if no event has occurred to cause any counter or statistic for the subset to be updated. If this variable is not defined for a particular <Name>, then the default value will be 60*60*24*7, which is one week's time.

SIGNIFICANT_ATTRIBUTES
A comma and/or space separated list of job ClassAd attributes that are to be added to the list of attributes for determining the sets of jobs considered as a unit (an auto cluster) in negotiation, when auto clustering is enabled. When defined, this list replaces the list that the *condor_negotiator* would define based upon machine ClassAds.

ADD_SIGNIFICANT_ATTRIBUTES
A comma and/or space separated list of job ClassAd attributes that will always be added to the list of attributes

that the *condor_negotiator* defines based upon machine ClassAds, for determining the sets of jobs considered as a unit (an auto cluster) in negotiation, when auto clustering is enabled.

REMOVE_SIGNIFICANT_ATTRIBUTES
A comma and/or space separated list of job ClassAd attributes that are removed from the list of attributes that the *condor_negotiator* defines based upon machine ClassAds, for determining the sets of jobs considered as a unit (an auto cluster) in negotiation, when auto clustering is enabled.

SCHEDD_SEND_RESCHEDULE
A boolean value which defaults to true. Set to false for schedds like those in the HTCondor-CE that have no negotiator associated with them, in order to reduce spurious error messages in the SchedLog file.

SCHEDD_AUDIT_LOG
The path and file name of the *condor_schedd* log that records user-initiated commands that modify the job queue. If not defined, there will be no *condor_schedd* audit log.

MAX_SCHEDD_AUDIT_LOG
Controls the maximum amount of time that a log will be allowed to grow. When it is time to rotate a log file, it will be saved to a file with an ISO timestamp suffix. The oldest rotated file receives the file name suffix `.old`. The `.old` files are overwritten each time the maximum number of rotated files (determined by the value of `MAX_NUM_SCHEDD_AUDIT_LOG`) is exceeded. A value of 0 specifies that the file may grow without bounds. The following suffixes may be used to qualify the integer:

`Sec` for seconds `Min` for minutes `Hr` for hours `Day` for days `Wk` for weeks

MAX_NUM_SCHEDD_AUDIT_LOG
The integer that controls the maximum number of rotations that the *condor_schedd* audit log is allowed to perform, before the oldest one will be rotated away. The default value is 1.

SCHEDD_USE_SLOT_WEIGHT
A boolean that defaults to `False`. When `True`, the *condor_schedd* does use configuration variable `SLOT_WEIGHT` to weight running and idle job counts in the submitter ClassAd.

EXTENDED_SUBMIT_COMMANDS
A long form ClassAd that defines extended submit commands and their associated job ad attributes for a specific Schedd. *condor_submit* will query the destination schedd for this ClassAd and use it to modify the internal table of submit commands before interpreting the submit file.

Each entry in this ClassAd will define a new submit command, the value will indicate the required data type to the submit file parser with the data type given by example from the value according to this list of types

- *string-list* - a quoted string containing a comma. e.g. `"a,b"`. *string-list* values are converted to canonical form.
- *filename* - a quoted string beginning with the word file. e.g. `"filename"`. *filename* values are converted to fully qualified file paths using the same rules as other submit filenames.
- *string* - a quoted string that does not match the above special rules. e.g. `"string"`. *string* values can be provided quoted or unquoted in the submit file. Unquoted values will have leading and trailing whitespace removed.
- *unsigned-integer* - any non-negative integer e.g. `0`. *unsigned-integer* values are evaluated as expressions and submit will fail if the result does not convert to an unsigned integer. A simple integer value will be stored in the job.
- *integer* - any negative integer e.g. `-1`. *integer* values are evaluated as expressions and submit will fail if the result does not convert to an integer. A simple integer value will be stored in the job.
- *boolean* - any boolean value e.g. `true`. *boolean* values are evaluated as expressions and submit will fail if the result does not convert to `true` or `false`.

- *expression* - any expression or floating point number that is not one of the above. e.g. `a+b`. *expression* values will be parsed as a classad expression and stored in the job.
- *error* - the literal `error` will tell submit to generate an error when the command is used. this provides a way for admins to disable existing submit commands.
- *undefined* - the literal `undefined` will be treated by *condor_submit* as if that attribute is not in this ad. This is intended to aid composibility of this ad across multiple configuration files.

The following example will add four new submit commands and disable the use of the the `accounting_group_user` submit command.

```
EXTENDED_SUBMIT_COMMANDS @=end
   LongJob = true
   Project = "string"
   FavoriteFruit = "a,b"
   SomeFile = "filename"
   accounting_group_user = error
@end
```

EXTENDED_SUBMIT_HELPFILE
A URL or file path to text describing how the *condor_schedd* extends the submit schema. Use this to document for users the extended submit commands defined by the configuration variable `EXTENDED_SUBMIT_COMMANDS`. *condor_submit* will display this URL or the text of this file when the user uses the `-capabilities` option.

SUBMIT_TEMPLATE_NAMES
A comma and/or space separated list of unique names, where each is used in the formation of a configuration variable name that will contain a set of submit commands. Each name in the list will be used in the name of the configuration variable `SUBMIT_TEMPLATE_<Name>`. Names are not case-sensitive. There is no default value. Submit templates are used by *condor_submit* when parsing submit files, so administrators or users can add submit templates to the configuration of *condor_submit* to customize the schema or to simplify the creation of submit files.

SUBMIT_TEMPLATE_<Name>
A single submit template containing one or more submit commands. The template can be invoked with or without arguments. The template can refer arguments by number using the `$(<N>)` where `<N>` is a value from 0 thru 9. `$(0)` expands to all of the arguments, `$(1)` to the first argument, `$(2)` to the second argument, and so on. The argument number can be followed by `?` to test if the argument was specified, or by `+` to expand to that argument and all subsequent arguments. Thus `$(0)` and `$(1+)` will expand to the same thing.

For example:

```
SUBMIT_TEMPLATE_NAMES = $(SUBMIT_TEMPLATE_NAMES) Slurm
SUBMIT_TEMPLATE_Slurm @=tpl
   if ! $(1?)
      error : Template:Slurm requires at least 1 argument - Slurm(project, [queue [,
↪ resource_args...])
   endif
   universe = Grid
   grid_resource = batch slurm $(3)
   batch_project = $(1)
   batch_queue = $(2:Default)
@tpl
```

This could be used in a submit file in this way:

```
use template : Slurm(Blue Book)
```

JOB_TRANSFORM_NAMES
A comma and/or space separated list of unique names, where each is used in the formation of a configuration variable name that will contain a set of rules governing the transformation of jobs during submission. Each name in the list will be used in the name of configuration variable JOB_TRANSFORM_<Name>. Transforms are applied in the order in which names appear in this list. Names are not case-sensitive. There is no default value.

JOB_TRANSFORM_<Name>
A single job transform specified as a set of transform rules. The syntax for these rules is specified in *ClassAd Transforms* The transform rules are applied to jobs that match the transform's REQUIREMENTS expression as they are submitted. <Name> corresponds to a name listed in JOB_TRANSFORM_NAMES. Names are not case-sensitive. There is no default value. For jobs submitted as late materialization factories, the factory Cluster ad is transformed at submit time. When job ads are later materialized, attribute values set by the transform will override values set by the job factory for those attributes.

SUBMIT_REQUIREMENT_NAMES
A comma and/or space separated list of unique names, where each is used in the formation of a configuration variable name that will represent an expression evaluated to decide whether or not to reject a job submission. Each name in the list will be used in the name of configuration variable SUBMIT_REQUIREMENT_<Name>. There is no default value.

SUBMIT_REQUIREMENT_<Name>
A boolean expression evaluated in the context of the *condor_schedd* daemon ClassAd, which is the SCHEDD. or MY. name space and the job ClassAd, which is the JOB. or TARGET. name space. When False, it causes the *condor_schedd* to reject the submission of the job or cluster of jobs. <Name> corresponds to a name listed in SUBMIT_REQUIREMENT_NAMES. There is no default value.

SUBMIT_REQUIREMENT_<Name>_REASON
An expression that evaluates to a string, to be printed for the job submitter when SUBMIT_REQUIREMENT_<Name> evaluates to False and the *condor_schedd* rejects the job. There is no default value.

SCHEDD_RESTART_REPORT
The complete path to a file that will be written with report information. The report is written when the *condor_schedd* starts. It contains statistics about its attempts to reconnect to the *condor_startd* daemons for all jobs that were previously running. The file is updated periodically as reconnect attempts succeed or fail. Once all attempts have completed, a copy of the report is emailed to address specified by CONDOR_ADMIN. The default value is $(LOG)/ScheddRestartReport. If a blank value is set, then no report is written or emailed.

JOB_SPOOL_PERMISSIONS
Control the permissions on the job's spool directory. Defaults to user which sets permissions to 0700. Possible values are user, group, and world. If set to group, then the directory is group-accessible, with permissions set to 0750. If set to world, then the directory is created with permissions set to 0755.

CHOWN_JOB_SPOOL_FILES
Prior to HTCondor 8.5.0 on unix, the condor_schedd would chown job files in the SPOOL directory between the condor account and the account of the job submitter. Now, these job files are always owned by the job submitter by default. To restore the older behavior, set this parameter to True. The default value is False.

IMMUTABLE_JOB_ATTRS
A comma and/or space separated list of attributes provided by the administrator that cannot be changed, once they have committed values. No attributes are in this list by default.

SYSTEM_IMMUTABLE_JOB_ATTRS
A predefined comma and/or space separated list of attributes that cannot be changed, once they have committed values. The hard-coded value is: Owner ClusterId ProcId MyType TargetType.

PROTECTED_JOB_ATTRS

A comma and/or space separated list of attributes provided by the administrator that can only be altered by the queue super-user, once they have committed values. No attributes are in this list by default.

SYSTEM_PROTECTED_JOB_ATTRS
A predefined comma and/or space separated list of attributes that can only be altered by the queue super-user, once they have committed values. The hard-code value is empty.

ALTERNATE_JOB_SPOOL
A ClassAd expression evaluated in the context of the job ad. If the result is a string, the value is used an an alternate spool directory under which the job's files will be stored. This alternate directory must already exist and have the same file ownership and permissions as the main `SPOOL` directory. Care must be taken that the value won't change during the lifetime of each job.

<OAuth2Service>_CLIENT_ID
The client ID string for an OAuth2 service named <`OAuth2Service`>. The client ID is passed on to the *condor_credmon_oauth* when a job requests OAuth2 credentials for a configured OAuth2 service.

<OAuth2Service>_CLIENT_SECRET_FILE
The path to the file containing the client secret string for an OAuth2 service named <`OAuth2Service`>. The client secret is passed on to the *condor_credmon_oauth* when a job requests OAuth2 credentials for a configured OAuth2 service.

<OAuth2Service>_RETURN_URL_SUFFIX
The path (`https://<hostname>/<path>`) that an OAuth2 service named <`OAuth2Service`> should be directed when returning after a user permits the submit host access to their account. Most often, this should be set to name of the OAuth2 service (e.g. `box`, `gdrive`, `onedrive`, etc.). The derived return URL is passed on to the *condor_credmon_oauth* when a job requests OAuth2 credentials for a configured OAuth2 service.

<OAuth2Service>_AUTHORIZATION_URL
The URL that the companion OAuth2 credmon WSGI application should redirect a user to in order to request access for a user's credentials for the OAuth2 service named <`OAuth2Service`>. This URL should be found in the service's API documentation. The authorization URL is passed on to the *condor_credmon_oauth* when a job requests OAuth2 credentials for a configured OAuth2 service.

<OAuth2Service>_TOKEN_URL
The URL that the *condor_credmon_oauth* should use in order to refresh a user's tokens for the OAuth2 service named <`OAuth2Service`>. This URL should be found in the service's API documentation. The token URL is passed on to the *condor_credmon_oauth* when a job requests OAuth2 credentials for a configured OAuth2 service.

5.5.9 condor_shadow Configuration File Entries

These settings affect the *condor_shadow*.

SHADOW_LOCK
This macro specifies the lock file to be used for access to the `ShadowLog` file. It must be a separate file from the `ShadowLog`, since the `ShadowLog` may be rotated and you want to synchronize access across log file rotations. This macro is defined relative to the `$(LOCK)` macro.

SHADOW_DEBUG
This macro (and other settings related to debug logging in the shadow) is described in .

SHADOW_QUEUE_UPDATE_INTERVAL
The amount of time (in seconds) between ClassAd updates that the *condor_shadow* daemon sends to the *condor_schedd* daemon. Defaults to 900 (15 minutes).

SHADOW_LAZY_QUEUE_UPDATE
This boolean macro specifies if the *condor_shadow* should immediately update the job queue for certain attributes (at this time, it only effects the `NumJobStarts` and `NumJobReconnects` counters) or if it should wait and only update the job queue on the next periodic update. There is a trade-off between performance and the semantics of these attributes, which is why the behavior is controlled by a configuration macro. If the *condor_shadow* do not use a lazy update, and immediately ensures the changes to the job attributes are written to the job queue on disk, the semantics for the attributes are very solid (there's only a tiny chance that the counters will be out of sync with reality), but this introduces a potentially large performance and scalability problem for a busy *condor_schedd*. If the *condor_shadow* uses a lazy update, there is no additional cost to the *condor_schedd*, but it means that *condor_q* will not immediately see the changes to the job attributes, and if the *condor_shadow* happens to crash or be killed during that time, the attributes are never incremented. Given that the most obvious usage of these counter attributes is for the periodic user policy expressions (which are evaluated directly by the *condor_shadow* using its own copy of the job's ClassAd, which is immediately updated in either case), and since the additional cost for aggressive updates to a busy *condor_schedd* could potentially cause major problems, the default is `True` to do lazy, periodic updates.

SHADOW_WORKLIFE
The integer number of seconds after which the *condor_shadow* will exit when the current job finishes, instead of fetching a new job to manage. Having the *condor_shadow* continue managing jobs helps reduce overhead and can allow the *condor_schedd* to achieve higher job completion rates. The default is 3600, one hour. The value 0 causes *condor_shadow* to exit after running a single job.

SHADOW_JOB_CLEANUP_RETRY_DELAY
This integer specifies the number of seconds to wait between tries to commit the final update to the job ClassAd in the *condor_schedd* 's job queue. The default is 30.

SHADOW_MAX_JOB_CLEANUP_RETRIES
This integer specifies the number of times to try committing the final update to the job ClassAd in the *condor_schedd* 's job queue. The default is 5.

SHADOW_CHECKPROXY_INTERVAL
The number of seconds between tests to see if the job proxy has been updated or should be refreshed. The default is 600 seconds (10 minutes). This variable's value should be small in comparison to the refresh interval required to keep delegated credentials from expiring (configured via `DELEGATE_JOB_GSI_CREDENTIALS_REFRESH` and `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`). If this variable's value is too small, proxy updates could happen very frequently, potentially creating a lot of load on the submit machine.

SHADOW_RUN_UNKNOWN_USER_JOBS
A boolean that defaults to `False`. When `True`, it allows the *condor_shadow* daemon to run jobs as user nobody when remotely submitted and from users not in the local password file.

SHADOW_STATS_LOG
The full path and file name of a file that stores TCP statistics for shadow file transfers. (Note that the shadow logs TCP statistics to this file by default. Adding `D_STATS` to the `SHADOW_DEBUG` value will cause TCP statistics to be logged to the normal shadow log file (`$(SHADOW_LOG)`).) If not defined, `SHADOW_STATS_LOG` defaults to `$(LOG)/XferStatsLog`. Setting `SHADOW_STATS_LOG` to `/dev/null` disables logging of shadow TCP file transfer statistics.

MAX_SHADOW_STATS_LOG
Controls the maximum size in bytes or amount of time that the shadow TCP statistics log will be allowed to grow. If not defined, `MAX_SHADOW_STATS_LOG` defaults to `$(MAX_DEFAULT_LOG)`, which currently defaults to 10 MiB in size. Values are specified with the same syntax as `MAX_DEFAULT_LOG`.

ALLOW_TRANSFER_REMAP_TO_MKDIR
A boolean value that when `True` allows the *condor_shadow* to create directories in a transfer output remap path when the directory does not exist already. The *condor_shadow* can not create directories if the remap is an absolute path or if the remap tries to write to a directory specified within `LIMIT_DIRECTORY_ACCESS`.

JOB_EPOCH_HISTORY
A full path and filename of a file where the *condor_shadow* will write to a per run job history file in an analogous way to that of the history file defined by the configuration variable `HISTORY`. It will be rotated in the same way, and has similar parameters that apply to the `HISTORY` file rotation apply to the *condor_shadow* daemon epoch history as well. This can be read with the *condor_history* command using the -epochs option. By default this option is not set.

```
$ condor_history -epochs
```

MAX_EPOCH_HISTORY_LOG
Defines the maximum size for the epoch history file, in bytes. It defaults to 20MB.

MAX_EPOCH_HISTORY_ROTATIONS
Controls the maximum number of backup epoch history files to be kept. It defaults to 2, which means that there may be up to three epoch history files (two backups, plus the epoch history file that is being currently written to). When the epoch history file is rotated, and this rotation would cause the number of backups to be too large, the oldest file is removed.

JOB_EPOCH_HISTORY_DIR
A full path to an existing directory that the *condor_shadow* will write the jobs current job ad to a per job run history file with the name `job.runs.X.Y.ads`. Where `X` is the jobs cluster id and `Y` is the jobs process id. For example, job 35.2 would write a job ad for each run to the file `job.runs.35.2.ads`. These files can be read through *condor_history* when ran with the -epochs and -directory options.

```
$ condor_history -epochs -directory
```

HTCondor does not automatically delete these files, so unchecked the directory can grow very large. Either an external entity needs to clean up or *condor_history* can use the -epochs options optional `:d` extension to read and delete the files.

```
$ condor_history -epochs:d -directory
```

5.5.10 condor_starter Configuration File Entries

These settings affect the *condor_starter*.

DISABLE_SETUID
HTCondor can prevent jobs from running setuid executables on Linux by setting the no-new-privileges flag. This can be enabled (i.e. to disallow setuid binaries) by setting `DISABLE_SETUID` to true.

JOB_RENICE_INCREMENT
When the *condor_starter* spawns an HTCondor job, it can do so with a nice-level. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority, such that these processes do not interfere with interactive use of the machine. For machines with lots of real memory and swap space, such that the only scarce resource is CPU time, use this macro in conjunction with a policy that allows HTCondor to always start jobs on the machines. HTCondor jobs would always run, but interactive response on the machines would never suffer. A user most likely will not notice HTCondor is running jobs. See *Policy Configuration for Execution Points and for Access Points* for more details on setting up a policy for starting and stopping jobs on a given machine.

The ClassAd expression is evaluated in the context of the job ad to an integer value, which is set by the *condor_starter* daemon for each job just before the job runs. The range of allowable values are integers in the range of 0 to 19 (inclusive), with a value of 19 being the lowest priority. If the integer value is outside this range, then on a Unix machine, a value greater than 19 is auto-decreased to 19; a value less than 0 is treated as 0. For values

outside this range, a Windows machine ignores the value and uses the default instead. The default value is 0, on Unix, and the idle priority class on a Windows machine.

STARTER_LOCAL_LOGGING
This macro determines whether the starter should do local logging to its own log file, or send debug information back to the *condor_shadow* where it will end up in the ShadowLog. It defaults to `True`.

STARTER_LOG_NAME_APPEND
A fixed value that sets the file name extension of the local log file used by the *condor_starter* daemon. Permitted values are `true`, `false`, `slot`, `cluster` and `jobid`. A value of `false` will suppress the use of a file extension. A value of `true` gives the default behavior of using the slot name, unless there is only a single slot. A value of `slot` uses the slot name. A value of `cluster` uses the job's `ClusterId` ClassAd attribute. A value of `jobid` uses the job's `ClusterId` and `ProcId` ClassAd attributes. If `cluster` or `jobid` are specified, the resulting log files will persist until deleted by the user, so these two options should only be used to assist in debugging, not as permanent options.

STARTER_DEBUG
This setting (and other settings related to debug logging in the starter) is described above in .

STARTER_NUM_THREADS_ENV_VARS
A string containing a list of job environment variables to set equal to the number of cores allocated into the slot. Many commonly used computing libraries and programs will look at the value of environment variables, such as `OMP_NUM_THREADS`, to control how many CPU cores to use. Defaults to CUBACORES, GOMAXPROCS, JULIA_NUM_THREADS, MKL_NUM_THREADS, NUMEXPR_NUM_THREADS, OMP_NUM_THREADS, OMP_THREAD_LIMIT, OPENBLAS_NUM_THREADS, ROOT_MAX_THREADS, TF_LOOP_PARALLEL_ITERATIONS, TF_NUM_THREADS.

STARTER_UPDATE_INTERVAL
An integer value representing the number of seconds between ClassAd updates that the *condor_starter* daemon sends to the *condor_shadow* and *condor_startd* daemons. Defaults to 300 (5 minutes).

STARTER_UPDATE_INTERVAL_TIMESLICE
A floating point value, specifying the highest fraction of time that the *condor_starter* daemon should spend collecting monitoring information about the job, such as disk usage. The default value is 0.1. If monitoring, such as checking disk usage takes a long time, the *condor_starter* will monitor less frequently than specified by `STARTER_UPDATE_INTERVAL`.

STARTER_UPDATE_INTERVAL_MAX
An integer value representing an upper bound on the number of seconds between updates controlled by `STARTER_UPDATE_INTERVAL` and `STARTER_UPDATE_INTERVAL_TIMESLICE`. It is recommended to leave this parameter at its default value, which is calculated as `STARTER_UPDATE_INTERVAL * ( 1 / STARTER_UPDATE_INTERVAL_TIMESLICE )`

USER_JOB_WRAPPER
The full path and file name of an executable or script. If specified, HTCondor never directly executes a job, but instead invokes this executable, allowing an administrator to specify the executable (wrapper script) that will handle the execution of all user jobs. The command-line arguments passed to this program will include the full path to the actual user job which should be executed, followed by all the command-line parameters to pass to the user job. This wrapper script must ultimately replace its image with the user job; thus, it must exec() the user job, not fork() it.

For Bourne type shells (*sh*, *bash*, *ksh*), the last line should be:

```
exec "$@"
```

For the C type shells (*csh*, *tcsh*), the last line should be:

```
exec $*:q
```

On Windows, the end should look like:

```
REM set some environment variables
set LICENSE_SERVER=192.168.1.202:5012
set MY_PARAMS=2

REM Run the actual job now
%*
```

This syntax is precise, to correctly handle program arguments which contain white space characters.

For Windows machines, the wrapper will either be a batch script with a file extension of `.bat` or `.cmd`, or an executable with a file extension of `.exe` or `.com`.

If the wrapper script encounters an error as it runs, and it is unable to run the user job, it is important that the wrapper script indicate this to the HTCondor system so that HTCondor does not assign the exit code of the wrapper script to the job. To do this, the wrapper script should write a useful error message to the file named in the environment variable `_CONDOR_WRAPPER_ERROR_FILE`, and then the wrapper script should exit with a non-zero value. If this file is created by the wrapper script, HTCondor assumes that the wrapper script has failed, and HTCondor will place the job back in the queue marking it as Idle, such that the job will again be run. The *condor_starter* will also copy the contents of this error file to the *condor_starter* log, so the administrator can debug the problem.

When a wrapper script is in use, the executable of a job submission may be specified by a relative path, as long as the submit description file also contains:

```
+PreserveRelativeExecutable = True
```

For example,

```
# Let this executable be resolved by user's path in the wrapper
cmd = sleep
+PreserveRelativeExecutable = True
```

Without this extra attribute:

```
# A typical fully-qualified executable path
cmd = /bin/sleep
```

CGROUP_MEMORY_LIMIT_POLICY
A string with possible values of `hard`, `custom` and `none`. The default value is `hard`. If set to `hard`, when the job tries to use more memory than the slot size, it will be put on hold with an appropriate message. Also, the cgroup soft limit will set to 90% of the hard limit to encourage the kernel to lower cacheable memory the job is using. If set to `none`, no limit will be enforced, but the memory usage of the job will be accurately measured by a cgroup. When set to custom, the additional knob CGROUP_HARD_MEMORY_LIMIT_EXPR must be set, which is a classad expression evaluated in the context of the machine and the job, respectively, to determine the hard limits.

DISABLE_SWAP_FOR_JOB
A boolean that defaults to false. When true, and cgroups are in effect, the *condor_starter* will set the memws to the same value as the hard memory limit. This will prevent the job from using any swap space. If it needs more memory than the hard limit, it will be put on hold. When false, the job is allowed to use any swap space configured by the operating system.

USE_VISIBLE_DESKTOP
This boolean variable is only meaningful on Windows machines. If `True`, HTCondor will allow the job to create windows on the desktop of the execute machine and interact with the job. This is particularly useful for debugging why an application will not run under HTCondor. If `False`, HTCondor uses the default behavior of creating a new, non-visible desktop to run the job on. See the *Microsoft Windows* section for details on how HTCondor interacts with the desktop.

STARTER_JOB_ENVIRONMENT
This macro sets the default environment inherited by jobs. The syntax is the same as the syntax for environment settings in the job submit file (see *condor_submit*). If the same environment variable is assigned by this macro and by the user in the submit file, the user's setting takes precedence.

JOB_INHERITS_STARTER_ENVIRONMENT
A matchlist or boolean value that defaults to `False`. When set to a matchlist it causes jobs to inherit all environment variables from the *condor_starter* that are selected by the match list and not already defined in the job ClassAd or by the `STARTER_JOB_ENVIRONMENT` configuration variable.

A matchlist is a comma, semicolon or space separated list of environment variable names and name patterns that match or reject names. Matchlist members are matched case-insensitively to each name in the environment and those that match are imported. Matchlist members can contain * as wildcard character which matches anything at that position. Members can have two * characters if one of them is at the end. Members can be prefixed with ! to force a matching environment variable to not be imported. The order of members in the Matchlist has no effect on the result. For backward compatibility a single value of `True` behaves as if the value was set to *. Prior to HTCondor version 10.1.0 all values other than `True` are treated as `False`.

NAMED_CHROOT
A comma and/or space separated list of full paths to one or more directories, under which the *condor_starter* may run a chroot-ed job. This allows HTCondor to invoke chroot() before launching a job, if the job requests such by defining the job ClassAd attribute `RequestedChroot` with a directory that matches one in this list. There is no default value for this variable.

STARTER_UPLOAD_TIMEOUT
An integer value that specifies the network communication timeout to use when transferring files back to the access point. The default value is set by the *condor_shadow* daemon to 300. Increase this value if the disk on the access point cannot keep up with large bursts of activity, such as many jobs all completing at the same time.

ASSIGN_CPU_AFFINITY
A boolean expression that defaults to `False`. When it evaluates to `True`, each job under this *condor_startd* is confined to using only as many cores as the configured number of slots. When using partitionable slots, each job will be bound to as many cores as requested by specifying **request_cpus**. When `True`, this configuration variable overrides any specification of `ENFORCE_CPU_AFFINITY`. The expression is evaluated in the context of the Job ClassAd.

ENFORCE_CPU_AFFINITY
This configuration variable is replaced by `ASSIGN_CPU_AFFINITY`. Do not enable this configuration variable unless using glidein or another unusual setup.

A boolean value that defaults to `False`. When `False`, the CPU affinity of processes in a job is not enforced. When `True`, the processes in an HTCondor job maintain their affinity to a CPU. This means that this job will only run on that particular CPU, even if other CPU cores are idle.

If `True` and `SLOT<N>_CPU_AFFINITY` is not set, the CPU that the job is locked to is the same as `SlotID - 1`. Note that slots are numbered beginning with the value 1, while CPU cores are numbered beginning with the value 0.

When `True`, more fine grained affinities may be specified with `SLOT<N>_CPU_AFFINITY`.

SLOT<N>_CPU_AFFINITY
This configuration variable is replaced by `ASSIGN_CPU_AFFINITY`. Do not enable this configuration variable

unless using glidein or another unusual setup.

A comma separated list of cores to which an HTCondor job running on a specific slot given by the value of <N> show affinity. Note that slots are numbered beginning with the value 1, while CPU cores are numbered beginning with the value 0. This affinity list only takes effect when ENFORCE_CPU_AFFINITY = True.

ENABLE_URL_TRANSFERS
A boolean value that when True causes the *condor_starter* for a job to invoke all plug-ins defined by FILETRANSFER_PLUGINS to determine their capabilities for handling protocols to be used in file transfer specified with a URL. When False, a URL transfer specified in a job's submit description file will cause an error issued by *condor_submit*. The default value is True.

FILETRANSFER_PLUGINS
A comma separated list of full and absolute path and executable names for plug-ins that will accomplish the task of doing file transfer when a job requests the transfer of an input file by specifying a URL. See *Custom File Transfer Plugins* for a description of the functionality required of a plug-in.

<PLUGIN>_TEST_URL
This configuration takes a URL to be tested against the specified <PLUGIN>. If this test fails, then that plugin is removed from the *condor_starter* classad attribute HasFileTransferPluginMethods. This attribute determines what plugin capabilities the *condor_starter* can utilize.

RUN_FILETRANSFER_PLUGINS_WITH_ROOT
A boolean value that affects only Unix platforms and defaults to False, causing file transfer plug-ins invoked for a job to run with both the real and the effective UID set to user that the job runs as. The user that the job runs as may be the job owner, nobody, or the slot user. The group is set to primary group of the user that the job runs as, and all supplemental groups are dropped. The default gives the behavior exhibited prior to the existence of this configuration variable. When set to True, file transfer plug-ins are invoked with a real UID of 0 (root), provided the HTCondor daemons also run as root. The effective UID is set to the user that the job runs as.

This configuration variable can permit plug-ins to do privileged operations, such as access a credential protected by file system permissions. The default value is recommended unless privileged operations are required.

MAX_FILE_TRANSFER_PLUGIN_LIFETIME:
An integer number of seconds (defaulting to twenty hours) after which the starter will kill a file transfer plug-in for taking too long. Currently, this causes the job to go on hold with ETIME (62) as the hold reason subcode.

ENABLE_CHIRP
A boolean value that defaults to True. An administrator would set the value to False to disable Chirp remote file access from execute machines.

ENABLE_CHIRP_UPDATES
A boolean value that defaults to True. If ENABLE_CHIRP is True, and ENABLE_CHIRP_UPDATES is False, then the user job can only read job attributes from the submit side; it cannot change them or write to the job event log. If ENABLE_CHIRP is False, the setting of this variable does not matter, as no Chirp updates are allowed in that case.

ENABLE_CHIRP_IO
A boolean value that defaults to True. If False, the file I/O *condor_chirp* commands are prohibited.

ENABLE_CHIRP_DELAYED
A boolean value that defaults to True. If False, the *condor_chirp* commands **get_job_attr_delayed** and **set_job_attr_delayed** are prohibited.

CHIRP_DELAYED_UPDATE_PREFIX
This is a string-valued and case-insensitive parameter with the default value of "Chirp*". The string is a list separated by spaces and/or commas. Each attribute passed to the either of the *condor_chirp* commands **set_job_attr_delayed** or **get_job_attr_delayed** must match against at least one element in the list. An attribute which does not match any list element fails. A list element may contain a wildcard character ("Chirp*"), which

marks where any number of characters matches. Thus, the default is to allow reads from and writes to only attributes which start with "Chirp".

Because this parameter must be set to the same value on both the submit and execute nodes, it is advised that this parameter not be changed from its built-in default.

CHIRP_DELAYED_UPDATE_MAX_ATTRS
This integer-valued parameter, which defaults to 100, represents the maximum number of pending delayed chirp updates buffered by the *condor_starter*. If the number of unique attributes updated by the *condor_chirp* command **set_job_attr_delayed** exceeds this parameter, it is possible for these updates to be ignored.

USE_PSS
A boolean value, that when True causes the *condor_starter* to measure the PSS (Proportional Set Size) of each HTCondor job. The default value is False. When running many short lived jobs, performance problems in the *condor_procd* have been observed, and a setting of False may relieve these problems.

MEMORY_USAGE_METRIC
A ClassAd expression that produces an initial value for the job ClassAd attribute MemoryUsage in jobs that are not vm universe.

MEMORY_USAGE_METRIC_VM
A ClassAd expression that produces an initial value for the job ClassAd attribute MemoryUsage in vm universe jobs.

STARTER_RLIMIT_AS
An integer ClassAd expression, expressed in MiB, evaluated by the *condor_starter* to set the RLIMIT_AS parameter of the setrlimit() system call. This limits the virtual memory size of each process in the user job. The expression is evaluated in the context of both the machine and job ClassAds, where the machine ClassAd is the MY. ClassAd, and the job ClassAd is the TARGET. ClassAd. There is no default value for this variable. Since values larger than 2047 have no real meaning on 32-bit platforms, values larger than 2047 result in no limit set on 32-bit platforms.

USE_PID_NAMESPACES
A boolean value that, when True, enables the use of per job PID namespaces for HTCondor jobs run on Linux kernels. Defaults to False.

PER_JOB_NAMESPACES
A boolean value that defaults to False. Relevant only for Linux platforms using file system namespaces. The default value of False ensures that there will be no private mount points, because auto mounts done by *autofs* would use the wrong name for private file system mounts. A True value is useful when private file system mounts are permitted and *autofs* (for NFS) is not used.

DYNAMIC_RUN_ACCOUNT_LOCAL_GROUP
For Windows platforms, a value that sets the local group to a group other than the default Users for the condor-slot<X> run account. Do not place the local group name within quotation marks.

JOB_EXECDIR_PERMISSIONS
Control the permissions on the job's scratch directory. Defaults to user which sets permissions to 0700. Possible values are user, group, and world. If set to group, then the directory is group-accessible, with permissions set to 0750. If set to world, then the directory is created with permissions set to 0755.

STARTER_STATS_LOG
The full path and file name of a file that stores TCP statistics for starter file transfers. (Note that the starter logs TCP statistics to this file by default. Adding D_STATS to the STARTER_DEBUG value will cause TCP statistics to be logged to the normal starter log file ($(STARTER_LOG)).) If not defined, STARTER_STATS_LOG defaults to $(LOG)/XferStatsLog. Setting STARTER_STATS_LOG to /dev/null disables logging of starter TCP file transfer statistics.

MAX_STARTER_STATS_LOG
Controls the maximum size in bytes or amount of time that the starter TCP statistics log will be allowed to grow.

If not defined, MAX_STARTER_STATS_LOG defaults to $(MAX_DEFAULT_LOG), which currently defaults to 10 MiB in size. Values are specified with the same syntax as MAX_DEFAULT_LOG.

SINGULARITY
The path to the Singularity binary. The default value is /usr/bin/singularity.

SINGULARITY_JOB
A boolean value specifying whether this startd should run jobs under Singularity. This can be an expression evaluted in the context of the slot ad and the job ad, where the slot ad is the "MY.", and the job ad is the "TARGET.". The default value is False.

SINGULARITY_IMAGE_EXPR
The path to the Singularity container image file. This can be an expression evaluted in the context of the slot ad and the job ad, where the slot ad is the "MY.", and the job ad is the "TARGET.". The default value is "SingularityImage".

SINGULARITY_TARGET_DIR
A directory within the Singularity image to which $_CONDOR_SCRATCH_DIR on the host should be mapped. The default value is "".

SINGULARITY_BIND_EXPR
A string value containing a list of bind mount specifications to be passed to Singularity. This can be an expression evaluted in the context of the slot ad and the job ad, where the slot ad is the "MY.", and the job ad is the "TARGET.". The default value is "SingularityBind".

SINGULARITY_IGNORE_MISSING_BIND_TARGET
A boolean value defaulting to false. If true, and the singularity image is a directory, and the target of a bind mount doesn't exist in the target, then skip this bind mount.

SINGULARITY_USE_PID_NAMESPACES
Controls if jobs using Singularity should run in a private PID namespace, with a default value of Auto. If set to Auto, then PID namespaces will be used if it is possible to do so, else not used. If set to True, then a PID namespaces must be used; if the installed Singularity cannot activate PID namespaces (perhaps due to insufficient permissions), then the slot attribute HasSingularity will be set to False so that jobs needing Singularity will match. If set to False, then PID namespaces must not be used.

SINGULARITY_EXTRA_ARGUMENTS
A string value or classad expression containing a list of extra arguments to be appended to the Singularity command line. This can be an expression evaluted in the context of the slot ad and the job ad, where the slot ad is the "MY.", and the job ad is the "TARGET.".

5.5.11 condor_submit Configuration File Entries

DEFAULT_UNIVERSE
The universe under which a job is executed may be specified in the submit description file. If it is not specified in the submit description file, then this variable specifies the universe (when defined). If the universe is not specified in the submit description file, and if this variable is not defined, then the default universe for a job will be the vanilla universe.

JOB_DEFAULT_NOTIFICATION
The default that sets email notification for jobs. This variable defaults to NEVER, such that HTCondor will not send email about events for jobs. Possible values are NEVER, ERROR, ALWAYS, or COMPLETE. If ALWAYS, the owner will be notified whenever the job completes. If COMPLETE, the owner will be notified when the job terminates. If ERROR, the owner will only be notified if the job terminates abnormally, or if the job is placed on hold because of a failure, and not by user request. If NEVER, the owner will not receive email.

JOB_DEFAULT_LEASE_DURATION
The default value for the **job_lease_duration** submit command when the submit file does not specify a value. The default value is 2400, which is 40 minutes.

JOB_DEFAULT_REQUESTMEMORY
The amount of memory in MiB to acquire for a job, if the job does not specify how much it needs using the **request_memory** submit command. If this variable is not defined, then the default is defined by the expression

```
ifThenElse(MemoryUsage =!= UNDEFINED,MemoryUsage,(ImageSize+1023)/1024)
```

JOB_DEFAULT_REQUESTDISK
The amount of disk in KiB to acquire for a job, if the job does not specify how much it needs using the **request_disk** submit command. If the job defines the value, then that value takes precedence. If not set, then then the default is defined as `DiskUsage`.

JOB_DEFAULT_REQUESTCPUS
The number of CPUs to acquire for a job, if the job does not specify how many it needs using the **request_cpus** submit command. If the job defines the value, then that value takes precedence. If not set, then then the default is 1.

DEFAULT_JOB_MAX_RETRIES
The default value for the maximum number of job retries, if the *condor_submit* retry feature is used. (Note that this value is only relevant if either **retry_until** or **success_exit_code** is defined in the submit file, and **max_retries** is not.) (See the *condor_submit* man page.) The default value if not defined is 2.

If you want *condor_submit* to automatically append an expression to the `Requirements` expression or `Rank` expression of jobs at your site use the following macros:

APPEND_REQ_VANILLA
Expression to be appended to vanilla job requirements.

APPEND_REQUIREMENTS
Expression to be appended to any type of universe jobs. However, if `APPEND_REQ_VANILLA` is defined, then ignore the `APPEND_REQUIREMENTS` for that universe.

APPEND_RANK
Expression to be appended to job rank. `APPEND_RANK_VANILLA` will override this setting if defined.

APPEND_RANK_VANILLA
Expression to append to vanilla job rank.

Note: The `APPEND_RANK_VANILLA` macro was called `APPEND_PREF_VANILLA` in previous versions of HTCondor.

In addition, you may provide default `Rank` expressions if your users do not specify their own with:

DEFAULT_RANK
Default rank expression for any job that does not specify its own rank expression in the submit description file. There is no default value, such that when undefined, the value used will be 0.0.

DEFAULT_RANK_VANILLA
Default rank for vanilla universe jobs. There is no default value, such that when undefined, the value used will be 0.0. When both `DEFAULT_RANK` and `DEFAULT_RANK_VANILLA` are defined, the value for `DEFAULT_RANK_VANILLA` is used for vanilla universe jobs.

SUBMIT_GENERATE_CUSTOM_RESOURCE_REQUIREMENTS
If `True`, *condor_submit* will treat any attribute in the job ClassAd that begins with `Request` as a request for a custom resource and will ad a clause to the Requirements expression insuring that on slots that have that resource will match the job. The default value is `True`.

SUBMIT_SKIP_FILECHECKS
If `True`, *condor_submit* behaves as if the **-disable** command-line option is used. This tells *condor_submit* to disable file permission checks when submitting a job for read permissions on all input files, such as those defined by commands **input** and **transfer_input_files** , as well as write permission to output files, such as a log file defined by **log** and output files defined with **output** or **transfer_output_files** . This can significantly decrease the amount of time required to submit a large group of jobs. The default value is `True`.

WARN_ON_UNUSED_SUBMIT_FILE_MACROS
A boolean variable that defaults to `True`. When `True`, *condor_submit* performs checks on the job's submit description file contents for commands that define a macro, but do not use the macro within the file. A warning is issued, but job submission continues. A definition of a new macro occurs when the lhs of a command is not a known submit command. This check may help spot spelling errors of known submit commands.

SUBMIT_DEFAULT_SHOULD_TRANSFER_FILES
Provides a default value for the submit command **should_transfer_files** if the submit file does not supply a value and when the value is not forced by some other command in the submit file, such as the universe. Valid values are YES, TRUE, ALWAYS, NO, FALSE, NEVER and IF_NEEDED. If the value is not one of these, then IF_NEEDED will be used.

SUBMIT_SEND_RESCHEDULE
A boolean expression that when False, prevents *condor_submit* from automatically sending a *condor_reschedule* command as it completes. The *condor_reschedule* command causes the *condor_schedd* daemon to start searching for machines with which to match the submitted jobs. When True, this step always occurs. In the case that the machine where the job(s) are submitted is managing a huge number of jobs (thousands or tens of thousands), this step would hurt performance in such a way that it became an obstacle to scalability. The default value is True.

SUBMIT_ATTRS
A comma-separated and/or space-separated list of ClassAd attribute names for which the attribute and value will be inserted into all the job ClassAds that *condor_submit* creates. In this way, it is like the "+" syntax in a submit description file. Attributes defined in the submit description file with "+" will override attributes defined in the configuration file with `SUBMIT_ATTRS`. Note that adding an attribute to a job's ClassAd will not function as a method for specifying default values of submit description file commands forgotten in a job's submit description file. The command in the submit description file results in actions by *condor_submit*, while the use of `SUBMIT_ATTRS` adds a job ClassAd attribute at a later point in time.

SUBMIT_ALLOW_GETENV
A boolean attribute which defaults to true. If set to false, the submit command "getenv = true" is an error. Any restricted form of "getenv = some_env_var_name" is still allowed.

LOG_ON_NFS_IS_ERROR
A boolean value that controls whether *condor_submit* prohibits job submit description files with job event log files on NFS. If `LOG_ON_NFS_IS_ERROR` is set to `True`, such submit files will be rejected. If `LOG_ON_NFS_IS_ERROR` is set to `False`, the job will be submitted. If not defined, `LOG_ON_NFS_IS_ERROR` defaults to `False`.

SUBMIT_MAX_PROCS_IN_CLUSTER
An integer value that limits the maximum number of jobs that would be assigned within a single cluster. Job submissions that would exceed the defined value fail, issuing an error message, and with no jobs submitted. The default value is 0, which does not limit the number of jobs assigned a single cluster number.

ENABLE_DEPRECATION_WARNINGS
A boolean value that defaults to `False`. When `True`, *condor_submit* issues warnings when a job requests features that are no longer supported.

INTERACTIVE_SUBMIT_FILE
The path and file name of a submit description file that *condor_submit* will use in the specification of an interactive job. The default is `$(RELEASE_DIR)`/libexec/interactive.sub when not defined.

CRED_MIN_TIME_LEFT
When a job uses an X509 user proxy, condor_submit will refuse to submit a job whose x509 expiration time is less than this many seconds in the future. The default is to only refuse jobs whose expiration time has already passed.

CONTAINER_SHARED_FS
This is a list of strings that name directories which are shared on the execute machines and may contain container images under them. The default value is /cvmfs. When a container universe job lists a *condor_image* that is under one of these directories, HTCondor knows not to try to transfer the file to the worker node.

5.5.12 condor_preen Configuration File Entries

These macros affect *condor_preen*.

PREEN_ADMIN
This macro sets the e-mail address where *condor_preen* will send e-mail (if it is configured to send email at all; see the entry for `PREEN`). Defaults to `$(CONDOR_ADMIN)`.

VALID_SPOOL_FILES
A comma or space separated list of files that *condor_preen* considers valid files to find in the `$(SPOOL)` directory, such that *condor_preen* will not remove these files. There is no default value. *condor_preen* will add to the list files and directories that are normally present in the `$(SPOOL)` directory. A single asterisk (*) wild card character is permitted in each file item within the list.

SYSTEM_VALID_SPOOL_FILES
A comma or space separated list of files that *condor_preen* considers valid files to find in the `$(SPOOL)` directory. The default value is all files known by HTCondor to be valid. This variable exists such that it can be queried; it should not be changed. *condor_preen* use it to initialize the the list files and directories that are normally present in the `$(SPOOL)` directory. A single asterisk (*) wild card character is permitted in each file item within the list.

INVALID_LOG_FILES
This macro contains a (comma or space separated) list of files that *condor_preen* considers invalid files to find in the `$(LOG)` directory. There is no default value.

5.5.13 condor_collector Configuration File Entries

These macros affect the *condor_collector*.

CLASSAD_LIFETIME
The default maximum age in seconds for ClassAds collected by the *condor_collector*. ClassAds older than the maximum age are discarded by the *condor_collector* as stale.

If present, the ClassAd attribute `ClassAdLifetime` specifies the ClassAd's lifetime in seconds. If `ClassAdLifetime` is not present in the ClassAd, the *condor_collector* will use the value of `$(CLASSAD_LIFETIME)`. This variable is defined in terms of seconds, and it defaults to 900 seconds (15 minutes).

To ensure that the *condor_collector* does not miss any ClassAds, the frequency at which all other subsystems that report using an update interval must be tuned. The configuration variables that set these subsystems are

- `UPDATE_INTERVAL` (for the *condor_startd* daemon)
- `NEGOTIATOR_UPDATE_INTERVAL`
- `SCHEDD_INTERVAL`

- MASTER_UPDATE_INTERVAL
- DEFRAG_UPDATE_INTERVAL
- HAD_UPDATE_INTERVAL

COLLECTOR_REQUIREMENTS
A boolean expression that filters out unwanted ClassAd updates. The expression is evaluated for ClassAd updates that have passed through enabled security authorization checks. The default behavior when this expression is not defined is to allow all ClassAd updates to take place. If `False`, a ClassAd update will be rejected.

Stronger security mechanisms are the better way to authorize or deny updates to the *condor_collector*. This configuration variable exists to help those that use host-based security, and do not trust all processes that run on the hosts in the pool. This configuration variable may be used to throw out ClassAds that should not be allowed. For example, for *condor_startd* daemons that run on a fixed port, configure this expression to ensure that only machine ClassAds advertising the expected fixed port are accepted. As a convenience, before evaluating the expression, some basic sanity checks are performed on the ClassAd to ensure that all of the ClassAd attributes used by HTCondor to contain IP:port information are consistent. To validate this information, the attribute to check is `TARGET.MyAddress`.

Please note that _all_ ClassAd updates are filtered. Unless your requirements are the same for all daemons, including the collector itself, you'll want to use the `MyType` attribute to limit your filter(s).

CLIENT_TIMEOUT
Network timeout that the *condor_collector* uses when talking to any daemons or tools that are sending it a ClassAd update. It is defined in seconds and defaults to 30.

QUERY_TIMEOUT
Network timeout when talking to anyone doing a query. It is defined in seconds and defaults to 60.

COLLECTOR_NAME
This macro is used to specify a short description of your pool. It should be about 20 characters long. For example, the name of the UW-Madison Computer Science HTCondor Pool is `"UW-Madison CS"`. While this macro might seem similar to `MASTER_NAME` or `SCHEDD_NAME`, it is unrelated. Those settings are used to uniquely identify (and locate) a specific set of HTCondor daemons, if there are more than one running on the same machine. The `COLLECTOR_NAME` setting is just used as a human-readable string to describe the pool.

COLLECTOR_UPDATE_INTERVAL
This variable is defined in seconds and defaults to 900 (every 15 minutes). It controls the frequency of the periodic updates sent to a central *condor_collector*.

COLLECTOR_SOCKET_BUFSIZE
This specifies the buffer size, in bytes, reserved for *condor_collector* network UDP sockets. The default is 10240000, or a ten megabyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 256000 or 128000).

Note: For some Linux distributions, it may be necessary to raise the OS's system-wide limit for network buffer sizes. The parameter that controls this limit is /proc/sys/net/core/rmem_max. You can see the values that the *condor_collector* actually uses by enabling D_FULLDEBUG for the collector and looking at the log line that looks like this:

Reset OS socket buffer size to 2048k (UDP), 255k (TCP).

For changes to this parameter to take effect, *condor_collector* must be restarted.

COLLECTOR_TCP_SOCKET_BUFSIZE
This specifies the TCP buffer size, in bytes, reserved for *condor_collector* network sockets. The default is 131072,

or a 128 kilobyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 65536 or 32768).

Note: See the note for `COLLECTOR_SOCKET_BUFSIZE` .

KEEP_POOL_HISTORY
This boolean macro is used to decide if the collector will write out statistical information about the pool to history files. The default is `False`. The location, size, and frequency of history logging is controlled by the other macros.

POOL_HISTORY_DIR
This macro sets the name of the directory where the history files reside (if history logging is enabled). The default is the `SPOOL` directory.

POOL_HISTORY_MAX_STORAGE
This macro sets the maximum combined size of the history files. When the size of the history files is close to this limit, the oldest information will be discarded. Thus, the larger this parameter's value is, the larger the time range for which history will be available. The default value is 10000000 (10 MB).

POOL_HISTORY_SAMPLING_INTERVAL
This macro sets the interval, in seconds, between samples for history logging purposes. When a sample is taken, the collector goes through the information it holds, and summarizes it. The information is written to the history file once for each 4 samples. The default (and recommended) value is 60 seconds. Setting this macro's value too low will increase the load on the collector, while setting it to high will produce less precise statistical information.

COLLECTOR_DAEMON_STATS
A boolean value that controls whether or not the *condor_collector* daemon keeps update statistics on incoming updates. The default value is `True`. If enabled, the *condor_collector* will insert several attributes into the ClassAds that it stores and sends. ClassAds without the `UpdateSequenceNumber` and `DaemonStartTime` attributes will not be counted, and will not have attributes inserted (all modern HTCondor daemons which publish ClassAds publish these attributes).

The attributes inserted are `UpdatesTotal`, `UpdatesSequenced`, and `UpdatesLost`. `UpdatesTotal` is the total number of updates (of this ClassAd type) the *condor_collector* has received from this host. `UpdatesSequenced` is the number of updates that the *condor_collector* could have as lost. In particular, for the first update from a daemon, it is impossible to tell if any previous ones have been lost or not. `UpdatesLost` is the number of updates that the *condor_collector* has detected as being lost. See *ClassAd Attributes Added by the condor_collector* for more information on the added attributes.

COLLECTOR_STATS_SWEEP
This value specifies the number of seconds between sweeps of the *condor_collector* 's per-daemon update statistics. Records for daemons which have not reported in this amount of time are purged in order to save memory. The default is two days. It is unlikely that you would ever need to adjust this.

COLLECTOR_DAEMON_HISTORY_SIZE
This variable controls the size of the published update history that the *condor_collector* inserts into the ClassAds it stores and sends. The default value is 128, which means that history is stored and published for the latest 128 updates. This variable's value is ignored, if `COLLECTOR_DAEMON_STATS` is not enabled.

If the value is a non-zero one, the *condor_collector* will insert attribute `UpdatesHistory` into the ClassAd (similar to `UpdatesTotal`). AttrUpdatesHistory is a hexadecimal string which represents a bitmap of the last `COLLECTOR_DAEMON_HISTORY_SIZE` updates. The most significant bit (MSB) of the bitmap represents the most recent update, and the least significant bit (LSB) represents the least recent. A value of zero means that the update was not lost, and a value of 1 indicates that the update was detected as lost.

For example, if the last update was not lost, the previous was lost, and the previous two not, the bitmap would be

0100, and the matching hex digit would be "4". Note that the MSB can never be marked as lost because its loss can only be detected by a non-lost update (a gap is found in the sequence numbers). Thus, UpdatesHistory = "0x40" would be the history for the last 8 updates. If the next updates are all successful, the values published, after each update, would be: 0x20, 0x10, 0x08, 0x04, 0x02, 0x01, 0x00.

See *ClassAd Attributes Added by the condor_collector* for more information on the added attribute.

COLLECTOR_CLASS_HISTORY_SIZE
This variable controls the size of the published update history that the *condor_collector* inserts into the *condor_collector* ClassAds it produces. The default value is zero.

If this variable has a non-zero value, the *condor_collector* will insert UpdatesClassHistory into the *condor_collector* ClassAd (similar to UpdatesHistory). These are added per class of ClassAd, however. The classes refer to the type of ClassAds. Additionally, there is a Total class created, which represents the history of all ClassAds that this *condor_collector* receives.

Note that the *condor_collector* always publishes Lost, Total and Sequenced counts for all ClassAd classes. This is similar to the statistics gathered if COLLECTOR_DAEMON_STATS is enabled.

COLLECTOR_QUERY_WORKERS
This macro sets the maximum number of child worker processes that the *condor_collector* can have, and defaults to a value of 4 on Linux and MacOS platforms. When receiving a large query request, the *condor_collector* may fork() a new process to handle the query, freeing the main process to handle other requests. Each forked child process will consume memory, potentially up to 50% or more of the memory consumed by the parent collector process. To limit the amount of memory consumed on the central manager to handle incoming queries, the default value for this macro is 4. When the number of outstanding worker processes reaches the maximum specified by this macro, any additional incoming query requests will be queued and serviced after an existing child worker completes. Note that on Windows platforms, this macro has a value of zero and cannot be changed.

COLLECTOR_QUERY_WORKERS_RESERVE_FOR_HIGH_PRIO
This macro defines the number of COLLECTOR_QUERY_WORKERS slots will be held in reserve to only service high priority query requests. Currently, high priority queries are defined as those coming from the *condor_negotiator* during the course of matchmaking, or via a "condor_sos condor_status" command. The idea here is the critical operation of matchmaking machines to jobs will take precedence over user condor_status invocations. Defaults to a value of 1. The maximum allowable value for this macro is equal to COLLECTOR_QUERY_WORKERS minus 1.

COLLECTOR_QUERY_WORKERS_PENDING
This macro sets the maximum of collector pending query requests that can be queued waiting for child workers to exit. Queries that would exceed this maximum are immediately aborted. When a forked child worker exits, a pending query will be pulled from the queue for service. Note the collector will confirm that the client has not closed the TCP socket (because it was tired of waiting) before going through all the work of actually forking a child and starting to service the query. Defaults to a value of 50.

COLLECTOR_QUERY_MAX_WORKTIME
This macro defines the maximum amount of time in seconds that a query has to complete before it is aborted. Queries that wait in the pending queue longer than this period of time will be aborted before forking. Queries that have already forked will also abort after the worktime has expired - this protects against clients on a very slow network connection. If set to 0, then there is no timeout. The default is 0.

HANDLE_QUERY_IN_PROC_POLICY
This variable sets the policy for which queries the *condor_collector* should handle in process rather than by forking a worker. It should be set to one of the following values

- always Handle all queries in process
- never Handle all queries using fork workers
- small_table Handle only queries of small tables in process
- small_query Handle only small queries in process

- `small_table_and_query` Handle only small queries on small tables in process
- `small_table_or_query` Handle small queries or small tables in process

A small table is any table of ClassAds in the collector other than Master,Startd,Generic and Any ads. A small query is a locate query, or any query with both a projection and a result limit that is smaller than 10. The default value is `small_table_or_query`.

COLLECTOR_DEBUG
This macro (and other macros related to debug logging in the *condor_collector* is described in .

CONDOR_VIEW_CLASSAD_TYPES
Provides the ClassAd types that will be forwarded to the `CONDOR_VIEW_HOST`. The ClassAd types can be found with *condor_status* **-any**. The default forwarding behavior of the *condor_collector* is equivalent to

```
CONDOR_VIEW_CLASSAD_TYPES=Machine,Submitter
```

There is no default value for this variable.

COLLECTOR_FORWARD_FILTERING
When this boolean variable is set to `True`, Machine and Submitter ad updates are not forwarded to the `CONDOR_VIEW_HOST` if certain attributes are unchanged from the previous update of the ad. The default is `False`, meaning all updates are forwarded.

COLLECTOR_FORWARD_WATCH_LIST
When `COLLECTOR_FORWARD_FILTERING` is set to `True`, this variable provides the list of attributes that controls whether a Machine or Submitter ad update is forwarded to the `CONDOR_VIEW_HOST`. If all attributes in this list are unchanged from the previous update, then the new update is not forwarded. The default value is `State, Cpus,Memory,IdleJobs`.

COLLECTOR_FORWARD_INTERVAL
When `COLLECTOR_FORWARD_FILTERING` is set to `True`, this variable limits how long forwarding of updates for a given ad can be filtered before an update must be forwarded. The default is one third of `CLASSAD_LIFETIME`.

COLLECTOR_FORWARD_CLAIMED_PRIVATE_ADS
When this boolean variable is set to `False`, the *condor_collector* will not forward the private portion of Machine ads to the `CONDOR_VIEW_HOST` if the ad's `State` is `Claimed`. The default value is `$(NEGOTIATOR_CONSIDER_PREEMPTION)`.

COLLECTOR_FORWARD_PROJECTION
An expression that evaluates to a string in the context of an update. The string is treated as a list of attributes to forward. If the string has no attributes, it is ignored. The intended use is to restrict the list of attributes forwarded for claimed Machine ads. When `$(NEGOTIATOR_CONSIDER_PREEMPTION)` is false, the negotiator needs only a few attributes from Machine ads that are in the `Claimed` state. A Suggested use might be

```
if ! $(NEGOTIATOR_CONSIDER_PREEMPTION)
   COLLECTOR_FORWARD_PROJECTION = IfThenElse(State is "Claimed", "$(FORWARD_CLAIMED_
↪ATTRS)", "")
   # forward only the few attributes needed by the Negotiator and a few more needed␣
↪by condor_status
   FORWARD_CLAIMED_ATTRS = Name MyType MyAddress StartdIpAddr Machine Requirements \
      State Activity AccountingGroup Owner RemoteUser SlotWeight ConcurrencyLimits \
      Arch OpSys Memory Cpus CondorLoadAvg EnteredCurrentActivity
endif
```

There is no default value for this variable.

The following macros control where, when, and for how long HTCondor persistently stores absent ClassAds. See section *Absent ClassAds* for more details.

ABSENT_REQUIREMENTS
A boolean expression evaluated by the *condor_collector* when a machine ClassAd would otherwise expire. If `True`, the ClassAd instead becomes absent. If not defined, the implementation will behave as if `False`, and no absent ClassAds will be stored.

ABSENT_EXPIRE_ADS_AFTER
The integer number of seconds after which the *condor_collector* forgets about an absent ClassAd. If 0, the ClassAds persist forever. Defaults to 30 days.

COLLECTOR_PERSISTENT_AD_LOG
The full path and file name of a file that stores machine ClassAds for every hibernating or absent machine. This forms a persistent storage of these ClassAds, in case the *condor_collector* daemon crashes.

To avoid *condor_preen* removing this log, place it in a directory other than the directory defined by `$(SPOOL)`. Alternatively, if this log file is to go in the directory defined by `$(SPOOL)`, add the file to the list given by `VALID_SPOOL_FILES`.

EXPIRE_INVALIDATED_ADS
A boolean value that defaults to `False`. When `True`, causes all invalidated ClassAds to be treated as if they expired. This permits invalidated ClassAds to be marked absent, as defined in *Absent ClassAds*.

5.5.14 condor_negotiator Configuration File Entries

These macros affect the *condor_negotiator*.

NEGOTIATOR_NAME
Used to give an alternative value to the `Name` attribute in the *condor_negotiator* 's ClassAd and the `NegotiatorName` attribute of its accounting ClassAds. This configuration macro is useful in the situation where there are two *condor_negotiator* daemons running on one machine, and both report to the same *condor_collector*. Different names will distinguish the two daemons.

See the description of for defaults and composition of valid HTCondor daemon names.

NEGOTIATOR_INTERVAL
Sets the maximum time the *condor_negotiator* will wait before starting a new negotiation cycle, counting from the start of the previous cycle. It is defined in seconds and defaults to 60 (1 minute).

NEGOTIATOR_MIN_INTERVAL
Sets the minimum time the *condor_negotiator* will wait before starting a new negotiation cycle, counting from the start of the previous cycle. It is defined in seconds and defaults to 5.

NEGOTIATOR_UPDATE_INTERVAL
This macro determines how often the *condor_negotiator* daemon sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 5 minutes).

NEGOTIATOR_CYCLE_DELAY
An integer value that represents the minimum number of seconds that must pass before a new negotiation cycle may start. The default value is 20. `NEGOTIATOR_CYCLE_DELAY` is intended only for use by HTCondor experts.

NEGOTIATOR_TIMEOUT
Sets the timeout that the negotiator uses on its network connections to the *condor_schedd* and *condor_startd* s. It is defined in seconds and defaults to 30.

NEGOTIATION_CYCLE_STATS_LENGTH
Specifies how many recent negotiation cycles should be included in the history that is published in the *condor_negotiator* 's ad. The default is 3 and the maximum allowed value is 100. Setting this value to 0 disables publication of negotiation cycle statistics. The statistics about recent cycles are stored in several attributes per cycle. Each of these attribute names will have a number appended to it to indicate how long ago

the cycle happened, for example: `LastNegotiationCycleDuration0`, `LastNegotiationCycleDuration1`, `LastNegotiationCycleDuration2`, ... The attribute numbered 0 applies to the most recent negotiation cycle. The attribute numbered 1 applies to the next most recent negotiation cycle, and so on. See *Negotiator ClassAd Attributes* for a list of attributes that are published.

NEGOTIATOR_NUM_THREADS
An integer that specifies the number of threads the negotiator should use when trying to match a job to slots. The default is 1. For sites with large number of slots, where the negotiator is running on a large machine, setting this to a larger value may result in faster negotiation times. Setting this to more than the number of cores will result in slow downs. An administrator setting this should also consider what other processes on the machine may need cores, such as the collector, and all of its forked children, the condor_master, and any helper programs or scripts running there.

PRIORITY_HALFLIFE
This macro defines the half-life of the user priorities. See *User priority* on User Priorities for details. It is defined in seconds and defaults to 86400 (1 day).

DEFAULT_PRIO_FACTOR
Sets the priority factor for local users as they first submit jobs, as described in *User Priorities and Negotiation*. Defaults to 1000.

NICE_USER_PRIO_FACTOR
Sets the priority factor for nice users, as described in *User Priorities and Negotiation*. Defaults to 10000000000.

NICE_USER_ACCOUNTING_GROUP_NAME
Sets the name used for the nice-user accounting group by *condor_submit*. Defaults to nice-user.

REMOTE_PRIO_FACTOR
Defines the priority factor for remote users, which are those users who who do not belong to the local domain. See *User Priorities and Negotiation* for details. Defaults to 10000000.

ACCOUNTANT_DATABASE_FILE
Defines the full path of the accountant database log file. The default value is `$(SPOOL)/Accountantnew.log`

ACCOUNTANT_LOCAL_DOMAIN
Describes the local UID domain. This variable is used to decide if a user is local or remote. A user is considered to be in the local domain if their UID domain matches the value of this variable. Usually, this variable is set to the local UID domain. If not defined, all users are considered local.

MAX_ACCOUNTANT_DATABASE_SIZE
This macro defines the maximum size (in bytes) that the accountant database log file can reach before it is truncated (which re-writes the file in a more compact format). If, after truncating, the file is larger than one half the maximum size specified with this macro, the maximum size will be automatically expanded. The default is 1 megabyte (1000000).

NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES
This macro tells the negotiator to not count resources that are suspended when calculating the number of resources a user is using. Defaults to false, that is, a user is still charged for a resource even when that resource has suspended the job.

NEGOTIATOR_SOCKET_CACHE_SIZE
This macro defines the maximum number of sockets that the *condor_negotiator* keeps in its open socket cache. Caching open sockets makes the negotiation protocol more efficient by eliminating the need for socket connection establishment for each negotiation cycle. The default is currently 500. To be effective, this parameter should be set to a value greater than the number of *condor_schedd* s submitting jobs to the negotiator at any time. If you lower this number, you must run *condor_restart* and not just *condor_reconfig* for the change to take effect.

NEGOTIATOR_INFORM_STARTD
Boolean setting that controls if the *condor_negotiator* should inform the *condor_startd* when it has been matched with a job. The default is `False`. When this is set to the default value of `False`, the *condor_startd* will never

enter the Matched state, and will go directly from Unclaimed to Claimed. Because this notification is done via UDP, if a pool is configured so that the execute hosts do not create UDP command sockets (see the setting for details), the *condor_negotiator* should be configured not to attempt to contact these *condor_startd* daemons by using the default value.

NEGOTIATOR_PRE_JOB_RANK
Resources that match a request are first sorted by this expression. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by NEGOTIATOR_POST_JOB_RANK, then by PREEMPTION_RANK (if the match would cause preemption and there are still any ties in the top choice). MY refers to attributes of the machine ClassAd and TARGET refers to the job ClassAd. The purpose of the pre job rank is to allow the pool administrator to override any other rankings, in order to optimize overall throughput. For example, it is commonly used to minimize preemption, even if the job rank prefers a machine that is busy. If explicitly set to be undefined, this expression has no effect on the ranking of matches. The default value prefers to match multi-core jobs to dynamic slots in a best fit manner:

```
NEGOTIATOR_PRE_JOB_RANK = (10000000 * My.Rank) + \
 (1000000 * (RemoteOwner =?= UNDEFINED)) - (100000 * Cpus) - Memory
```

NEGOTIATOR_POST_JOB_RANK
Resources that match a request are first sorted by NEGOTIATOR_PRE_JOB_RANK. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by NEGOTIATOR_POST_JOB_RANK, then by PREEMPTION_RANK (if the match would cause preemption and there are still any ties in the top choice). MY. refers to attributes of the machine ClassAd and TARGET. refers to the job ClassAd. The purpose of the post job rank is to allow the pool administrator to choose between machines that the job ranks equally. The default value is

```
NEGOTIATOR_POST_JOB_RANK = \
 (RemoteOwner =?= UNDEFINED) * \
 (ifThenElse(isUndefined(KFlops), 1000, Kflops) - \
 SlotID - 1.0e10*(Offline=?=True))
```

PREEMPTION_REQUIREMENTS
When considering user priorities, the negotiator will not preempt a job running on a given machine unless this expression evaluates to True, and the owner of the idle job has a better priority than the owner of the running job. The PREEMPTION_REQUIREMENTS expression is evaluated within the context of the candidate machine ClassAd and the candidate idle job ClassAd; thus the MY scope prefix refers to the machine ClassAd, and the TARGET scope prefix refers to the ClassAd of the idle (candidate) job. There is no direct access to the currently running job, but attributes of the currently running job that need to be accessed in PREEMPTION_REQUIREMENTS can be placed in the machine ClassAd using STARTD_JOB_ATTRS . If not explicitly set in the HTCondor configuration file, the default value for this expression is False. PREEMPTION_REQUIREMENTS should include the term (SubmitterGroup =?= RemoteGroup), if a preemption policy that respects group quotas is desired. Note that this variable does not influence other potential causes of preemption, such as the RANK of the *condor_startd*, or PREEMPT expressions. See *condor_startd Policy Configuration* for a general discussion of limiting preemption.

PREEMPTION_REQUIREMENTS_STABLE
A boolean value that defaults to True, implying that all attributes utilized to define the PREEMPTION_REQUIREMENTS variable will not change within a negotiation period time interval. If utilized attributes will change during the negotiation period time interval, then set this variable to False.

PREEMPTION_RANK
Resources that match a request are first sorted by NEGOTIATOR_PRE_JOB_RANK. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by NEGOTIATOR_POST_JOB_RANK, then by PREEMPTION_RANK (if the match would cause preemption and there are still any ties in the top choice). MY refers to attributes of the machine ClassAd and TARGET refers to the job ClassAd. This expression is used to rank machines that the job and the other negotiation expressions rank the same. For example, if the job has no preference, it is usually preferable to preempt a job with a small ImageSize

instead of a job with a large ImageSize. The default value first considers the user's priority and chooses the user with the worst priority. Then, among the running jobs of that user, it chooses the job with the least accumulated run time:

```
PREEMPTION_RANK = (RemoteUserPrio * 1000000) - \
 ifThenElse(isUndefined(TotalJobRunTime), 0, TotalJobRunTime)
```

PREEMPTION_RANK_STABLE
A boolean value that defaults to True, implying that all attributes utilized to define the PREEMPTION_RANK variable will not change within a negotiation period time interval. If utilized attributes will change during the negotiation period time interval, then set this variable to False.

NEGOTIATOR_SLOT_CONSTRAINT
An expression which constrains which machine ClassAds are fetched from the *condor_collector* by the *condor_negotiator* during a negotiation cycle.

NEGOTIATOR_SUBMITTER_CONSTRAINT
An expression which constrains which submitter ClassAds are fetched from the *condor_collector* by the *condor_negotiator* during a negotiation cycle. The *condor_negotiator* will ignore the jobs of submitters whose submitter ads don't match this constraint.

NEGOTIATOR_JOB_CONSTRAINT
An expression which constrains which job ClassAds are considered for matchmaking by the *condor_negotiator*. This parameter is read by the *condor_negotiator* and sent to the *condor_schedd* for evaluation. *condor_schedd* s older than version 8.7.7 will ignore this expression and so will continue to send all jobs to the *condor_negotiator*.

NEGOTIATOR_TRIM_SHUTDOWN_THRESHOLD
This setting is not likely to be customized, except perhaps within a glidein setting. An integer expression that evaluates to a value within the context of the *condor_negotiator* ClassAd, with a default value of 0. When this expression evaluates to an integer X greater than 0, the *condor_negotiator* will not make matches to machines that contain the ClassAd attribute DaemonShutdown which evaluates to True, when that shut down time is X seconds into the future. The idea here is a mechanism to prevent matching with machines that are quite close to shutting down, since the match would likely be a waste of time.

NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT or GROUP_DYNAMIC_MACH_CONSTRAINT
This optional expression specifies which machine ClassAds should be counted when computing the size of the pool. It applies both for group quota allocation and when there are no groups. The default is to count all machine ClassAds. When extra slots exist for special purposes, as, for example, suspension slots or file transfer slots, this expression can be used to inform the *condor_negotiator* that only normal slots should be counted when computing how big each group's share of the pool should be.

The name NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT replaces GROUP_DYNAMIC_MACH_CONSTRAINT as of HTCondor version 7.7.3. Using the older name causes a warning to be logged, although the behavior is unchanged.

NEGOTIATOR_DEBUG
This macro (and other settings related to debug logging in the negotiator) is described in .

NEGOTIATOR_MAX_TIME_PER_SUBMITTER
The maximum number of seconds the *condor_negotiator* will spend with each individual submitter during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will skip over requests from this submitter until the next negotiation cycle. It defaults to 60 seconds.

NEGOTIATOR_MAX_TIME_PER_SCHEDD
The maximum number of seconds the *condor_negotiator* will spend with each individual *condor_schedd* during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will skip over requests from this *condor_schedd* until the next negotiation cycle. It defaults to 120 seconds.

NEGOTIATOR_MAX_TIME_PER_CYCLE
The maximum number of seconds the *condor_negotiator* will spend in total across all submitters during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will skip over requests from all submitters until the next negotiation cycle. It defaults to 1200 seconds.

NEGOTIATOR_MAX_TIME_PER_PIESPIN
The maximum number of seconds the *condor_negotiator* will spend with a submitter in one pie spin. A negotiation cycle is composed of at least one pie spin, possibly more, depending on whether there are still machines left over after computing fair shares and negotiating with each submitter. By limiting the maximum length of a pie spin or the maximum time per submitter per negotiation cycle, the *condor_negotiator* is protected against spending a long time talking to one submitter, for example someone with a very slow *condor_schedd* daemon. But, this can result in unfair allocation of machines or some machines not being allocated at all. See *User Priorities and Negotiation* for a description of a pie slice. It defaults to 120 seconds.

NEGOTIATOR_DEPTH_FIRST
A boolean value which defaults to false. When partitionable slots are enabled, and this parameter is true, the negotiator tries to pack as many jobs as possible on each machine before moving on to the next machine.

USE_RESOURCE_REQUEST_COUNTS
A boolean value that defaults to `True`. When `True`, the latency of negotiation will be reduced when there are many jobs next to each other in the queue with the same auto cluster, and many matches are being made. When `True`, the *condor_schedd* tells the *condor_negotiator* to send X matches at a time, where X equals number of consecutive jobs in the queue within the same auto cluster.

NEGOTIATOR_RESOURCE_REQUEST_LIST_SIZE
An integer tuning parameter used by the *condor_negotiator* to control the number of resource requests fetched from a *condor_schedd* per network round-trip. With higher values, the latency of negotiation can be significantly be reduced when negotiating with a *condor_schedd* running HTCondor version 8.3.0 or more recent, especially over a wide-area network. Setting this value too high, however, could cause the *condor_schedd* to unnecessarily block on network I/O. The default value is 200. If `USE_RESOURCE_REQUEST_COUNTS` is set to `False`, then this variable will be unconditionally set to a value of 1.

NEGOTIATOR_MATCH_EXPRS
A comma-separated list of macro names that are inserted as ClassAd attributes into matched job ClassAds. The attribute name in the ClassAd will be given the prefix `NegotiatorMatchExpr`, if the macro name does not already begin with that. Example:

```
NegotiatorName = "My Negotiator"
NEGOTIATOR_MATCH_EXPRS = NegotiatorName
```

As a result of the above configuration, jobs that are matched by this *condor_negotiator* will contain the following attribute when they are sent to the *condor_startd*:

```
NegotiatorMatchExprNegotiatorName = "My Negotiator"
```

The expressions inserted by the *condor_negotiator* may be useful in *condor_startd* policy expressions, when the *condor_startd* belongs to multiple HTCondor pools.

NEGOTIATOR_MATCHLIST_CACHING
A boolean value that defaults to `True`. When `True`, it enables an optimization in the *condor_negotiator* that works with auto clustering. In determining the sorted list of machines that a job might use, the job goes to the first machine off the top of the list. If `NEGOTIATOR_MATCHLIST_CACHING` is `True`, and if the next job is part of the same auto cluster, meaning that it is a very similar job, the *condor_negotiator* will reuse the previous list of machines, instead of recreating the list from scratch.

NEGOTIATOR_CONSIDER_PREEMPTION
For expert users only. A boolean value that defaults to `True`. When `False`, it can cause the *condor_negotiator*

to run faster and also have better spinning pie accuracy. Only set this to `False` if `PREEMPTION_REQUIREMENTS` is `False`, and if all *condor_startd* rank expressions are `False`.

NEGOTIATOR_CONSIDER_EARLY_PREEMPTION
A boolean value that when `False` (the default), prevents the *condor_negotiator* from matching jobs to claimed slots that cannot immediately be preempted due to `MAXJOBRETIREMENTTIME` .

ALLOW_PSLOT_PREEMPTION
A boolean value that defaults to `False`. When set to `True` for the *condor_negotiator*, it enables a new matchmaking mode in which one or more dynamic slots can be preempted in order to make enough resources available in their parent partitionable slot for a job to successfully match to the partitionable slot.

STARTD_AD_REEVAL_EXPR
A boolean value evaluated in the context of each machine ClassAd within a negotiation cycle that determines whether the ClassAd from the *condor_collector* is to replace the stashed ClassAd utilized during the previous negotiation cycle. When `True`, the ClassAd from the *condor_collector* does replace the stashed one. When not defined, the default value is to replace the stashed ClassAd if the stashed ClassAd's sequence number is older than its potential replacement.

NEGOTIATOR_UPDATE_AFTER_CYCLE
A boolean value that defaults to `False`. When `True`, it will force the *condor_negotiator* daemon to publish an update to the *condor_collector* at the end of every negotiation cycle. This is useful if monitoring statistics for the previous negotiation cycle.

NEGOTIATOR_READ_CONFIG_BEFORE_CYCLE
A boolean value that defaults to `False`. When `True`, the *condor_negotiator* will re-read the configuration prior to beginning each negotiation cycle. Note that this operation will update configured behaviors such as concurrency limits, but not data structures constructed during a full reconfiguration, such as the group quota hierarchy. A full reconfiguration, for example as accomplished with *condor_reconfig*, remains the best way to guarantee that all *condor_negotiator* configuration is completely updated.

<NAME>_LIMIT
An integer value that defines the amount of resources available for jobs which declare that they use some consumable resource as described in *Concurrency Limits*. `<Name>` is a string invented to uniquely describe the resource.

CONCURRENCY_LIMIT_DEFAULT
An integer value that describes the number of resources available for any resources that are not explicitly named defined with the configuration variable `<NAME>_LIMIT`. If not defined, no limits are set for resources not explicitly identified using `<NAME>_LIMIT`.

CONCURRENCY_LIMIT_DEFAULT_<NAME>
If set, this defines a default concurrency limit for all resources that start with `<NAME>.`

The following configuration macros affect negotiation for group users.

GROUP_NAMES
A comma-separated list of the recognized group names, case insensitive. If undefined (the default), group support is disabled. Group names must not conflict with any user names. That is, if there is a physics group, there may not be a physics user. Any group that is defined here must also have a quota, or the group will be ignored. Example:

```
GROUP_NAMES = group_physics, group_chemistry
```

GROUP_QUOTA_<groupname>
A floating point value to represent a static quota specifying an integral number of machines for the hierarchical group identified by `<groupname>`. It is meaningless to specify a non integer value, since only integral numbers of machines can be allocated. Example:

```
GROUP_QUOTA_group_physics = 20
GROUP_QUOTA_group_chemistry = 10
```

When both static and dynamic quotas are defined for a specific group, the static quota is used and the dynamic quota is ignored.

GROUP_QUOTA_DYNAMIC_<groupname>
A floating point value in the range 0.0 to 1.0, inclusive, representing a fraction of a pool's machines (slots) set as a dynamic quota for the hierarchical group identified by <groupname>. For example, the following specifies that a quota of 25% of the total machines are reserved for members of the group_biology group.

```
GROUP_QUOTA_DYNAMIC_group_biology = 0.25
```

The group name must be specified in the GROUP_NAMES list.

This section has not yet been completed

GROUP_PRIO_FACTOR_<groupname>
A floating point value greater than or equal to 1.0 to specify the default user priority factor for <groupname>. The group name must also be specified in the GROUP_NAMES list. GROUP_PRIO_FACTOR_<groupname> is evaluated when the negotiator first negotiates for the user as a member of the group. All members of the group inherit the default priority factor when no other value is present. For example, the following setting specifies that all members of the group named group_physics inherit a default user priority factor of 2.0:

```
GROUP_PRIO_FACTOR_group_physics = 2.0
```

GROUP_AUTOREGROUP
A boolean value (defaults to False) that when True, causes users who submitted to a specific group to also negotiate a second time with the <none> group, to be considered with the independent job submitters. This allows group submitted jobs to be matched with idle machines even if the group is over its quota. The user name that is used for accounting and prioritization purposes is still the group user as specified by AccountingGroup in the job ClassAd.

GROUP_AUTOREGROUP_<groupname>
This is the same as GROUP_AUTOREGROUP, but it is settable on a per-group basis. If no value is specified for a given group, the default behavior is determined by GROUP_AUTOREGROUP, which in turn defaults to False.

GROUP_ACCEPT_SURPLUS
A boolean value that, when True, specifies that groups should be allowed to use more than their configured quota when there is not enough demand from other groups to use all of the available machines. The default value is False.

GROUP_ACCEPT_SURPLUS_<groupname>
A boolean value applied as a group-specific version of GROUP_ACCEPT_SURPLUS. When not specified, the value of GROUP_ACCEPT_SURPLUS applies to the named group.

GROUP_QUOTA_ROUND_ROBIN_RATE
The maximum sum of weighted slots that should be handed out to an individual submitter in each iteration within a negotiation cycle. If slot weights are not being used by the *condor_negotiator*, as specified by NEGOTIATOR_USE_SLOT_WEIGHTS = False, then this value is just the (unweighted) number of slots. The default value is a very big number, effectively infinite. Setting the value to a number smaller than the size of the pool can help avoid starvation. An example of the starvation problem is when there are a subset of machines in a pool with large memory, and there are multiple job submitters who desire all of these machines. Normally, HTCondor will decide how much of the full pool each person should get, and then attempt to hand out that number of resources to each person. Since the big memory machines are only a subset of pool, it may happen that they are all given to the first person contacted, and the remainder requiring large memory machines get nothing.

Setting GROUP_QUOTA_ROUND_ROBIN_RATE to a value that is small compared to the size of subsets of machines will reduce starvation at the cost of possibly slowing down the rate at which resources are allocated.

GROUP_QUOTA_MAX_ALLOCATION_ROUNDS
An integer that specifies the maximum number of times within one negotiation cycle the *condor_negotiator* will calculate how many slots each group deserves and attempt to allocate them. The default value is 3. The reason it may take more than one round is that some groups may not have jobs that match some of the available machines, so some of the slots that were withheld for those groups may not get allocated in any given round.

NEGOTIATOR_USE_SLOT_WEIGHTS
A boolean value with a default of True. When True, the *condor_negotiator* pays attention to the machine ClassAd attribute SlotWeight. When False, each slot effectively has a weight of 1.

NEGOTIATOR_USE_WEIGHTED_DEMAND
A boolean value that defaults to True. When False, the behavior is the same as for HTCondor versions prior to 7.9.6. If True, when the *condor_schedd* advertises IdleJobs in the submitter ClassAd, which represents the number of idle jobs in the queue for that submitter, it will also advertise the total number of requested cores across all idle jobs from that submitter, WeightedIdleJobs. If partitionable slots are being used, and if hierarchical group quotas are used, and if any hierarchical group quotas set GROUP_ACCEPT_SURPLUS to True, and if configuration variable SlotWeight is set to the number of cores, then setting this configuration variable to True allows the amount of surplus allocated to each group to be calculated correctly.

GROUP_SORT_EXPR
A floating point ClassAd expression that controls the order in which the *condor_negotiator* considers groups when allocating resources. The smallest magnitude positive value goes first. The default value is set such that group <none> always goes last when considering group quotas, and groups are considered in starvation order (the group using the smallest fraction of its resource quota is considered first).

NEGOTIATOR_ALLOW_QUOTA_OVERSUBSCRIPTION
A boolean value that defaults to True. When True, the behavior of resource allocation when considering groups is more like it was in the 7.4 stable series of HTCondor. In implementation, when True, the static quotas of subgroups will not be scaled when the sum of these static quotas of subgroups sums to more than the group's static quota. This behavior is desirable when using static quotas, unless the sum of subgroup quotas is considerably less than the group's quota, as scaling is currently based on the number of machines available, not assigned quotas (for static quotas).

5.5.15 condor_procd Configuration File Macros

USE_PROCD
This boolean variable determines whether the *condor_procd* will be used for managing process families. If the *condor_procd* is not used, each daemon will run the process family tracking logic on its own. Use of the *condor_procd* results in improved scalability because only one instance of this logic is required. The *condor_procd* is required when using group ID-based process tracking (see *Group ID-Based Process Tracking*. In this case, the USE_PROCD setting will be ignored and a *condor_procd* will always be used. By default, the *condor_master* will start a *condor_procd* that all other daemons that need process family tracking will use. A daemon that uses the *condor_procd* will start a *condor_procd* for use by itself and all of its child daemons.

PROCD_MAX_SNAPSHOT_INTERVAL
This setting determines the maximum time that the *condor_procd* will wait between probes of the system for information about the process families it is tracking.

PROCD_LOG
Specifies a log file for the *condor_procd* to use. Note that by design, the *condor_procd* does not include most of the other logic that is shared amongst the various HTCondor daemons. This means that the *condor_procd* does not include the normal HTCondor logging subsystem, and thus multiple debug levels are not supported.

PROCD_LOG defaults to $(LOG)/ProcLog. Note that enabling D_PROCFAMILY in the debug level for any other daemon will cause it to log all interactions with the *condor_procd*.

MAX_PROCD_LOG
Controls the maximum length in bytes to which the *condor_procd* log will be allowed to grow. The log file will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging will be twice the maximum length of this log file. A value of 0 specifies that the file may grow without bounds. The default is 10 MiB.

PROCD_ADDRESS
This specifies the address that the *condor_procd* will use to receive requests from other HTCondor daemons. On Unix, this should point to a file system location that can be used for a named pipe. On Windows, named pipes are also used but they do not exist in the file system. The default setting is $(RUN)/procd_pipe on Unix and \\.\pipe\procd_pipe on Windows.

USE_GID_PROCESS_TRACKING
A boolean value that defaults to False. When True, a job's initial process is assigned a dedicated GID which is further used by the *condor_procd* to reliably track all processes associated with a job. When True, values for MIN_TRACKING_GID and MAX_TRACKING_GID must also be set, or HTCondor will abort, logging an error message. See *Group ID-Based Process Tracking* for a detailed description.

MIN_TRACKING_GID
An integer value, that together with MAX_TRACKING_GID specify a range of GIDs to be assigned on a per slot basis for use by the *condor_procd* in tracking processes associated with a job. See *Group ID-Based Process Tracking* for a detailed description.

MAX_TRACKING_GID
An integer value, that together with MIN_TRACKING_GID specify a range of GIDs to be assigned on a per slot basis for use by the *condor_procd* in tracking processes associated with a job. See *Group ID-Based Process Tracking* for a detailed description.

BASE_CGROUP
The path to the directory used as the virtual file system for the implementation of Linux kernel cgroups. This variable defaults to the string htcondor, and is only used on Linux systems. To disable cgroup tracking, define this to an empty string. See *Cgroup-Based Process Tracking* for a description of cgroup-based process tracking. An administrator can configure distinct cgroup roots for different slot types within the same startd by prefixing the *BASE_CGROUP* macro with the slot type. e.g. setting SLOT_TYPE_1.BASE_CGROUP = hiprio_cgroup and SLOT_TYPE_2.BASE_CGROUP = low_prio

5.5.16 condor_credd Configuration File Macros

These macros affect the *condor_credd* and its credmon plugin.

CREDD_HOST
The host name of the machine running the *condor_credd* daemon.

CREDD_POLLING_TIMEOUT
An integer value representing the number of seconds that the *condor_credd*, *condor_starter*, and *condor_schedd* daemons will wait for valid credentials to be produced by a credential monitor (CREDMON) service. The default value is 20.

CREDD_CACHE_LOCALLY
A boolean value that defaults to False. When True, the first successful password fetch operation to the *condor_credd* daemon causes the password to be stashed in a local, secure password store. Subsequent uses of that password do not require communication with the *condor_credd* daemon.

CRED_SUPER_USERS
A comma and/or space separated list of user names on a given machine that are permitted to store credentials for any user when using the *condor_store_cred* command. When not on this list, users can only store their own credentials. Entries in this list can contain a single '*' wildcard character, which matches any sequence of characters.

SKIP_WINDOWS_LOGON_NETWORK
A boolean value that defaults to `False`. When `True`, Windows authentication skips trying authentication with the `LOGON_NETWORK` method first, and attempts authentication with `LOGON_INTERACTIVE` method. This can be useful if many authentication failures are noticed, potentially leading to users getting locked out.

CREDMON_KRB
The path to the credmon daemon process when using the Kerberos credentials type. The default is /usr/sbin/condor_credmon_krb

CREDMON_OAUTH
The path to the credmon daemon process when using the OAuth2 credentials type. The default is /usr/sbin/condor_credmon_oauth.

CREDMON_OAUTH_TOKEN_MINIMUM
The minimum time in seconds that OAuth2 tokens should have remaining on them when they are generated. The default is 40 minutes. This is currently implemented only in the vault credmon, not the default oauth credmon.

CREDMON_OAUTH_TOKEN_REFRESH
The time in seconds between renewing OAuth2 tokens. The default is half of `CREDMON_OAUTH_TOKEN_MINIMUM`. This is currently implemented only in the vault credmon, not the default oauth credmon.

5.5.17 condor_gridmanager Configuration File Entries

These macros affect the *condor_gridmanager*.

GRIDMANAGER_LOG
Defines the path and file name for the log of the *condor_gridmanager*. The owner of the file is the condor user.

GRIDMANAGER_CHECKPROXY_INTERVAL
The number of seconds between checks for an updated X509 proxy credential. The default is 10 minutes (600 seconds).

GRIDMANAGER_PROXY_REFRESH_TIME
For remote schedulers that allow for X.509 proxy refresh, the *condor_gridmanager* will not forward a refreshed proxy until the lifetime left for the proxy on the remote machine falls below this value. The value is in seconds and the default is 21600 (6 hours).

GRIDMANAGER_MINIMUM_PROXY_TIME
The minimum number of seconds before expiration of the X509 proxy credential for the gridmanager to continue operation. If seconds until expiration is less than this number, the gridmanager will shutdown and wait for a refreshed proxy credential. The default is 3 minutes (180 seconds).

HOLD_JOB_IF_CREDENTIAL_EXPIRES
True or False. Defaults to True. If True, and for grid universe jobs only, HTCondor-G will place a job on hold `GRIDMANAGER_MINIMUM_PROXY_TIME` seconds before the proxy expires. If False, the job will stay in the last known state, and HTCondor-G will periodically check to see if the job's proxy has been refreshed, at which point management of the job will resume.

GRIDMANAGER_SELECTION_EXPR
By default, the gridmanager operates on a per-`Owner` basis. That is, the *condor_schedd* starts a distinct *condor_gridmanager* for each grid universe job with a distinct `Owner`. For additional isolation and/or scalability,

you may set this macro to a ClassAd expression. It will be evaluated against each grid universe job, and jobs with the same evaluated result will go to the same gridmanager. For instance, if you want to isolate job going to different remote sites from each other, the following expression works:

```
GRIDMANAGER_SELECTION_EXPR = GridResource
```

GRIDMANAGER_LOG_APPEND_SELECTION_EXPR
A boolean value that defaults to `False`. When `True`, the evaluated value of `GRIDMANAGER_SELECTION_EXPR` (if set) is appended to the value of `GRIDMANAGER_LOG` for each *condor_gridmanager* instance. The value is sanitized to remove characters that have special meaning to the shell. This allows each *condor_gridmanager* instance that runs concurrently to write to a separate daemon log.

GRIDMANAGER_CONTACT_SCHEDD_DELAY
The minimum number of seconds between connections to the *condor_schedd*. The default is 5 seconds.

GRIDMANAGER_JOB_PROBE_INTERVAL
The number of seconds between active probes for the status of a submitted job. The default is 1 minute (60 seconds). Intervals specific to grid types can be set by appending the name of the grid type to the configuration variable name, as the example

```
GRIDMANAGER_JOB_PROBE_INTERVAL_ARC = 300
```

GRIDMANAGER_JOB_PROBE_RATE
The maximum number of job status probes per second that will be issued to a given remote resource. The time between status probes for individual jobs may be lengthened beyond `GRIDMANAGER_JOB_PROBE_INTERVAL` to enforce this rate. The default is 5 probes per second. Rates specific to grid types can be set by appending the name of the grid type to the configuration variable name, as the example

```
GRIDMANAGER_JOB_PROBE_RATE_ARC = 15
```

GRIDMANAGER_RESOURCE_PROBE_INTERVAL
When a resource appears to be down, how often (in seconds) the *condor_gridmanager* should ping it to test if it is up again. The default is 5 minutes (300 seconds).

GRIDMANAGER_EMPTY_RESOURCE_DELAY
The number of seconds that the *condor_gridmanager* retains information about a grid resource, once the *condor_gridmanager* has no active jobs on that resource. An active job is a grid universe job that is in the queue, for which `JobStatus` is anything other than Held. Defaults to 300 seconds.

GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE
An integer value that limits the number of jobs that a *condor_gridmanager* daemon will submit to a resource. A comma-separated list of pairs that follows this integer limit will specify limits for specific remote resources. Each pair is a host name and the job limit for that host. Consider the example:

```
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE = 200, foo.edu, 50, bar.com, 100
```

In this example, all resources have a job limit of 200, except foo.edu, which has a limit of 50, and bar.com, which has a limit of 100.

Limits specific to grid types can be set by appending the name of the grid type to the configuration variable name, as the example

```
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE_PBS = 300
```

In this example, the job limit for all PBS resources is 300. Defaults to 1000.

GAHP_DEBUG_HIDE_SENSITIVE_DATA
A boolean value that determines when sensitive data such as security keys and passwords are hidden, when

communication to or from a GAHP server is written to a daemon log. The default is `True`, hiding sensitive data.

GRIDMANAGER_GAHP_CALL_TIMEOUT
The number of seconds after which a pending GAHP command should time out. The default is 5 minutes (300 seconds).

GRIDMANAGER_GAHP_RESPONSE_TIMEOUT
The *condor_gridmanager* will assume a GAHP is hung if this many seconds pass without a response. The default is 20.

GRIDMANAGER_MAX_PENDING_REQUESTS
The maximum number of GAHP commands that can be pending at any time. The default is 50.

GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT
The number of times to retry a command that failed due to a timeout or a failed connection. The default is 3.

EC2_RESOURCE_TIMEOUT
The number of seconds after which if an EC2 grid universe job fails to ping the EC2 service, the job will be put on hold. Defaults to -1, which implements an infinite length, such that a failure to ping the service will never put the job on hold.

EC2_GAHP_RATE_LIMIT
The minimum interval, in whole milliseconds, between requests to the same EC2 service with the same credentials. Defaults to 100.

BATCH_GAHP_CHECK_STATUS_ATTEMPTS
The number of times a failed status command issued to the *blahpd* should be retried. These retries allow the *condor_gridmanager* to tolerate short-lived failures of the underlying batch system. The default value is 5.

C_GAHP_LOG
The complete path and file name of the HTCondor GAHP server's log. The default value is `/tmp/CGAHPLog.$(USERNAME)`.

MAX_C_GAHP_LOG
The maximum size of the `C_GAHP_LOG`.

C_GAHP_WORKER_THREAD_LOG
The complete path and file name of the HTCondor GAHP worker process' log. The default value is `/temp/CGAHPWorkerLog.$(USERNAME)`.

C_GAHP_CONTACT_SCHEDD_DELAY
The number of seconds that the *condor_C-gahp* daemon waits between consecutive connections to the remote *condor_schedd* in order to send batched sets of commands to be executed on that remote *condor_schedd* daemon. The default value is 5.

C_GAHP_MAX_FILE_REQUESTS
Limits the number of file transfer commands of each type (input, output, proxy refresh) that are performed before other (potentially higher-priority) commands are read and performed. The default value is 10.

BLAHPD_LOCATION
The complete path to the directory containing the *blahp* software, which is required for grid-type batch jobs. The default value is `$(RELEASE_DIR)`.

GAHP_SSL_CADIR
The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs to be used by GAHP servers when authenticating with remote services.

GAHP_SSL_CAFILE
The path and file name of a file containing one or more trusted CA's certificates to be used by GAHP servers when authenticating with remote services.

CONDOR_GAHP
The complete path and file name of the HTCondor GAHP executable. The default value is `$(SBIN)`/condor_c-gahp.

EC2_GAHP
The complete path and file name of the EC2 GAHP executable. The default value is `$(SBIN)`/ec2_gahp.

BATCH_GAHP
The complete path and file name of the batch GAHP executable, to be used for Slurm, PBS, LSF, SGE, and similar batch systems. The default location is `$(BIN)`/blahpd.

ARC_GAHP
The complete path and file name of the ARC GAHP executable. The default value is `$(SBIN)`/arc_gahp.

ARC_GAHP_COMMAND_LIMIT
On systems where libcurl uses NSS for security, start a new *arc_gahp* process when the existing one has handled the given number of commands. The default is 1000.

ARC_GAHP_USE_THREADS
Controls whether the *arc_gahp* should run multiple HTTPS requests in parallel in different threads. The default is `False`.

GCE_GAHP
The complete path and file name of the GCE GAHP executable. The default value is `$(SBIN)`/gce_gahp.

AZURE_GAHP
The complete path and file name of the Azure GAHP executable. The default value is `$(SBIN)`/AzureGAHPServer.py on Windows and `$(SBIN)`/AzureGAHPServer on other platforms.

BOINC_GAHP
The complete path and file name of the BOINC GAHP executable. The default value is `$(SBIN)`/boinc_gahp.

5.5.18 condor_job_router Configuration File Entries

These macros affect the *condor_job_router* daemon.

JOB_ROUTER_ROUTE_NAMES
An ordered list of the names of enabled routes. In version 8.9.7 or later, routes whose names are listed here should each have a `JOB_ROUTER_ROUTE_<NAME>` configuration variable that specifies the route.

Routes will be matched to jobs in the order their names are declared in this list. Routes not declared in this list will be disabled.

If routes are specified in the deprecated *JOB_ROUTER_ENTRIES*, *JOB_ROUTER_ENTRIES_FILE* and *JOB_ROUTER_ENTRIES_CMD* configuration variables, then `JOB_ROUTER_ROUTE_NAMES` is optional. if it is empty, the order in which routes are considered will be the order in which their names hash.

JOB_ROUTER_ROUTE_<NAME>
Specification of a single route in transform syntax. `<NAME>` should be one of the route names specified in `JOB_ROUTER_ROUTE_NAMES`. The transform syntax is specified in the *ClassAd Transforms* section of this manual.

JOB_ROUTER_PRE_ROUTE_TRANSFORM_NAMES
An ordered list of the names of transforms that should be applied when a job is being routed before the route transform is applied. Each transform name listed here should have a corresponding `JOB_ROUTER_TRANSFORM_<NAME>` configuration variable.

JOB_ROUTER_POST_ROUTE_TRANSFORM_NAMES
An ordered list of the names of transforms that should be applied when a job is being routed after the route transform is applied. Each transform name listed here should have a corresponding JOB_ROUTER_TRANSFORM_<NAME> configuration variable.

JOB_ROUTER_TRANSFORM_<NAME>
Specification of a single pre-route or post-route transform. <NAME> should be one of the route names specified in JOB_ROUTER_PRE_ROUTE_TRANSFORM_NAMES or in JOB_ROUTER_POST_ROUTE_TRANSFORM_NAMES. The transform syntax is specified in the *ClassAd Transforms* section of this manual.

JOB_ROUTER_DEFAULTS

> **Warning:** This macro is deprecated and will be removed for V24 of HTCondor. The actual removal of this configuration macro will occur during the lifetime of the HTCondor V23 feature series.

Deprecated, use instead. Defined by a single ClassAd in New ClassAd syntax, used to provide default values for routes in the *condor_job_router* daemon's routing table that are specified by the also deprecated JOB_ROUTER_ENTRIES*. The enclosing square brackets are optional.

JOB_ROUTER_ENTRIES

> **Warning:** This macro is deprecated and will be removed for V24 of HTCondor. The actual removal of this configuration macro will occur during the lifetime of the HTCondor V23 feature series.

Deprecated, use instead. Specification of the job routing table. It is a list of ClassAds, in New ClassAd syntax, where each individual ClassAd is surrounded by square brackets, and the ClassAds are separated from each other by spaces. Each ClassAd describes one entry in the routing table, and each describes a site that jobs may be routed to.

A *condor_reconfig* command causes the *condor_job_router* daemon to rebuild the routing table. Routes are distinguished by a routing table entry's ClassAd attribute Name. Therefore, a Name change in an existing route has the potential to cause the inaccurate reporting of routes.

Instead of setting job routes using this configuration variable, they may be read from an external source using the JOB_ROUTER_ENTRIES_FILE or be dynamically generated by an external program via the JOB_ROUTER_ENTRIES_CMD configuration variable.

Routes specified by any of these 3 configuration variables are merged with the JOB_ROUTER_DEFAULTS before being used.

JOB_ROUTER_ENTRIES_FILE

> **Warning:** This macro is deprecated and will be removed for V24 of HTCondor. The actual removal of this configuration macro will occur during the lifetime of the HTCondor V23 feature series.

Deprecated, use instead. A path and file name of a file that contains the ClassAds, in New ClassAd syntax, describing the routing table. The specified file is periodically reread to check for new information. This occurs every $(JOB_ROUTER_ENTRIES_REFRESH) seconds.

JOB_ROUTER_ENTRIES_CMD

Warning: This macro is deprecated and will be removed for V24 of HTCondor. The actual removal of this configuration macro will occur during the lifetime of the HTCondor V23 feature series.

Deprecated, use instead. Specifies the command line of an external program to run. The output of the program defines or updates the routing table, and the output must be given in New ClassAd syntax. The specified command is periodically rerun to regenerate or update the routing table. This occurs every `$(JOB_ROUTER_ENTRIES_REFRESH)` seconds. Specify the full path and file name of the executable within this command line, as no assumptions may be made about the current working directory upon command invocation. To enter spaces in any command-line arguments or in the command name itself, surround the right hand side of this definition with double quotes, and use single quotes around individual arguments that contain spaces. This is the same as when dealing with spaces within job arguments in an HTCondor submit description file.

JOB_ROUTER_ENTRIES_REFRESH
The number of seconds between updates to the routing table described by `JOB_ROUTER_ENTRIES_FILE` or `JOB_ROUTER_ENTRIES_CMD`. The default value is 0, meaning no periodic updates occur. With the default value of 0, the routing table can be modified when a *condor_reconfig* command is invoked or when the *condor_job_router* daemon restarts.

JOB_ROUTER_LOCK
This specifies the name of a lock file that is used to ensure that multiple instances of condor_job_router never run with the same `JOB_ROUTER_NAME`. Multiple instances running with the same name could lead to mismanagement of routed jobs. The default value is $(LOCK)/$(JOB_ROUTER_NAME)Lock.

JOB_ROUTER_SOURCE_JOB_CONSTRAINT
Specifies a global `Requirements` expression that must be true for all newly routed jobs, in addition to any `Requirements` specified within a routing table entry. In addition to the configurable constraints, the *condor_job_router* also has some hard-coded constraints. It avoids recursively routing jobs by requiring that the job's attribute `RoutedBy` does not match `JOB_ROUTER_NAME` . When not running as root, it also avoids routing jobs belonging to other users.

JOB_ROUTER_MAX_JOBS
An integer value representing the maximum number of jobs that may be routed, summed over all routes. The default value is -1, which means an unlimited number of jobs may be routed.

JOB_ROUTER_DEFAULT_MAX_JOBS_PER_ROUTE
An integer value representing the maximum number of jobs that may be routed to a single route when the route does not specify a `MaxJobs` value. The default value is 100.

JOB_ROUTER_DEFAULT_MAX_IDLE_JOBS_PER_ROUTE
An integer value representing the maximum number of jobs in a single route that may be in the idle state. When the number of jobs routed to that site exceeds this number, no more jobs will be routed to it. A route may specify `MaxIdleJobs` to override this number. The default value is 50.

MAX_JOB_MIRROR_UPDATE_LAG
An integer value that administrators will rarely consider changing, representing the maximum number of seconds the *condor_job_router* daemon waits, before it decides that routed copies have gone awry, due to the failure of events to appear in the *condor_schedd* 's job queue log file. The default value is 600. As the *condor_job_router* daemon uses the *condor_schedd* 's job queue log file entries for synchronization of routed copies, when an expected log file event fails to appear after this wait period, the *condor_job_router* daemon acts presuming the expected event will never occur.

JOB_ROUTER_POLLING_PERIOD
An integer value representing the number of seconds between cycles in the *condor_job_router* daemon's task loop. The default is 10 seconds. A small value makes the *condor_job_router* daemon quick to see new candidate jobs for routing. A large value makes the *condor_job_router* daemon generate less overhead at the cost of being slower to see new candidates for routing. For very large job queues where a few minutes of routing latency is no problem, increasing this value to a few hundred seconds would be reasonable.

JOB_ROUTER_NAME
A unique identifier utilized to name multiple instances of the *condor_job_router* daemon on the same machine. Each instance must have a different name, or all but the first to start up will refuse to run. The default is `"jobrouter"`.

Changing this value when routed jobs already exist is not currently gracefully handled. However, it can be done if one also uses *condor_qedit* to change the value of `ManagedManager` and `RoutedBy` from the old name to the new name. The following commands may be helpful:

```
$ condor_qedit -constraint \
'RoutedToJobId =!= undefined && ManagedManager == "insert_old_name"' \
  ManagedManager '"insert_new_name"'
$ condor_qedit -constraint \
'RoutedBy == "insert_old_name"' RoutedBy '"insert_new_name"'
```

JOB_ROUTER_RELEASE_ON_HOLD
A boolean value that defaults to `True`. It controls how the *condor_job_router* handles the routed copy when it goes on hold. When `True`, the *condor_job_router* leaves the original job ClassAd in the same state as when claimed. When `False`, the *condor_job_router* does not attempt to reset the original job ClassAd to a pre-claimed state upon yielding control of the job.

JOB_ROUTER_SCHEDD1_SPOOL
The path to the spool directory for the *condor_schedd* serving as the source of jobs for routing. If not specified, this defaults to `$(SPOOL)`. If specified, this parameter must point to the spool directory of the *condor_schedd* identified by `JOB_ROUTER_SCHEDD1_NAME`.

JOB_ROUTER_SCHEDD2_SPOOL
The path to the spool directory for the *condor_schedd* to which the routed copy of the jobs are submitted. If not specified, this defaults to `$(SPOOL)`. If specified, this parameter must point to the spool directory of the *condor_schedd* identified by `JOB_ROUTER_SCHEDD2_NAME`. Note that when *condor_job_router* is running as root and is submitting routed jobs to a different *condor_schedd* than the source *condor_schedd*, it is required that *condor_job_router* have permission to impersonate the job owners of the routed jobs. It is therefore usually necessary to configure `QUEUE_SUPER_USER_MAY_IMPERSONATE` in the configuration of the target *condor_schedd*.

JOB_ROUTER_SCHEDD1_NAME
The advertised daemon name of the *condor_schedd* serving as the source of jobs for routing. If not specified, this defaults to the local *condor_schedd*. If specified, this parameter must name the same *condor_schedd* whose spool is configured in `JOB_ROUTER_SCHEDD1_SPOOL`. If the named *condor_schedd* is not advertised in the local pool, `JOB_ROUTER_SCHEDD1_POOL` will also need to be set.

JOB_ROUTER_SCHEDD2_NAME
The advertised daemon name of the *condor_schedd* to which the routed copy of the jobs are submitted. If not specified, this defaults to the local *condor_schedd*. If specified, this parameter must name the same *condor_schedd* whose spool is configured in `JOB_ROUTER_SCHEDD2_SPOOL`. If the named *condor_schedd* is not advertised in the local pool, `JOB_ROUTER_SCHEDD2_POOL` will also need to be set. Note that when *condor_job_router* is running as root and is submitting routed jobs to a different *condor_schedd* than the source *condor_schedd*, it is required that *condor_job_router* have permission to impersonate the job owners of the routed jobs. It is therefore usually necessary to configure `QUEUE_SUPER_USER_MAY_IMPERSONATE` in the configuration of the target *condor_schedd*.

JOB_ROUTER_SCHEDD1_POOL
The Condor pool (*condor_collector* address) of the *condor_schedd* serving as the source of jobs for routing. If not specified, defaults to the local pool.

JOB_ROUTER_SCHEDD2_POOL
The Condor pool (*condor_collector* address) of the *condor_schedd* to which the routed copy of the jobs are submitted. If not specified, defaults to the local pool.

JOB_ROUTER_ROUND_ROBIN_SELECTION
A boolean value that controls which route is chosen for a candidate job that matches multiple routes. When set to `False`, the default, the first matching route is always selected. When set to `True`, the Job Router attempts to distribute jobs across all matching routes, round robin style.

JOB_ROUTER_CREATE_IDTOKEN_NAMES
An list of the names of IDTOKENs that the JobRouter should create and refresh. IDTOKENS whose names are listed here should each have a `JOB_ROUTER_CREATE_IDTOKEN_<NAME>` configuration variable that specifies the the filename, ownership and properties of the IDTOKEN.

JOB_ROUTER_IDTOKEN_REFRESH
An integer value of secounds that controls the rate at which the JobRouter will refresh the IDTOKENS listed by the `JOB_ROUTER_CREATE_IDTOKEN_NAMES` configuration variable.

JOB_ROUTER_CREATE_IDTOKEN_<NAME>
Specification of a single IDTOKEN that will be created an refreshed by the JobRouter. `<NAME>` should be one of the IDTOKEN names specified in `JOB_ROUTER_CREATE_IDTOKEN_NAMES`. The filename, ownership and properties of the IDTOKEN are defined by the following attributes. Each attribute value must be a classad expression that evaluates to a string, except `lifetime` which must evaluate to an integer.

kid
The ID of the token signing key to use, equivalent to the `-key` argument of *condor_token_create* and the `kid` attribute of *condor_token_list*. Defaults to "POOL"

sub
The subject or user identity, equivalent to the `-identity` argument of *condor_token_create* and the `sub` attribute of *condor_token_list*. Defaults the token name.

scope
List of allowed authorizations, equivalent to the `-authz` argument of *condor_token_create* and the `scope` attribute of *condor_token_list*.

lifetime
Time in seconds that the IDTOKEN is valid after creation, equivalent to the `-lifetime` argument of *condor_token_create*. The `exp` attribute of *condor_token_list* is the creation time of the token plus this value.

file
The filename of the IDTOKEN file, equivalent to the `-token` argument of *condor_token_create*. Defaults to the token name.

dir
The directory that the IDTOKEN file will be created and refreshed into. Defaults to `$(SEC_TOKEN_DIRECTORY)`.

owner
If specified, the IDTOKEN file will be owned by this user. If not specified, the IDTOKEN file will be owned by the owner of *condor_job_router* process. This attribute is optional if the *condor_job_router* is running as an ordinary user but required if it is running as a Windows service or as the `root` or `condor` user. The owner specified here should be the same as the `Owner` attribute of the jobs that this IDTOKEN is intended to be sent to.

JOB_ROUTER_SEND_ROUTE_IDTOKENS
List of the names of the IDTOKENS to add to the input file transfer list of each routed job. This list should be one or more of the IDTOKEN names specified by the `JOB_ROUTER_CREATE_IDTOKEN_NAMES`. If the route has a `SendIDTokens` definition, this configuration variable is not used for that route.

5.5.19 condor_lease_manager Configuration File Entries

These macros affect the *condor_lease_manager*.

The *condor_lease_manager* expects to use the syntax

```
<subsystem name>.<parameter name>
```

in configuration. This allows multiple instances of the *condor_lease_manager* to be easily configured using the syntax

```
<subsystem name>.<local name>.<parameter name>
```

LeaseManager.GETADS_INTERVAL
An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* pulls relevant resource ClassAds from the *condor_collector*. The default value is 60 seconds, with a minimum value of 2 seconds.

LeaseManager.UPDATE_INTERVAL
An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* sends its ClassAds to the *condor_collector*. The default value is 60 seconds, with a minimum value of 5 seconds.

LeaseManager.PRUNE_INTERVAL
An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* prunes its leases. This involves checking all leases to see if they have expired. The default value is 60 seconds, with no minimum value.

LeaseManager.DEBUG_ADS
A boolean value that defaults to `False`. When `True`, it enables extra debugging information about the resource ClassAds that it retrieves from the *condor_collector* and about the search ClassAds that it sends to the *condor_collector*.

LeaseManager.MAX_LEASE_DURATION
An integer value representing seconds which determines the maximum duration of a lease. This can be used to provide a hard limit on lease durations. Normally, the *condor_lease_manager* honors the `MaxLeaseDuration` attribute from the resource ClassAd. If this configuration variable is defined, it limits the effective maximum duration for all resources to this value. The default value is 1800 seconds.

Note that leases can be renewed, and thus can be extended beyond this limit. To provide a limit on the total duration of a lease, use `LeaseManager.MAX_TOTAL_LEASE_DURATION`.

LeaseManager.MAX_TOTAL_LEASE_DURATION
An integer value representing seconds used to limit the total duration of leases, over all its renewals. The default value is 3600 seconds.

LeaseManager.DEFAULT_MAX_LEASE_DURATION
The *condor_lease_manager* uses the `MaxLeaseDuration` attribute from the resource ClassAd to limit the lease duration. If this attribute is not present in a resource ClassAd, then this configuration variable is used instead. This integer value is given in units of seconds, with a default value of 60 seconds.

LeaseManager.CLASSAD_LOG
This variable defines a full path and file name to the location where the *condor_lease_manager* keeps persistent state information. This variable has no default value.

LeaseManager.QUERY_ADTYPE
This parameter controls the type of the query in the ClassAd sent to the *condor_collector*, which will control the types of ClassAds returned by the *condor_collector*. This parameter must be a valid ClassAd type name, with a default value of `"Any"`.

LeaseManager.QUERY_CONSTRAINTS
A ClassAd expression that controls the constraint in the query sent to the *condor_collector*. It is used to further constrain the types of ClassAds from the *condor_collector*. There is no default value, resulting in no constraints being placed on query.

5.5.20 Configuration File Entries for DAGMan

These macros affect the operation of DAGMan and DAGMan jobs within HTCondor.

Note: Many, if not all, of these configuration variables will be most appropriately set on a per DAG basis, rather than in the global HTCondor configuration files. Per DAG configuration is explained in *Configuration Specific to a DAG*. Also note that configuration settings of a running *condor_dagman* job are not changed by doing a *condor_reconfig*.

General

DAGMAN_CONFIG_FILE
The path and name of the configuration file to be used by *condor_dagman*. This configuration variable is set automatically by *condor_submit_dag*, and it should not be explicitly set by the user. Defaults to the empty string.

DAGMAN_USE_STRICT
An integer defining the level of strictness *condor_dagman* will apply when turning warnings into fatal errors, as follows:

- 0: no warnings become errors
- 1: severe warnings become errors
- 2: medium-severity warnings become errors
- 3: almost all warnings become errors

Using a strictness value greater than 0 may help find problems with a DAG that may otherwise escape notice. The default value if not defined is 1.

DAGMAN_STARTUP_CYCLE_DETECT
A boolean value that defaults to `False`. When `True`, causes *condor_dagman* to check for cycles in the DAG before submitting DAG node jobs, in addition to its run time cycle detection. Note that setting this value to `True` will impose significant startup delays for large DAGs.

DAGMAN_ABORT_DUPLICATES
A boolean value that controls whether to attempt to abort duplicate instances of *condor_dagman* running the same DAG on the same machine. When *condor_dagman* starts up, if no DAG lock file exists, *condor_dagman* creates the lock file and writes its PID into it. If the lock file does exist, and `DAGMAN_ABORT_DUPLICATES` is set to `True`, *condor_dagman* checks whether a process with the given PID exists, and if so, it assumes that there is already another instance of *condor_dagman* running the same DAG. Note that this test is not foolproof: it is possible that, if *condor_dagman* crashes, the same PID gets reused by another process before *condor_dagman* gets rerun on that DAG. This should be quite rare, however. If not defined, `DAGMAN_ABORT_DUPLICATES` defaults to `True`. **Note: users should rarely change this setting.**

DAGMAN_USE_SHARED_PORT
A boolean value that controls whether *condor_dagman* will attempt to connect to the shared port daemon. If not defined, `DAGMAN_USE_SHARED_PORT` defaults to `False`. There is no reason to ever change this value; it was introduced to prevent spurious shared port-related error messages from appearing in `dagman.out` files.

DAGMAN_USE_DIRECT_SUBMIT
A boolean value that controls whether *condor_dagman* submits jobs using *condor_submit* or by opening a direct connection to the *condor_schedd*. `DAGMAN_USE_DIRECT_SUBMIT` defaults to `True`. When set to `True` *condor_dagman* will submit jobs to the local Schedd by connecting to it directly. This is faster than using *condor_submit*, especially for very large DAGs; But this method will ignore some submit file features such as `max_materialize` and more than one `QUEUE` statement.

DAGMAN_USE_JOIN_NODES
A boolean value that defaults to `True`. When `True`, causes *condor_dagman* to break up many-PARENT-many-CHILD relationships with an intermediate *join node*. When these sets are large, this significantly optimizes the graph structure by reducing the number of dependencies, resulting in a significant improvement to the *condor_dagman* memory footprint, parse time, and submit speed.

DAGMAN_PUT_FAILED_JOBS_ON_HOLD
A boolean value that when set to `True` causes DAGMan to automatically retry a node with its job submitted on hold, if any of the nodes job procs fail. This only applies for job failures and not `PRE`, `POST`, or `HOLD` script failures within a DAG node. The job is only put on hold if the node has no more declared `RETRY` attempts. The default value is `False`.

DAGMAN_DEFAULT_APPEND_VARS
A boolean value that defaults to `False`. When `True`, variables parsed in the DAG file *VARS* line will be appended to the given Job submit description file unless *VARS* specifies *PREPEND* or *APPEND*. When `False`, the parsed variables will be prepended unless specified.

DAGMAN_MANAGER_JOB_APPEND_GETENV
A comma separated list of variable names to add to the DAGMan `.condor.sub` file's `getenv` option. This will in turn add any found matching environment variables to the DAGMan proper jobs **environment**. Setting this value to `True` will result in `getenv = true`. The Base `.condor.sub` values for `getenv` are the following.

General Shell	PATH	HOME	USER
	TZ	LANG	LC_ALL
	PYTHONPATH	PERL*	
HTCondor	CONDOR_CONFIG	CONDOR_*	
Scitoken	BEARER_TOKEN	BEAERER_TOKEN_FILE	XDG_RUNTIME_DIR
Misc.	PEGASUS_*		

DAGMAN_NODE_RECORD_INFO
A string that when set to `RETRY` will cause DAGMan to insert a nodes current retry attempt number into the nodes job ad as the attribute `DAGManNodeRetry` at submission time. This knob is not set by default.

DAGMAN_RECORD_MACHINE_ATTRS
A comma separated list of machine attributes that DAGMan will insert into a node jobs submit description for `job_ad_information_attrs` and `job_machine_attrs`. This will result in the listed machine attributes to be injected into the nodes produced job ads and userlog. This knob is not set by default.

Throttling

DAGMAN_MAX_JOBS_IDLE
An integer value that controls the maximum number of idle procs allowed within the DAG before *condor_dagman* temporarily stops submitting jobs. *condor_dagman* will resume submitting jobs once the number of idle procs falls below the specified limit. `DAGMAN_MAX_JOBS_IDLE` currently counts each individual proc within a cluster as a job, which is inconsistent with `DAGMAN_MAX_JOBS_SUBMITTED`. Note that submit description files that queue multiple procs can cause the `DAGMAN_MAX_JOBS_IDLE` limit to be exceeded. If a submit description file contains `queue 5000` and `DAGMAN_MAX_JOBS_IDLE` is set to 250, this will result in 5000 procs being submitted

to the *condor_schedd*, not 250; in this case, no further jobs will then be submitted by *condor_dagman* until the number of idle procs falls below 250. The default value is 1000. To disable this limit, set the value to 0. This configuration option can be overridden by the *condor_submit_dag* **-maxidle** command-line argument (see *condor_submit_dag*).

DAGMAN_MAX_JOBS_SUBMITTED
An integer value that controls the maximum number of node jobs (clusters) within the DAG that will be submitted to HTCondor at one time. A single invocation of *condor_submit* by *condor_dagman* counts as one job, even if the submit file produces a multi-proc cluster. The default value is 0 (unlimited). This configuration option can be overridden by the *condor_submit_dag* **-maxjobs** command-line argument (see *condor_submit_dag*).

DAGMAN_MAX_PRE_SCRIPTS
An integer defining the maximum number of PRE scripts that any given *condor_dagman* will run at the same time. The value 0 allows any number of PRE scripts to run. The default value if not defined is 20. Note that the `DAGMAN_MAX_PRE_SCRIPTS` value can be overridden by the *condor_submit_dag* **-maxpre** command line option.

DAGMAN_MAX_POST_SCRIPTS
An integer defining the maximum number of POST scripts that any given *condor_dagman* will run at the same time. The value 0 allows any number of POST scripts to run. The default value if not defined is 20. Note that the `DAGMAN_MAX_POST_SCRIPTS` value can be overridden by the *condor_submit_dag* **-maxpost** command line option.

DAGMAN_MAX_HOLD_SCRIPTS
An integer defining the maximum number of HOLD scripts that any given *condor_dagman* will run at the same time. The default value 0 allows any number of HOLD scripts to run.

DAGMAN_REMOVE_JOBS_AFTER_LIMIT_CHANGE
A boolean that determines if after changing some of these throttle limits, *condor_dagman* should forceably remove jobs to meet the new limit. Defaults to `False`.

Priority, node semantics

DAGMAN_DEFAULT_PRIORITY
An integer value defining the minimum priority of node jobs running under this *condor_dagman* job. Defaults to 0.

DAGMAN_SUBMIT_DEPTH_FIRST
A boolean value that controls whether to submit ready DAG node jobs in (more-or-less) depth first order, as opposed to breadth-first order. Setting `DAGMAN_SUBMIT_DEPTH_FIRST` to `True` does not override dependencies defined in the DAG. Rather, it causes newly ready nodes to be added to the head, rather than the tail, of the ready node list. If there are no PRE scripts in the DAG, this will cause the ready nodes to be submitted depth-first. If there are PRE scripts, the order will not be strictly depth-first, but it will tend to favor depth rather than breadth in executing the DAG. If `DAGMAN_SUBMIT_DEPTH_FIRST` is set to `True`, consider also setting and to `True`. If not defined, `DAGMAN_SUBMIT_DEPTH_FIRST` defaults to `False`.

DAGMAN_ALWAYS_RUN_POST
A boolean value defining whether *condor_dagman* will ignore the return value of a PRE script when deciding whether to run a POST script. The default is `False`, which means that the failure of a PRE script causes the POST script to not be executed. Changing this to `True` will restore the previous behavior of *condor_dagman*, which is that a POST script is always executed, even if the PRE script fails.

Node job submission/removal

DAGMAN_USER_LOG_SCAN_INTERVAL
An integer value representing the number of seconds that *condor_dagman* waits between checking the workflow log file for status updates. Setting this value lower than the default increases the CPU time *condor_dagman* spends checking files, perhaps fruitlessly, but increases responsiveness to nodes completing or failing. The legal range of values is 1 to INT_MAX. If not defined, it defaults to 5 seconds. This default may be automatically decreased if is set to a small value. If so, this will be noted in the `dagman.out` file.

DAGMAN_MAX_SUBMITS_PER_INTERVAL
An integer that controls how many individual jobs *condor_dagman* will submit in a row before servicing other requests (such as a *condor_rm*). The legal range of values is 1 to 1000. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 1000, the value 1000 will be used. If not defined, it defaults to 100. This default may be automatically decreased if is set to a small value. If so, this will be noted in the `dagman.out` file.

Note: The maximum rate at which DAGMan can submit jobs is DAGMAN_MAX_SUBMITS_PER_INTERVAL / DAGMAN_USER_LOG_SCAN_INTERVAL.

DAGMAN_MAX_SUBMIT_ATTEMPTS
An integer that controls how many times in a row *condor_dagman* will attempt to execute *condor_submit* for a given job before giving up. Note that consecutive attempts use an exponential backoff, starting with 1 second. The legal range of values is 1 to 16. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 16, the value 16 will be used. Note that a value of 16 would result in *condor_dagman* trying for approximately 36 hours before giving up. If not defined, it defaults to 6 (approximately two minutes before giving up).

DAGMAN_MAX_JOB_HOLDS
An integer value defining the maximum number of times a node job is allowed to go on hold. As a job goes on hold this number of times, it is removed from the queue. For example, if the value is 2, as the job goes on hold for the second time, it will be removed. At this time, this feature is not fully compatible with node jobs that have more than one `ProcID`. The number of holds of each process in the cluster count towards the total, rather than counting individually. So, this setting should take that possibility into account, possibly using a larger value. A value of 0 allows a job to go on hold any number of times. The default value if not defined is 100.

DAGMAN_HOLD_CLAIM_TIME
An integer defining the number of seconds that *condor_dagman* will cause a hold on a claim after a job is finished, using the job ClassAd attribute `KeepClaimIdle`. The default value is 20. A value of 0 causes *condor_dagman* not to set the job ClassAd attribute.

DAGMAN_SUBMIT_DELAY
An integer that controls the number of seconds that *condor_dagman* will sleep before submitting consecutive jobs. It can be increased to help reduce the load on the *condor_schedd* daemon. The legal range of values is any non negative integer. If defined with a value less than 0, the value 0 will be used.

DAGMAN_PROHIBIT_MULTI_JOBS
A boolean value that controls whether *condor_dagman* prohibits node job submit description files that queue multiple job procs other than parallel universe. If a DAG references such a submit file, the DAG will abort during the initialization process. If not defined, `DAGMAN_PROHIBIT_MULTI_JOBS` defaults to `False`.

DAGMAN_GENERATE_SUBDAG_SUBMITS
A boolean value specifying whether *condor_dagman* itself should create the `.condor.sub` files for nested DAGs. If set to `False`, nested DAGs will fail unless the `.condor.sub` files are generated manually by running *condor_submit_dag -no_submit* on each nested DAG, or the *-do_recurse* flag is passed to *condor_submit_dag* for the top-level DAG. DAG nodes specified with the `SUBDAG EXTERNAL` keyword or with submit description file names ending in `.condor.sub` are considered nested DAGs. The default value if not defined is `True`.

DAGMAN_REMOVE_NODE_JOBS
A boolean value that controls whether *condor_dagman* removes its node jobs itself when it is removed (in addition to the *condor_schedd* removing them). Note that setting DAGMAN_REMOVE_NODE_JOBS to True is the safer option (setting it to False means that there is some chance of ending up with "orphan" node jobs). Setting DAGMAN_REMOVE_NODE_JOBS to False is a performance optimization (decreasing the load on the *condor_schedd* when a *condor_dagman* job is removed). Note that even if DAGMAN_REMOVE_NODE_JOBS is set to False, *condor_dagman* will remove its node jobs in some cases, such as a DAG abort triggered by an *ABORT-DAG-ON* command. Defaults to True.

DAGMAN_MUNGE_NODE_NAMES
A boolean value that controls whether *condor_dagman* automatically renames nodes when running multiple DAGs. The renaming is done to avoid possible name conflicts. If this value is set to True, all node names have the DAG number followed by the period character (.) prepended to them. For example, the first DAG specified on the *condor_submit_dag* command line is considered DAG number 0, the second is DAG number 1, etc. So if DAG number 2 has a node named B, that node will internally be renamed to 2.B. If not defined, DAGMAN_MUNGE_NODE_NAMES defaults to True. **Note: users should rarely change this setting.**

DAGMAN_SUPPRESS_JOB_LOGS
A boolean value specifying whether events should be written to a log file specified in a node job's submit description file. The default value is False, such that events are written to a log file specified by a node job.

DAGMAN_SUPPRESS_NOTIFICATION
A boolean value defining whether jobs submitted by *condor_dagman* will use email notification when certain events occur. If True, all jobs submitted by *condor_dagman* will have the equivalent of the submit command notification = never set. This does not affect the notification for events relating to the *condor_dagman* job itself. Defaults to True.

DAGMAN_CONDOR_SUBMIT_EXE
The executable that *condor_dagman* will use to submit HTCondor jobs. If not defined, *condor_dagman* looks for *condor_submit* in the path. **Note: users should rarely change this setting.**

DAGMAN_CONDOR_RM_EXE
The executable that *condor_dagman* will use to remove HTCondor jobs. If not defined, *condor_dagman* looks for *condor_rm* in the path. **Note: users should rarely change this setting.**

DAGMAN_ABORT_ON_SCARY_SUBMIT
A boolean value that controls whether to abort a DAG upon detection of a scary submit event. An example of a scary submit event is one in which the HTCondor ID does not match the expected value. Note that in all HTCondor versions prior to 6.9.3, *condor_dagman* did not abort a DAG upon detection of a scary submit event. This behavior is what now happens if DAGMAN_ABORT_ON_SCARY_SUBMIT is set to False. If not defined, DAGMAN_ABORT_ON_SCARY_SUBMIT defaults to True. **Note: users should rarely change this setting.**

Rescue/retry

DAGMAN_AUTO_RESCUE
A boolean value that controls whether *condor_dagman* automatically runs Rescue DAGs. If DAGMAN_AUTO_RESCUE is True and the DAG input file my.dag is submitted, and if a Rescue DAG such as the examples my.dag.rescue001 or my.dag.rescue002 exists, then the largest magnitude Rescue DAG will be run. If not defined, DAGMAN_AUTO_RESCUE defaults to True.

DAGMAN_MAX_RESCUE_NUM
An integer value that controls the maximum Rescue DAG number that will be written, in the case that DAGMAN_OLD_RESCUE is False, or run if DAGMAN_AUTO_RESCUE is True. The maximum legal value is 999; the minimum value is 0, which prevents a Rescue DAG from being written at all, or automatically run. If not defined, DAGMAN_MAX_RESCUE_NUM defaults to 100.

DAGMAN_RESET_RETRIES_UPON_RESCUE
A boolean value that controls whether node retries are reset in a Rescue DAG. If this value is `False`, the number of node retries written in a Rescue DAG is decreased, if any retries were used in the original run of the DAG; otherwise, the original number of retries is allowed when running the Rescue DAG. If not defined, `DAGMAN_RESET_RETRIES_UPON_RESCUE` defaults to `True`.

DAGMAN_WRITE_PARTIAL_RESCUE
A boolean value that controls whether *condor_dagman* writes a partial or a full DAG file as a Rescue DAG. If not defined, `DAGMAN_WRITE_PARTIAL_RESCUE` defaults to `True`. **Note: users should rarely change this setting.**

DAGMAN_RETRY_SUBMIT_FIRST
A boolean value that controls whether a failed submit is retried first (before any other submits) or last (after all other ready jobs are submitted). If this value is set to `True`, when a job submit fails, the job is placed at the head of the queue of ready jobs, so that it will be submitted again before any other jobs are submitted. This had been the behavior of *condor_dagman*. If this value is set to `False`, when a job submit fails, the job is placed at the tail of the queue of ready jobs. If not defined, it defaults to `True`.

DAGMAN_RETRY_NODE_FIRST
A boolean value that controls whether a failed node with retries is retried first (before any other ready nodes) or last (after all other ready nodes). If this value is set to `True`, when a node with retries fails after the submit succeeded, the node is placed at the head of the queue of ready nodes, so that it will be tried again before any other jobs are submitted. If this value is set to `False`, when a node with retries fails, the node is placed at the tail of the queue of ready nodes. This had been the behavior of *condor_dagman*. If not defined, it defaults to `False`.

Log files

DAGMAN_DEFAULT_NODE_LOG
The default name of a file to be used as a job event log by all node jobs of a DAG.

This configuration variable uses a special syntax in which **@** instead of **$** indicates an evaluation of special variables. Normal HTCondor configuration macros may be used with the normal **$** syntax.

Special variables to be used only in defining this configuration variable:

- `@(DAG_DIR)`: The directory in which the primary DAG input file resides. If more than one DAG input file is specified to *condor_submit_dag*, the primary DAG input file is the leftmost one on the command line.
- `@(DAG_FILE)`: The name of the primary DAG input file. It does not include the path.
- `@(CLUSTER)`: The `ClusterId` attribute of the *condor_dagman* job.
- `@(OWNER)`: The user name of the user who submitted the DAG.
- `@(NODE_NAME)`: For SUBDAGs, this is the node name of the SUBDAG in the upper level DAG; for a top-level DAG, it is the string `"undef"`.

If not defined, `@(DAG_DIR)/@(DAG_FILE).nodes.log` is the default value.

Notes:

- Using `$(LOG)` in defining a value for `DAGMAN_DEFAULT_NODE_LOG` will not have the expected effect, because `$(LOG)` is defined as `"."` for *condor_dagman*. To place the default log file into the log directory, write the expression relative to a known directory, such as `$(LOCAL_DIR)/log` (see examples below).
- A default log file placed in the spool directory will need extra configuration to prevent *condor_preen* from removing it; modify `VALID_SPOOL_FILES`. Removal of the default log file during a run will cause severe problems.

- **The value defined for DAGMAN_DEFAULT_NODE_LOG must ensure that the file is unique for each DAG.** Therefore, the value should always include `@(DAG_FILE)`. For example,

```
DAGMAN_DEFAULT_NODE_LOG = $(LOCAL_DIR)/log/@(DAG_FILE).nodes.log
```

  is okay, but

```
DAGMAN_DEFAULT_NODE_LOG = $(LOCAL_DIR)/log/dag.nodes.log
```

  will cause failure when more than one DAG is run at the same time on a given access point.

DAGMAN_LOG_ON_NFS_IS_ERROR
A boolean value that controls whether *condor_dagman* prohibits a DAG workflow log from being on an NFS file system. This value is ignored if `CREATE_LOCKS_ON_LOCAL_DISK` and `ENABLE_USERLOG_LOCKING` are both `True`. If a DAG uses such a workflow log file file and `DAGMAN_LOG_ON_NFS_IS_ERROR` is `True` (and not ignored), the DAG will abort during the initialization process. If not defined, `DAGMAN_LOG_ON_NFS_IS_ERROR` defaults to `False`.

DAGMAN_ALLOW_ANY_NODE_NAME_CHARACTERS
Allows any characters to be used in DAGMan node names, even characters that are considered illegal because they are used internally as separators. Turning this feature on could lead to instability when using splices or munged node names. The default value is `False`.

DAGMAN_ALLOW_EVENTS
An integer that controls which bad events are considered fatal errors by *condor_dagman*. This macro replaces and expands upon the functionality of the `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION` macro. If `DAGMAN_ALLOW_EVENTS` is set, it overrides the setting of `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION`. **Note: users should rarely change this setting.**

The `DAGMAN_ALLOW_EVENTS` value is a logical bitwise OR of the following values:

- 0 = allow no bad events
- 1 = allow all bad events, except the event `"job re-run after terminated event"`
- 2 = allow terminated/aborted event combination
- 4 = allow a `"job re-run after terminated event"` bug
- 8 = allow garbage or orphan events
- 16 = allow an execute or terminate event before job's submit event
- 32 = allow two terminated events per job, as sometimes seen with grid jobs
- 64 = allow duplicated events in general

The default value is 114, which allows terminated/aborted event combination, allows an execute and/or terminated event before job's submit event, allows double terminated events, and allows general duplicate events.

As examples, a value of 6 instructs *condor_dagman* to allow both the terminated/aborted event combination and the `"job re-run after terminated event"` bug. A value of 0 means that any bad event will be considered a fatal error.

A value of 5 will never abort the DAG because of a bad event. But this value should almost never be used, because the `"job re-run after terminated event"` bug breaks the semantics of the DAG.

Debug output

DAGMAN_DEBUG
This variable is described in .

DAGMAN_VERBOSITY
An integer value defining the verbosity of output to the `dagman.out` file, as follows (each level includes all output from lower debug levels):

- level = 0; never produce output, except for usage info
- level = 1; very quiet, output severe errors
- level = 2; output errors and warnings
- level = 3; normal output
- level = 4; internal debugging output
- level = 5; internal debugging output; outer loop debugging
- level = 6; internal debugging output; inner loop debugging
- level = 7; internal debugging output; rarely used

The default value if not defined is 3.

DAGMAN_DEBUG_CACHE_ENABLE
A boolean value that determines if log line caching for the `dagman.out` file should be enabled in the *condor_dagman* process to increase performance (potentially by orders of magnitude) when writing the `dagman.out` file to an NFS server. Currently, this cache is only utilized in Recovery Mode. If not defined, it defaults to `False`.

DAGMAN_DEBUG_CACHE_SIZE
An integer value representing the number of bytes of log lines to be stored in the log line cache. When the cache surpasses this number, the entries are written out in one call to the logging subsystem. A value of zero is not recommended since each log line would surpass the cache size and be emitted in addition to bracketing log lines explaining that the flushing was happening. The legal range of values is 0 to INT_MAX. If defined with a value less than 0, the value 0 will be used. If not defined, it defaults to 5 Megabytes.

DAGMAN_PENDING_REPORT_INTERVAL
An integer value representing the number of seconds that controls how often *condor_dagman* will print a report of pending nodes to the `dagman.out` file. The report will only be printed if *condor_dagman* has been waiting at least `DAGMAN_PENDING_REPORT_INTERVAL` seconds without seeing any node job events, in order to avoid cluttering the `dagman.out` file. This feature is mainly intended to help diagnose *condor_dagman* processes that are stuck waiting indefinitely for a job to finish. If not defined, `DAGMAN_PENDING_REPORT_INTERVAL` defaults to 600 seconds (10 minutes).

MAX_DAGMAN_LOG
This variable is described in . If not defined, `MAX_DAGMAN_LOG` defaults to 0 (unlimited size).

HTCondor attributes

DAGMAN_COPY_TO_SPOOL
A boolean value that when `True` copies the *condor_dagman* binary to the spool directory when a DAG is submitted. Setting this variable to `True` allows long-running DAGs to survive a DAGMan version upgrade. For running large numbers of small DAGs, leave this variable unset or set it to `False`. The default value if not defined is `False`. **Note: users should rarely change this setting.**

DAGMAN_INSERT_SUB_FILE
A file name of a file containing submit description file commands to be inserted into the `.condor.sub` file created by *condor_submit_dag*. The specified file is inserted into the `.condor.sub` file before the **queue** command and before any commands specified with the **-append** *condor_submit_dag* command line option. Note that the `DAGMAN_INSERT_SUB_FILE` value can be overridden by the *condor_submit_dag* **-insert_sub_file** command line option.

DAGMAN_ON_EXIT_REMOVE
Defines the `OnExitRemove` ClassAd expression placed into the *condor_dagman* submit description file by *condor_submit_dag*. The default expression is designed to ensure that *condor_dagman* is automatically re-queued by the *condor_schedd* daemon if it exits abnormally or is killed (for example, during a reboot). If this results in *condor_dagman* staying in the queue when it should exit, consider changing to a less restrictive expression, as in the example

```
(ExitBySignal == false || ExitSignal =!= 9)
```

If not defined, `DAGMAN_ON_EXIT_REMOVE` defaults to the expression

```
( ExitSignal =?= 11 || (ExitCode =!= UNDEFINED && ExitCode >=0 && ExitCode <= 2))
```

5.5.21 Configuration File Entries Relating to Security

These macros affect the secure operation of HTCondor. Many of these macros are described in the *Security* section.

SEC_*_AUTHENTICATION
Whether authentication is required for a specified permission level. Acceptable values are `REQUIRED`, `PREFERRED`, `OPTIONAL`, and `NEVER`. For example, setting `SEC_READ_AUTHENTICATION = REQUIRED` indicates that any command requiring `READ` authorization will fail unless authentication is performed. The special value, `SEC_DEFAULT_AUTHENTICATION`, controls the default setting if no others are specified.

SEC_*_ENCRYPTION
Whether encryption is required for a specified permission level. Encryption prevents another entity on the same network from understanding the contents of the transfer between client and server. Acceptable values are `REQUIRED`, `PREFERRED`, `OPTIONAL`, and `NEVER`. For example, setting `SEC_WRITE_ENCRYPTION = REQUIRED` indicates that any command requiring `WRITE` authorization will fail unless the channel is encrypted. The special value, `SEC_DEFAULT_ENCRYPTION`, controls the default setting if no others are specified.

SEC_*_INTEGRITY
Whether integrity-checking is required for a specified permission level. Integrity checking allows the client and server to detect changes (malicious or otherwise) to the contents of the transfer. Acceptable values are `REQUIRED`, `PREFERRED`, `OPTIONAL`, and `NEVER`. For example, setting `SEC_WRITE_INTEGRITY = REQUIRED` indicates that any command requiring `WRITE` authorization will fail unless the channel is integrity-checked. The special value, `SEC_DEFAULT_INTEGRITY`, controls the default setting if no others are specified.

As a special exception, file transfers are not integrity checked unless they are also encrypted.

SEC_*_NEGOTIATION
Whether the client and server should negotiate security parameters (such as encryption, integrity, and authentication) for a given authorization level. For example, setting `SEC_DEFAULT_NEGOTIATION = REQUIRED` will require a security negotiation for all permission levels by default. There is very little penalty for security negotiation and it is strongly suggested to leave this as the default (`REQUIRED`) at all times.

SEC_*_AUTHENTICATION_METHODS
An ordered list of allowed authentication methods for a given authorization level. This set of configuration variables controls both the ordering and the allowed methods. Currently allowed values are `SSL`, `KERBEROS`, `PASSWORD`, `FS` (non-Windows), `FS_REMOTE` (non-Windows), `NTSSPI`, `MUNGE`, `CLAIMTOBE`, `IDTOKENS`, `SCITOKENS`, and `ANONYMOUS`. See the *Security* section for a discussion of the relative merits of each method; some, such as `CLAIMTOBE` provide effectively no security at all. The default authentication methods are `NTSSPI,FS,IDTOKENS,KERBEROS,SSL`.

These methods are tried in order until one succeeds or they all fail; for this reason, we do not recommend changing the default method list.

The special value, `SEC_DEFAULT_AUTHENTICATION_METHODS`, controls the default setting if no others are specified.

SEC_*_CRYPTO_METHODS
An ordered list of allowed cryptographic algorithms to use for encrypting a network session at a specified authorization level. The server will select the first entry in its list that both server and client allow. Possible values are `AES`, `3DES`, and `BLOWFISH`. The special parameter name `SEC_DEFAULT_CRYPTO_METHODS` controls the default setting if no others are specified. There is little benefit in varying the setting per authorization level; it is recommended to leave these settings untouched.

HOST_ALIAS
Specifies the fully qualified host name that clients authenticating this daemon with SSL should expect the daemon's certificate to match. The alias is advertised to the *condor_collector* as part of the address of the daemon. When this is not set, clients validate the daemon's certificate host name by matching it against DNS A records for the host they are connected to. See `SSL_SKIP_HOST_CHECK` for ways to disable this validation step.

USE_COLLECTOR_HOST_CNAME
A boolean value that determines what hostname a client should expect when validating the collector's certificate during SSL authentication. When set to `True`, the hostname given to the client is used. When set to `False`, if the given hostname is a DNS CNAME, the client resolves it to a DNS A record and uses that hostname. The default value is `True`.

DELEGATE_JOB_GSI_CREDENTIALS
A boolean value that defaults to `True` for HTCondor version 6.7.19 and more recent versions. When `True`, a job's X.509 credentials are delegated, instead of being copied. This results in a more secure communication when not encrypted.

DELEGATE_FULL_JOB_GSI_CREDENTIALS
A boolean value that controls whether HTCondor will delegate a full or limited X.509 proxy. The default value of `False` indicates the limited X.509 proxy.

DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME
An integer value that specifies the maximum number of seconds for which delegated proxies should be valid. The default value is one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. The job may override this configuration setting by using the **delegate_job_GSI_credentials_lifetime** submit file command. This configuration variable currently only applies to proxies delegated for non-grid jobs and HTCondor-C jobs. This variable has no effect if `DELEGATE_JOB_GSI_CREDENTIALS` is `False`.

DELEGATE_JOB_GSI_CREDENTIALS_REFRESH
A floating point number between 0 and 1 that indicates the fraction of a proxy's lifetime at which point delegated credentials with a limited lifetime should be renewed. The renewal is attempted periodically at or near the

specified fraction of the lifetime of the delegated credential. The default value is 0.25. This setting has no effect if DELEGATE_JOB_GSI_CREDENTIALS is False or if DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME is 0. For non-grid jobs, the precise timing of the proxy refresh depends on SHADOW_CHECKPROXY_INTERVAL . To ensure that the delegated proxy remains valid, the interval for checking the proxy should be, at most, half of the interval for refreshing it.

USE_VOMS_ATTRIBUTES
A boolean value that controls whether HTCondor will attempt to extract and verify VOMS attributes from X.509 credentials. The default is False.

SEC_<access-level>_SESSION_DURATION
The amount of time in seconds before a communication session expires. A session is a record of necessary information to do communication between a client and daemon, and is protected by a shared secret key. The session expires to reduce the window of opportunity where the key may be compromised by attack. A short session duration increases the frequency with which daemons have to re-authenticate with each other, which may impact performance.

If the client and server are configured with different durations, the shorter of the two will be used. The default for daemons is 86400 seconds (1 day) and the default for command-line tools is 60 seconds. The shorter default for command-line tools is intended to prevent daemons from accumulating a large number of communication sessions from the short-lived tools that contact them over time. A large number of security sessions consumes a large amount of memory. It is therefore important when changing this configuration setting to preserve the small session duration for command-line tools.

One example of how to safely change the session duration is to explicitly set a short duration for tools and *condor_submit* and a longer duration for everything else:

```
SEC_DEFAULT_SESSION_DURATION = 50000
TOOL.SEC_DEFAULT_SESSION_DURATION = 60
SUBMIT.SEC_DEFAULT_SESSION_DURATION = 60
```

Another example of how to safely change the session duration is to explicitly set the session duration for a specific daemon:

```
COLLECTOR.SEC_DEFAULT_SESSION_DURATION = 50000
```

SEC_<access-level>_SESSION_LEASE
The maximum number of seconds an unused security session will be kept in a daemon's session cache before being removed to save memory. The default is 3600. If the server and client have different configurations, the smaller one will be used.

SEC_INVALIDATE_SESSIONS_VIA_TCP
Use TCP (if True) or UDP (if False) for responding to attempts to use an invalid security session. This happens, for example, if a daemon restarts and receives incoming commands from other daemons that are still using a previously established security session. The default is True.

FS_REMOTE_DIR
The location of a file visible to both server and client in Remote File System authentication. The default when not defined is the directory /shared/scratch/tmp.

ENCRYPT_EXECUTE_DIRECTORY
A boolean value that, when True, causes the execute directory for jobs on Linux or Windows platforms to be encrypted. Defaults to False. Note that even if False, the user can require encryption of the execute directory on a per-job basis by setting **encrypt_execute_directory** to True in the job submit description file. Enabling this functionality requires that the HTCondor service is run as user root on Linux platforms, or as a system service on Windows platforms. On Linux platforms, the encryption method is *ecryptfs*, and therefore requires an installation

of the `ecryptfs-utils` package. On Windows platforms, the encryption method is the EFS (Encrypted File System) feature of NTFS.

ENCRYPT_EXECUTE_DIRECTORY_FILENAMES
A boolean value relevant on Linux platforms only. Defaults to `False`. On Windows platforms, file names are not encrypted, so this variable has no effect. When using an encrypted execute directory, the contents of the files will always be encrypted. On Linux platforms, file names may or may not be encrypted. There is some overhead and there are restrictions on encrypting file names (see the *ecryptfs* documentation). As a result, the default does not encrypt file names on Linux platforms, and the administrator may choose to enable encryption behavior by setting this configuration variable to `True`.

ECRYPTFS_ADD_PASSPHRASE
The path to the *ecryptfs-add-passphrase* command-line utility. If the path is not fully-qualified, then safe system path subdirectories such as `/bin` and `/usr/bin` will be searched. The default value is `ecryptfs-add-passphrase`, causing the search to be within the safe system path subdirectories. This configuration variable is used on Linux platforms when a job sets **encrypt_execute_directory** to `True` in the submit description file.

SEC_TCP_SESSION_TIMEOUT
The length of time in seconds until the timeout on individual network operations when establishing a UDP security session via TCP. The default value is 20 seconds. Scalability issues with a large pool would be the only basis for a change from the default value.

SEC_TCP_SESSION_DEADLINE
An integer representing the total length of time in seconds until giving up when establishing a security session. Whereas `SEC_TCP_SESSION_TIMEOUT` specifies the timeout for individual blocking operations (connect, read, write), this setting specifies the total time across all operations, including non-blocking operations that have little cost other than holding open the socket. The default value is 120 seconds. The intention of this setting is to avoid waiting for hours for a response in the rare event that the other side freezes up and the socket remains in a connected state. This problem has been observed in some types of operating system crashes.

SEC_DEFAULT_AUTHENTICATION_TIMEOUT
The length of time in seconds that HTCondor should attempt authenticating network connections before giving up. The default imposes no time limit, so the attempt never gives up. Like other security settings, the portion of the configuration variable name, `DEFAULT`, may be replaced by a different access level to specify the timeout to use for different types of commands, for example `SEC_CLIENT_AUTHENTICATION_TIMEOUT`.

SEC_PASSWORD_FILE
For Unix machines, the path and file name of the file containing the pool password for password authentication.

SEC_PASSWORD_DIRECTORY
The path to the directory containing signing key files for token authentication. Defaults to `/etc/condor/passwords.d` on Unix and to `$(RELEASE_DIR)\tokens.sk` on Windows.

TRUST_DOMAIN
An arbitrary string used by the IDTOKENS authentication method; it defaults to . When HTCondor creates an IDTOKEN, it sets the issuer (`iss`) field to this value. When an HTCondor client attempts to authenticate using the IDTOKENS method, it only presents an IDTOKEN to the server if the server's reported issuer matches the token's.

Note that the issuer (`iss`) field is for the _server_. Each IDTOKEN also contains a subject (`sub`) field, which identifies the user. IDTOKENS generated by *condor_token_fetch* will always be of the form `user@UID_DOMAIN`.

If you have configured the same signing key on two different machines, and want tokens issued by one machine to be accepted by the other (e.g. an access point and a central manager), those two machines must have the same value for this setting.

SEC_TOKEN_FETCH_ALLOWED_SIGNING_KEYS

A comma or space -separated list of signing key names that can be used to create a token if requested by *condor_token_fetch*. Defaults to POOL.

SEC_TOKEN_ISSUER_KEY
The default signing key name to use to create a token if requested by *condor_token_fetch*. Defaults to POOL.

SEC_TOKEN_POOL_SIGNING_KEY_FILE
The path and filename for the file containing the default signing key for token authentication. Defaults to /etc/condor/passwords.d/POOL on Unix and to $(RELEASE_DIR)\tokens.sk\POOL on Windows.

SEC_TOKEN_SYSTEM_DIRECTORY
For Unix machines, the path to the directory containing tokens for daemon-to-daemon authentication with the token method. Defaults to /etc/condor/tokens.d.

SEC_TOKEN_DIRECTORY
For Unix machines, the path to the directory containing tokens for user authentication with the token method. Defaults to ~/.condor/tokens.d.

SEC_TOKEN_REVOCATION_EXPR
A ClassAd expression evaluated against tokens during authentication; if SEC_TOKEN_REVOCATION_EXPR is set and evaluates to true, then the token is revoked and the authentication attempt is denied.

SEC_TOKEN_REQUEST_LIMITS
If set, this is a comma-separated list of authorization levels that limit the authorizations a token request can receive. For example, if SEC_TOKEN_REQUEST_LIMITS is set to READ, WRITE, then a token cannot be issued with the authorization DAEMON even if this would otherwise be permissible.

AUTH_SSL_SERVER_CAFILE
The path and file name of a file containing one or more trusted CA's certificates for the server side of a communication authenticating with SSL. On Linux, this defaults to /etc/pki/tls/certs/ca-bundle.crt.

AUTH_SSL_CLIENT_CAFILE
The path and file name of a file containing one or more trusted CA's certificates for the client side of a communication authenticating with SSL. On Linux, this defaults to /etc/pki/tls/certs/ca-bundle.crt.

AUTH_SSL_SERVER_CADIR
The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the server side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

AUTH_SSL_CLIENT_CADIR
The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the client side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

AUTH_SSL_SERVER_CERTFILE
A comma-separated list of filenames to search for a public certificate to be used for the server side of SSL authentication. The first file that contains a valid credential (in combination with AUTH_SSL_SERVER_KEYFILE) will be used.

AUTH_SSL_CLIENT_CERTFILE
The path and file name of the file containing the public certificate for the client side of a communication authenticating with SSL. If no client certificate is provided, then the client may authenticate as the user anonymous@ssl.

AUTH_SSL_SERVER_KEYFILE
A comma-separated list of filenames to search for a private key to be used for the server side of SSL authentication. The first file that contains a valid credential (in combination with AUTH_SSL_SERVER_CERTFILE) will be used.

AUTH_SSL_CLIENT_KEYFILE
The path and file name of the file containing the private key for the client side of a communication authenticating with SSL.

AUTH_SSL_REQUIRE_CLIENT_CERTIFICATE
A boolean value that controls whether the client side of a communication authenticating with SSL must have a credential. If set to `True` and the client doesn't have a credential, then the SSL authentication will fail and other authentication methods will be tried. The default is `False`.

AUTH_SSL_ALLOW_CLIENT_PROXY
A boolean value that controls whether a daemon will accept an X.509 proxy certificate from a client during SSL authentication. The default is `False`.

AUTH_SSL_USE_CLIENT_PROXY_ENV_VAR
A boolean value that controls whether a client checks environment variable *X509_USER_PROXY* for the location the X.509 credential to use for SSL authentication with a daemon. If this parameter is `True` and *X509_USER_PROXY* is set, then that file is used instead of the files specified by *AUTH_SSL_CLIENT_CERTFILE* and *AUTH_SSL_CLIENT_KEYFILE*. The default is `False`.

SSL_SKIP_HOST_CHECK
A boolean variable that controls whether a host check is performed by the client during an SSL authentication of a Condor daemon. This check requires the daemon's host name to match either the "distinguished name" or a subject alternate name embedded in the server's host certificate When the default value of `False` is set, the check is not skipped. When `True`, this check is skipped, and hosts will not be rejected due to a mismatch of certificate and host name.

COLLECTOR_BOOTSTRAP_SSL_CERTIFICATE
A boolean variable that controls whether the *condor_collector* should generate its own CA and host certificate at startup. When `True`, if the SSL certificate file given by `AUTH_SSL_SERVER_CERTFILE` doesn't exist, the *condor_collector* will generate its own CA, then use that CA to generate an SSL host certificate. The certificate and key files are written to the locations given by `AUTH_SSL_SERVER_CERTFILE` and `AUTH_SSL_SERVER_KEYFILE`, respectively. The locations of the CA files are controlled by `TRUST_DOMAIN_CAFILE` and `TRUST_DOMAIN_CAKEY`. The default value is `True` on unix platforms and `False` on Windows.

TRUST_DOMAIN_CAFILE
A path specifying the location of the CA the *condor_collector* will automatically generate if needed when `COLLECTOR_BOOTSTRAP_SSL_CERTIFICATE` is `True`. This CA will be used to generate a host certificate and key if one isn't provided in `AUTH_SSL_SERVER_KEYFILE` . On Linux, this defaults to `/etc/condor/trust_domain_ca.pem`.

TRUST_DOMAIN_CAKEY
A path specifying the location of the private key for the CA generated at `TRUST_DOMAIN_CAFILE`. On Linux, this defaults `/etc/condor/trust_domain_ca_privkey.pem`.

BOOTSTRAP_SSL_SERVER_TRUST
A boolean variable controlling whether tools and daemons automatically trust the SSL host certificate presented on first authentication. When the default of `false` is set, daemons only trust host certificates from known CAs and tools may prompt the user for confirmation if the certificate is not trusted (see `BOOTSTRAP_SSL_SERVER_TRUST_PROMPT_USER`). After the first authentication, the method and certificate are persisted to a `known_hosts` file; subsequent authentications will succeed only if the certificate is unchanged from the one in the `known_hosts` file.

BOOTSTRAP_SSL_SERVER_TRUST_PROMPT_USER
A boolean variable that controls if tools will prompt the user about whether to trust an SSL host certificate from an unknown CA. The default value is `True`.

SEC_SYSTEM_KNOWN_HOSTS
The location of the `known_hosts` file for daemon authentication. This defaults to `/etc/condor/known_hosts` on Linux. Tools will always save their `known_hosts` file inside `$HOME/.condor`.

CERTIFICATE_MAPFILE
A path and file name of the unified map file.

CERTIFICATE_MAPFILE_ASSUME_HASH_KEYS
For HTCondor version 8.5.8 and later. When this is true, the second field of the CERTIFICATE_MAPFILE is not interpreted as a regular expression unless it begins and ends with the slash / character.

SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION
This is a special authentication mechanism designed to minimize overhead in the *condor_schedd* when communicating with the execute machine. When this is enabled, the *condor_negotiator* sends the *condor_schedd* a secret key generated by the *condor_startd*. This key is used to establish a strong security session between the execute and submit daemons without going through the usual security negotiation protocol. This is especially important when operating at large scale over high latency networks (for example, on a pool with one *condor_schedd* daemon and thousands of *condor_startd* daemons on a network with a 0.1 second round trip time).

The default value is True. To have any effect, it must be True in the configuration of both the execute side (*condor_startd*) as well as the submit side (*condor_schedd*). When True, all other security negotiation between the submit and execute daemons is bypassed. All inter-daemon communication between the submit and execute side will use the *condor_startd* daemon's settings for SEC_DAEMON_ENCRYPTION and SEC_DAEMON_INTEGRITY; the configuration of these values in the *condor_schedd*, *condor_shadow*, and *condor_starter* are ignored.

Important: for this mechanism to be secure, integrity and encryption, should be enabled in the startd configuration. Also, some form of strong mutual authentication (e.g. SSL) should be enabled between all daemons and the central manager. Otherwise, the shared secret which is exchanged in matchmaking cannot be safely encrypted when transmitted over the network.

The *condor_schedd* and *condor_shadow* will be authenticated as submit-side@matchsession when they talk to the *condor_startd* and *condor_starter*. The *condor_startd* and *condor_starter* will be authenticated as execute-side@matchsession when they talk to the *condor_schedd* and *condor_shadow*. These identities is automatically added to the DAEMON, READ, and CLIENT authorization levels in these daemons when needed.

This same mechanism is also used to allow the *condor_negotiator* to authenticate with the *condor_schedd*. The submitter ads contain a unique security key; any entity that can obtain the key from the collector (by default, anyone with NEGOTIATOR permission) is authorized to perform negotiation with the *condor_schedd*. This implies, when SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION is enabled, the HTCondor administrator does not need to explicitly setup authentication from the negotiator to the submit host.

SEC_USE_FAMILY_SESSION
The "family" session is a special security session that's shared between an HTCondor daemon and all of its descendant daemons. It allows a family of daemons to communicate securely without an expensive authentication negotiation on each network connection. It bypasses the security authorization settings. The default value is True.

SEC_ENABLE_REMOTE_ADMINISTRATION
A boolean parameter that controls whether daemons should include a secret administration key when they advertise themselves to the **condor_collector**. Anyone with this key is authorized to send ADMINISTRATOR-level commands to the daemon. The **condor_collector** will only provide this key to clients who are authorized at the ADMINISTRATOR level to the **condor_collector**. The default value is True.

When this parameter is enabled for all daemons, control of who is allowed to administer the pool can be consolidated in the **condor_collector** and its security configuration.

KERBEROS_SERVER_KEYTAB
The path and file name of the keytab file that holds the necessary Kerberos principals. If not defined, this variable's value is set by the installed Kerberos; it is /etc/v5srvtab on most systems.

KERBEROS_SERVER_PRINCIPAL
An exact Kerberos principal to use. The default value is $(KERBEROS_SERVER_SERVICE)/<hostname>@<realm>, where KERBEROS_SERVER_SERVICE defaults to host. When both KERBEROS_SERVER_PRINCIPAL and KERBEROS_SERVER_SERVICE are defined, this value takes precedence.

KERBEROS_SERVER_USER
The user name that the Kerberos server principal will map to after authentication. The default value is condor.

KERBEROS_SERVER_SERVICE
A string representing the Kerberos service name. This string is suffixed with a slash character (/) and the host name in order to form the Kerberos server principal. This value defaults to `host`. When both `KERBEROS_SERVER_PRINCIPAL` and `KERBEROS_SERVER_SERVICE` are defined, the value of `KERBEROS_SERVER_PRINCIPAL` takes precedence.

KERBEROS_CLIENT_KEYTAB
The path and file name of the keytab file for the client in Kerberos authentication. This variable has no default value.

SCITOKENS_FILE
The path and file name of a file containing a SciToken for use by the client during the SCITOKENS authentication methods. This variable has no default value. If left unset, HTCondor will use the bearer token discovery protocol defined by the WLCG (https://zenodo.org/record/3937438) to find one.

SEC_CREDENTIAL_SWEEP_DELAY
The number of seconds to wait before cleaning up unused credentials. Defaults to 3,600 seconds (1 hour).

SEC_CREDENTIAL_DIRECTORY_KRB
The directory that users' Kerberos credentials should be stored in. This variable has no default value.

SEC_CREDENTIAL_DIRECTORY_OAUTH
The directory that users' OAuth2 credentials should be stored in. This directory must be owned by root:condor with the setgid flag enabled.

SEC_CREDENTIAL_PRODUCER
A script for *condor_submit* to execute to produce credentials while using the Kerberos type of credentials. No parameters are passed, and credentials most be sent to stdout.

SEC_CREDENTIAL_STORER
A script for *condor_submit* to execute to produce credentials while using the OAuth2 type of credentials. The oauth services specified in the `use_auth_services` line in the submit file are passed as parameters to the script, and the script should use `condor_store_cred` to store credentials for each service. Additional modifiers to each service may be passed: &handle=, &scopes=, or &audience=. The handle should be appended after an underscore to the service name used with `condor_store_cred`, the comma-separated list of scopes should be passed to the command with the -S option, and the audience should be passed to it with the -A option.

LEGACY_ALLOW_SEMANTICS
A boolean parameter that defaults to `False`. In HTCondor 8.8 and prior, if *ALLOW_DAEMON* or *DENY_DAEMON* wasn't set in the configuration files, then the value of *ALLOW_WRITE* or *DENY_DAEMON* (respectively) was used for these parameters. Setting *LEGACY_ALLOW_SEMANTICS* to `True` enables this old behavior. This is a potential security concern, so this setting should only be used to ease the upgrade of an existing pool from 8.8 or prior to 9.0 or later.

5.5.22 Configuration File Entries Relating to Virtual Machines

These macros affect how HTCondor runs **vm** universe jobs on a matched machine within the pool. They specify items related to the *condor_vm-gahp*.

VM_GAHP_SERVER
The complete path and file name of the *condor_vm-gahp*. The default value is `$(SBIN)`/condor_vm-gahp.

VM_GAHP_LOG
The complete path and file name of the *condor_vm-gahp* log. If not specified on a Unix platform, the *con-*

dor_starter log will be used for *condor_vm-gahp* log items. There is no default value for this required configuration variable on Windows platforms.

MAX_VM_GAHP_LOG
Controls the maximum length (in bytes) to which the *condor_vm-gahp* log will be allowed to grow.

VM_TYPE
Specifies the type of supported virtual machine software. It will be the value `kvm` or `xen`. There is no default value for this required configuration variable.

VM_MEMORY
An integer specifying the maximum amount of memory in MiB to be shared among the VM universe jobs run on this machine.

VM_MAX_NUMBER
An integer limit on the number of executing virtual machines. When not defined, the default value is the same `NUM_CPUS`. When it evaluates to `Undefined`, as is the case when not defined with a numeric value, no meaningful limit is imposed.

VM_STATUS_INTERVAL
An integer number of seconds that defaults to 60, representing the interval between job status checks by the *condor_starter* to see if the job has finished. A minimum value of 30 seconds is enforced.

VM_GAHP_REQ_TIMEOUT
An integer number of seconds that defaults to 300 (five minutes), representing the amount of time HTCondor will wait for a command issued from the *condor_starter* to the *condor_vm-gahp* to be completed. When a command times out, an error is reported to the *condor_startd*.

VM_RECHECK_INTERVAL
An integer number of seconds that defaults to 600 (ten minutes), representing the amount of time the *condor_startd* waits after a virtual machine error as reported by the *condor_starter*, and before checking a final time on the status of the virtual machine. If the check fails, HTCondor disables starting any new vm universe jobs by removing the `VM_Type` attribute from the machine ClassAd.

VM_SOFT_SUSPEND
A boolean value that defaults to `False`, causing HTCondor to free the memory of a vm universe job when the job is suspended. When `True`, the memory is not freed.

VM_NETWORKING
A boolean variable describing if networking is supported. When not defined, the default value is `False`.

VM_NETWORKING_TYPE
A string describing the type of networking, required and relevant only when `VM_NETWORKING` is `True`. Defined strings are

```
bridge
nat
nat, bridge
```

VM_NETWORKING_DEFAULT_TYPE
Where multiple networking types are given in `VM_NETWORKING_TYPE`, this optional configuration variable identifies which to use. Therefore, for

```
VM_NETWORKING_TYPE = nat, bridge
```

this variable may be defined as either `nat` or `bridge`. Where multiple networking types are given in `VM_NETWORKING_TYPE`, and this variable is not defined, a default of `nat` is used.

VM_NETWORKING_BRIDGE_INTERFACE
For Xen and KVM only, a required string if bridge networking is to be enabled. It specifies the networking

interface that vm universe jobs will use.

LIBVIRT_XML_SCRIPT
For Xen and KVM only, a path and executable specifying a program. When the *condor_vm-gahp* is ready to start a Xen or KVM **vm** universe job, it will invoke this program to generate the XML description of the virtual machine, which it then provides to the virtualization software. The job ClassAd will be provided to this program via standard input. This program should print the XML to standard output. If this configuration variable is not set, the *condor_vm-gahp* will generate the XML itself. The provided script in `$(LIBEXEC)`/libvirt_simple_script.awk will generate the same XML that the *condor_vm-gahp* would.

LIBVIRT_XML_SCRIPT_ARGS
For Xen and KVM only, the command-line arguments to be given to the program specified by `LIBVIRT_XML_SCRIPT`.

The following configuration variables are specific to the Xen virtual machine software.

XEN_BOOTLOADER
A required full path and executable for the Xen bootloader, if the kernel image includes a disk image.

5.5.23 Configuration File Entries Relating to High Availability

These macros affect the high availability operation of HTCondor.

MASTER_HA_LIST
Similar to `DAEMON_LIST`, this macro defines a list of daemons that the *condor_master* starts and keeps its watchful eyes on. However, the `MASTER_HA_LIST` daemons are run in a High Availability mode. The list is a comma or space separated list of subsystem names (as listed in *Pre-Defined Macros*). For example,

```
MASTER_HA_LIST = SCHEDD
```

The High Availability feature allows for several *condor_master* daemons (most likely on separate machines) to work together to insure that a particular service stays available. These *condor_master* daemons ensure that one and only one of them will have the listed daemons running.

To use this feature, the lock URL must be set with `HA_LOCK_URL`.

Currently, only file URLs are supported (those with `file:...`). The default value for `MASTER_HA_LIST` is the empty string, which disables the feature.

HA_LOCK_URL
This macro specifies the URL that the *condor_master* processes use to synchronize for the High Availability service. Currently, only file URLs are supported; for example, `file:/share/spool`. Note that this URL must be identical for all *condor_master* processes sharing this resource. For *condor_schedd* sharing, we recommend setting up `SPOOL` on an NFS share and having all High Availability *condor_schedd* processes sharing it, and setting the `HA_LOCK_URL` to point at this directory as well. For example:

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
VALID_SPOOL_FILES = SCHEDD.lock
```

A separate lock is created for each High Availability daemon.

There is no default value for `HA_LOCK_URL`.

Lock files are in the form <SUBSYS>.lock. *condor_preen* is not currently aware of the lock files and will delete them if they are placed in the SPOOL directory, so be sure to add <SUBSYS>.lock to VALID_SPOOL_FILES for each High Availability daemon.

HA_<SUBSYS>_LOCK_URL
This macro controls the High Availability lock URL for a specific subsystem as specified in the configuration variable name, and it overrides the system-wide lock URL specified by HA_LOCK_URL. If not defined for each subsystem, HA_<SUBSYS>_LOCK_URL is ignored, and the value of HA_LOCK_URL is used.

List of possible subsystems to set <SUBSYS> can be found at .

HA_LOCK_HOLD_TIME
This macro specifies the number of seconds that the *condor_master* will hold the lock for each High Availability daemon. Upon gaining the shared lock, the *condor_master* will hold the lock for this number of seconds. Additionally, the *condor_master* will periodically renew each lock as long as the *condor_master* and the daemon are running. When the daemon dies, or the *condor_master* exists, the *condor_master* will immediately release the lock(s) it holds.

HA_LOCK_HOLD_TIME defaults to 3600 seconds (one hour).

HA_<SUBSYS>_LOCK_HOLD_TIME
This macro controls the High Availability lock hold time for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by HA_LOCK_HOLD_TIME. If not defined for each subsystem, HA_<SUBSYS>_LOCK_HOLD_TIME is ignored, and the value of HA_LOCK_HOLD_TIME is used.

List of possible subsystems to set <SUBSYS> can be found at .

HA_POLL_PERIOD
This macro specifies how often the *condor_master* polls the High Availability locks to see if any locks are either stale (meaning not updated for HA_LOCK_HOLD_TIME seconds), or have been released by the owning *condor_master*. Additionally, the *condor_master* renews any locks that it holds during these polls.

HA_POLL_PERIOD defaults to 300 seconds (five minutes).

HA_<SUBSYS>_POLL_PERIOD
This macro controls the High Availability poll period for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by HA_POLL_PERIOD. If not defined for each subsystem, HA_<SUBSYS>_POLL_PERIOD is ignored, and the value of HA_POLL_PERIOD is used.

List of possible subsystems to set <SUBSYS> can be found at .

MASTER_<SUBSYS>_CONTROLLER
Used only in HA configurations involving the *condor_had*.

The *condor_master* has the concept of a controlling and controlled daemon, typically with the *condor_had* daemon serving as the controlling process. In this case, all *condor_on* and *condor_off* commands directed at controlled daemons are given to the controlling daemon, which then handles the command, and, when required, sends appropriate commands to the *condor_master* to do the actual work. This allows the controlling daemon to know the state of the controlled daemon.

As of 6.7.14, this configuration variable must be specified for all configurations using *condor_had*. To configure the *condor_negotiator* controlled by *condor_had*:

```
MASTER_NEGOTIATOR_CONTROLLER = HAD
```

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined by .

HAD_LIST
A comma-separated list of all *condor_had* daemons in the form IP:port or hostname:port. Each central manager machine that runs the *condor_had* daemon should appear in this list. If HAD_USE_PRIMARY is set to True, then the first machine in this list is the primary central manager, and all others in the list are backups.

All central manager machines must be configured with an identical HAD_LIST. The machine addresses are identical to the addresses defined in COLLECTOR_HOST.

HAD_USE_PRIMARY
Boolean value to determine if the first machine in the HAD_LIST configuration variable is a primary central manager. Defaults to False.

HAD_CONTROLLEE
This variable is used to specify the name of the daemon which the *condor_had* daemon controls. This name should match the daemon name in the *condor_master* daemon's DAEMON_LIST definition. The default value is NEGOTIATOR.

HAD_CONNECTION_TIMEOUT
The time (in seconds) that the *condor_had* daemon waits before giving up on the establishment of a TCP connection. The failure of the communication connection is the detection mechanism for the failure of a central manager machine. For a LAN, a recommended value is 2 seconds. The use of authentication (by HTCondor) increases the connection time. The default value is 5 seconds. If this value is set too low, *condor_had* daemons will incorrectly assume the failure of other machines.

HAD_ARGS
Command line arguments passed by the *condor_master* daemon as it invokes the *condor_had* daemon. To make high availability work, the *condor_had* daemon requires the port number it is to use. This argument is of the form

```
-p $(HAD_PORT_NUMBER)
```

where HAD_PORT_NUMBER is a helper configuration variable defined with the desired port number. Note that this port number must be the same value here as used in HAD_LIST. There is no default value.

HAD
The path to the *condor_had* executable. Normally it is defined relative to $(SBIN). This configuration variable has no default value.

MAX_HAD_LOG
Controls the maximum length in bytes to which the *condor_had* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 MiB.

HAD_DEBUG
Logging level for the *condor_had* daemon. See <SUBSYS>_DEBUG for values.

HAD_LOG
Full path and file name of the log file. The default value is $(LOG)/HADLog.

HAD_FIPS_MODE
Controls what type of checksum will be sent along with files that are replicated. Set it to 0 for MD5 checksums and to 1 for SHA-2 checksums. Prior to versions 8.8.13 and 8.9.12 only MD5 checksums are supported. In the 10.0 and later release of HTCondor, MD5 support will be removed and only SHA-2 will be supported. This configuration variable is intended to provide a transition between the 8.8 and 9.0 releases. Once all machines in your pool involved in HAD replication have been upgraded to 9.0 or later, you should set the value of this configuration variable to 1. Default value is 0 in HTCondor versions before 9.12 and 1 in version 9.12 and later.

REPLICATION_LIST
A comma-separated list of all *condor_replication* daemons in the form IP:port or hostname:port. Each central manager machine that runs the *condor_had* daemon should appear in this list. All potential central manager machines must be configured with an identical REPLICATION_LIST.

STATE_FILE
A full path and file name of the file protected by the replication mechanism. When not defined, the default path

and file used is

```
$(SPOOL)/Accountantnew.log
```

REPLICATION_INTERVAL
Sets how often the *condor_replication* daemon initiates its tasks of replicating the `$(STATE_FILE)`. It is defined in seconds and defaults to 300 (5 minutes).

MAX_TRANSFER_LIFETIME
A timeout period within which the process that transfers the state file must complete its transfer. The recommended value is `2 * average size of state file / network rate`. It is defined in seconds and defaults to 300 (5 minutes).

HAD_UPDATE_INTERVAL
Like `UPDATE_INTERVAL`, determines how often the *condor_had* is to send a ClassAd update to the *condor_collector*. Updates are also sent at each and every change in state. It is defined in seconds and defaults to 300 (5 minutes).

HAD_USE_REPLICATION
A boolean value that defaults to `False`. When `True`, the use of *condor_replication* daemons is enabled.

REPLICATION_ARGS
Command line arguments passed by the *condor_master* daemon as it invokes the *condor_replication* daemon. To make high availability work, the *condor_replication* daemon requires the port number it is to use. This argument is of the form

```
-p $(REPLICATION_PORT_NUMBER)
```

where `REPLICATION_PORT_NUMBER` is a helper configuration variable defined with the desired port number. Note that this port number must be the same value as used in `REPLICATION_LIST`. There is no default value.

REPLICATION
The full path and file name of the *condor_replication* executable. It is normally defined relative to `$(SBIN)`. There is no default value.

MAX_REPLICATION_LOG
Controls the maximum length in bytes to which the *condor_replication* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix `.old`. The `.old` file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 MiB.

REPLICATION_DEBUG
Logging level for the *condor_replication* daemon. See `<SUBSYS>_DEBUG` for values.

REPLICATION_LOG
Full path and file name to the log file. The default value is `$(LOG)`/ReplicationLog.

TRANSFERER
The full path and file name of the *condor_transferer* executable. The default value is `$(LIBEXEC)`/condor_transferer.

TRANSFERER_LOG
Full path and file name to the log file. The default value is `$(LOG)`/TransfererLog.

TRANSFERER_DEBUG
Logging level for the *condor_transferer* daemon. See `<SUBSYS>_DEBUG` for values.

MAX_TRANSFERER_LOG
Controls the maximum length in bytes to which the *condor_transferer* daemon log will be allowed to grow. A value of 0 specifies that this file may grow without bounds. The default is 1 MiB.

5.5.24 Configuration File Entries Relating to condor_ssh_to_job

These macros affect how HTCondor deals with *condor_ssh_to_job*, a tool that allows users to interactively debug jobs. With these configuration variables, the administrator can control who can use the tool, and how the *ssh* programs are invoked. The manual page for *condor_ssh_to_job* is at *condor_ssh_to_job*.

ENABLE_SSH_TO_JOB
A boolean expression read by the *condor_starter*, that when `True` allows the owner of the job or a queue super user on the *condor_schedd* where the job was submitted to connect to the job via *ssh*. The expression may refer to attributes of both the job and the machine ClassAds. The job ClassAd attributes may be referenced by using the prefix `TARGET.`, and the machine ClassAd attributes may be referenced by using the prefix `MY.`. When `False`, it prevents *condor_ssh_to_job* from starting an *ssh* session. The default value is `True`.

SCHEDD_ENABLE_SSH_TO_JOB
A boolean expression read by the *condor_schedd*, that when `True` allows the owner of the job or a queue super user to connect to the job via *ssh* if the execute machine also allows *condor_ssh_to_job* access (see `ENABLE_SSH_TO_JOB`). The expression may refer to attributes of only the job ClassAd. When `False`, it prevents *condor_ssh_to_job* from starting an *ssh* session for all jobs managed by the *condor_schedd*. The default value is `True`.

SSH_TO_JOB_<SSH-CLIENT>_CMD
A string read by the *condor_ssh_to_job* tool. It specifies the command and arguments to use when invoking the program specified by `<SSH-CLIENT>`. Values substituted for the placeholder `<SSH-CLIENT>` may be SSH, SFTP, SCP, or any other *ssh* client capable of using a command as a proxy for the connection to *sshd*. The entire command plus arguments string is enclosed in double quote marks. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in a *condor_submit* file. The following substitutions are made within the arguments:

> %h: is substituted by the remote host %i: is substituted by the ssh key %k: is substituted by the known hosts file %u: is substituted by the remote user %x: is substituted by a proxy command suitable for use with the *OpenSSH* ProxyCommand option %%: is substituted by the percent mark character

The default string is:

```
"ssh -oUser=%u -oIdentityFile=%i -oStrictHostKeyChecking=yes -oUserKnownHostsFile=%k
-oGlobalKnownHostsFile=%k -oProxyCommand=%x %h"
```

When the `<SSH-CLIENT>` is *scp*, %h is omitted.

SSH_TO_JOB_SSHD
The path and executable name of the *ssh* daemon. The value is read by the *condor_starter*. The default value is `/usr/sbin/sshd`.

SSH_TO_JOB_SSHD_ARGS
A string, read by the *condor_starter* that specifies the command-line arguments to be passed to the *sshd* to handle an incoming ssh connection on its `stdin` or `stdout` streams in inetd mode. Enclose the entire arguments string in double quote marks. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in an HTCondor submit description file. Within the arguments, the characters %f are replaced by the path to the *sshd* configuration file the characters %% are replaced by a single percent character. The default value is the string "-i -e -f %f".

SSH_TO_JOB_SSHD_CONFIG_TEMPLATE
A string, read by the *condor_starter* that specifies the path and file name of an *sshd* configuration template file. The template is turned into an *sshd* configuration file by replacing macros within the template that specify such things as the paths to key files. The macro replacement is

done by the script `$(LIBEXEC)/condor_ssh_to_job_sshd_setup`. The default value is `$(LIB)/condor_ssh_to_job_sshd_config_template`.

SSH_TO_JOB_SSH_KEYGEN
A string, read by the *condor_starter* that specifies the path to *ssh_keygen*, the program used to create ssh keys.

SSH_TO_JOB_SSH_KEYGEN_ARGS
A string, read by the *condor_starter* that specifies the command-line arguments to be passed to the *ssh_keygen* to generate an ssh key. Enclose the entire arguments string in double quotes. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in an HTCondor submit description file. Within the arguments, the characters %f are replaced by the path to the key file to be generated, and the characters %% are replaced by a single percent character. The default value is the string "-N '' -C '' -q -f %f -t rsa". If the user specifies additional arguments with the command condor_ssh_to_job -keygen-options, then those arguments are placed after the arguments specified by the value of `SSH_TO_JOB_SSH_KEYGEN_ARGS`.

5.5.25 condor_rooster Configuration File Macros

condor_rooster is an optional daemon that may be added to the *condor_master* daemon's `DAEMON_LIST`. It is responsible for waking up hibernating machines when their `UNHIBERNATE` expression becomes `True`. In the typical case, a pool runs a single instance of *condor_rooster* on the central manager. However, if the network topology requires that Wake On LAN packets be sent to specific machines from different locations, *condor_rooster* can be run on any machine(s) that can read from the pool's *condor_collector* daemon.

For *condor_rooster* to wake up hibernating machines, the collecting of offline machine ClassAds must be enabled. See variable for details on how to do this.

ROOSTER_INTERVAL
The integer number of seconds between checks for offline machines that should be woken. The default value is 300.

ROOSTER_MAX_UNHIBERNATE
An integer specifying the maximum number of machines to wake up per cycle. The default value of 0 means no limit.

ROOSTER_UNHIBERNATE
A boolean expression that specifies which machines should be woken up. The default expression is `Offline && Unhibernate`. If network topology or other considerations demand that some machines in a pool be woken up by one instance of *condor_rooster*, while others be woken up by a different instance, `ROOSTER_UNHIBERNATE` may be set locally such that it is different for the two instances of *condor_rooster*. In this way, the different instances will only try to wake up their respective subset of the pool.

ROOSTER_UNHIBERNATE_RANK
A ClassAd expression specifying which machines should be woken up first in a given cycle. Higher ranked machines are woken first. If the number of machines to be woken up is limited by `ROOSTER_MAX_UNHIBERNATE` , the rank may be used for determining which machines are woken before reaching the limit.

ROOSTER_WAKEUP_CMD
A string representing the command line invoked by *condor_rooster* that is to wake up a machine. The command and any arguments should be enclosed in double quote marks, the same as **arguments** syntax in an HTCondor submit description file. The default value is "$(BIN)/condor_power -d -i". The command is expected to read from its standard input a ClassAd representing the offline machine.

5.5.26 condor_shared_port Configuration File Macros

These configuration variables affect the *condor_shared_port* daemon. For general discussion of the *condor_shared_port* daemon, see *Reducing Port Usage with the condor_shared_port Daemon*.

USE_SHARED_PORT
A boolean value that specifies whether HTCondor daemons should rely on the *condor_shared_port* daemon for receiving incoming connections. Under Unix, write access to the location defined by DAEMON_SOCKET_DIR is required for this to take effect. The default is True.

SHARED_PORT_PORT
The default TCP port used by the *condor_shared_port* daemon. If COLLECTOR_USES_SHARED_PORT is the default value of True, and the *condor_master* launches a *condor_collector* daemon, then the *condor_shared_port* daemon will ignore this value and use the TCP port assigned to the *condor_collector* via the COLLECTOR_HOST configuration variable.

The default value is $(COLLECTOR_PORT), which defaults to 9618. Note that this causes all HTCondor hosts to use TCP port 9618 by default, differing from previous behavior. The previous behavior has only the *condor_collector* host using a fixed port. To restore this previous behavior, set SHARED_PORT_PORT to 0, which will cause the *condor_shared_port* daemon to use a randomly selected port in the range LOWPORT - HIGHPORT, as defined in *Port Usage in HTCondor*.

SHARED_PORT_DAEMON_AD_FILE
This specifies the full path and name of a file used to publish the address of *condor_shared_port*. This file is read by the other daemons that have USE_SHARED_PORT=True and which are therefore sharing the same port. The default typically does not need to be changed.

SHARED_PORT_MAX_WORKERS
An integer that specifies the maximum number of sub-processes created by *condor_shared_port* while servicing requests to connect to the daemons that are sharing the port. The default is 50.

DAEMON_SOCKET_DIR
This specifies the directory where Unix versions of HTCondor daemons will create named sockets so that incoming connections can be forwarded to them by *condor_shared_port*. If this directory does not exist, it will be created. The maximum length of named socket paths plus names is restricted by the operating system, so using a path that is longer than 90 characters may cause failures.

Write access to this directory grants permission to receive connections through the shared port. By default, the directory is created to be owned by HTCondor and is made to be only writable by HTCondor. One possible reason to broaden access to this directory is if execute nodes are accessed via CCB and the submit node is behind a firewall with only one open port, which is the port assigned to *condor_shared_port*. In this case, commands that interact with the execute node, such as *condor_ssh_to_job*, will not be able to operate unless run by a user with write access to DAEMON_SOCKET_DIR. In this case, one could grant tmp-like permissions to this directory so that all users can receive CCB connections back through the firewall. But, consider the wisdom of having a firewall in the first place, if it will be circumvented in this way.

On Linux platforms, daemons use abstract named sockets instead of normal named sockets. Abstract sockets are not not tied to a file in the file system. The *condor_master* picks a random prefix for abstract socket names and shares it privately with the other daemons. When searching for the recipient of an incoming connection, *condor_shared_port* will check for both an abstract socket and a named socket in the directory indicated by this variable. The named socket allows command-line tools such as *condor_ssh_to_job* to use *condor_shared_port* as described.

On Linux platforms, setting SHARED_PORT_AUDIT_LOG causes HTCondor to log the following information about each connection made through the DAEMON_SOCKET_DIR: the source address, the socket file name, and the target process's PID, UID, GID, executable path, and command line. An administrator may use this logged information to deter abuse.

The default value is `auto`, causing the use of the directory `$(LOCK)/daemon_sock`. On Unix platforms other than Linux, if that path is longer than the 90 characters maximum, then the *condor_master* will instead create a directory under `/tmp` with a name that looks like `/tmp/condor_shared_port_<XXXXXX>`, where `<XXXXXX>` is replaced with random characters. The *condor_master* then tells the other daemons the exact name of the directory it created, and they use it.

If a different value is set for `DAEMON_SOCKET_DIR`, then that directory is used, without regard for the length of the path name. Ensure that the length is not longer than 90 characters.

SHARED_PORT_ARGS
Like all daemons started by the *condor_master* daemon, the command line arguments to the invocation of the *condor_shared_port* daemon can be customized. The arguments can be used to specify a non-default port number for the *condor_shared_port* daemon as in this example, which specifies port 4080:

```
SHARED_PORT_ARGS = -p 4080
```

It is recommended to use configuration variable `SHARED_PORT_PORT` to set a non-default port number, instead of using this configuration variable.

SHARED_PORT_AUDIT_LOG
On Linux platforms, the path and file name of the *condor_shared_port* log that records connections made via the `DAEMON_SOCKET_DIR`. If not defined, there will be no *condor_shared_port* audit log.

MAX_SHARED_PORT_AUDIT_LOG
On Linux platforms, controls the maximum amount of time that the *condor_shared_port* audit log will be allowed to grow. When it is time to rotate a log file, the log file will be saved to a file named with an ISO timestamp suffix. The oldest rotated file receives the file name suffix `.old`. The `.old` files are overwritten each time the maximum number of rotated files (determined by the value of `MAX_NUM_SHARED_PORT_AUDIT_LOG`) is exceeded. A value of 0 specifies that the file may grow without bounds. The following suffixes may be used to qualify the integer:

`Sec` for seconds `Min` for minutes `Hr` for hours `Day` for days `Wk` for weeks

MAX_NUM_SHARED_PORT_AUDIT_LOG
On Linux platforms, the integer that controls the maximum number of rotations that the *condor_shared_port* audit log is allowed to perform, before the oldest one will be rotated away. The default value is 1.

5.5.27 Configuration File Entries Relating to Job Hooks

These macros control the various hooks that interact with HTCondor. Currently, there are two independent sets of hooks. One is a set of fetch work hooks, some of which are invoked by the *condor_startd* to optionally fetch work, and some are invoked by the *condor_starter*. See *Job Hooks That Fetch Work* for more details. The other set replace functionality of the *condor_job_router* daemon. Documentation for the *condor_job_router* daemon is in *The HTCondor Job Router*.

SLOT<N>_JOB_HOOK_KEYWORD
For the fetch work hooks, the keyword used to define which set of hooks a particular compute slot should invoke. The value of <N> is replaced by the slot identification number. For example, on slot 1, the variable name will be called `[SLOT1_JOB_HOOK_KEYWORD`. There is no default keyword. Sites that wish to use these job hooks must explicitly define the keyword and the corresponding hook paths.

STARTD_JOB_HOOK_KEYWORD
For the fetch work hooks, the keyword used to define which set of hooks a particular *condor_startd* should invoke. This setting is only used if a slot-specific keyword is not defined for a given compute slot. There is no default keyword. Sites that wish to use job hooks must explicitly define the keyword and the corresponding hook paths.

<Keyword>_HOOK_FETCH_WORK
For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* wants to fetch work. <Keyword> is the hook keyword defined to distinguish between sets of hooks. There is no default.

<Keyword>_HOOK_REPLY_FETCH
For the fetch work hooks, the full path to the program to invoke when the hook defined by <Keyword>_HOOK_FETCH_WORK returns data and the the *condor_startd* decides if it is going to accept the fetched job or not. <Keyword> is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_REPLY_CLAIM
For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* finishes fetching a job and decides what to do with it. <Keyword> is the hook keyword defined to distinguish between sets of hooks. There is no default.

<Keyword>_HOOK_PREPARE_JOB
For the fetch work hooks, the full path to the program invoked by the *condor_starter* before it runs the job. <Keyword> is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_UPDATE_JOB_INFO
This configuration variable is used by both fetch work hooks and by *condor_job_router* hooks.

For the fetch work hooks, the full path to the program invoked by the *condor_starter* periodically as the job runs, allowing the *condor_starter* to present an updated and augmented job ClassAd to the program. See *Job Hooks That Fetch Work* for the list of additional attributes included. When the job is first invoked, the *condor_starter* will invoke the program after $(STARTER_INITIAL_UPDATE_INTERVAL) seconds. Thereafter, the *condor_starter* will invoke the program every $(STARTER_UPDATE_INTERVAL) seconds. <Keyword> is the hook keyword defined to distinguish between sets of hooks.

As a Job Router hook, the full path to the program invoked when the Job Router polls the status of routed jobs at intervals set by JOB_ROUTER_POLLING_PERIOD. <Keyword> is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

<Keyword>_HOOK_EVICT_CLAIM
For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* needs to evict a fetched claim. <Keyword> is the hook keyword defined to distinguish between sets of hooks. There is no default.

<Keyword>_HOOK_JOB_EXIT
For the fetch work hooks, the full path to the program invoked by the *condor_starter* whenever a job exits, either on its own or when being evicted from an execution slot. <Keyword> is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_JOB_EXIT_TIMEOUT
For the fetch work hooks, the number of seconds the *condor_starter* will wait for the hook defined by <Keyword>_HOOK_JOB_EXIT hook to exit, before continuing with job clean up. Defaults to 30 seconds. <Keyword> is the hook keyword defined to distinguish between sets of hooks.

FetchWorkDelay
An expression that defines the number of seconds that the *condor_startd* should wait after an invocation of <Keyword>_HOOK_FETCH_WORK completes before the hook should be invoked again. The expression is evaluated in the context of the slot ClassAd, and the ClassAd of the currently running job (if any). The expression must evaluate to an integer. If not defined, the *condor_startd* will wait 300 seconds (five minutes) between attempts to fetch work. For more information about this expression, see *Job Hooks That Fetch Work*.

JOB_ROUTER_HOOK_KEYWORD
For the Job Router hooks, the keyword used to define the set of hooks the *condor_job_router* is to invoke to replace functionality of routing translation. There is no default keyword. Use of these hooks requires the explicit definition of the keyword and the corresponding hook paths.

<Keyword>_HOOK_TRANSLATE_JOB
A Job Router hook, the full path to the program invoked when the Job Router has determined that a job meets the

definition for a route. This hook is responsible for doing the transformation of the job. <Keyword> is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

<Keyword>_HOOK_JOB_FINALIZE
A Job Router hook, the full path to the program invoked when the Job Router has determined that the job completed. <Keyword> is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

<Keyword>_HOOK_JOB_CLEANUP
A Job Router hook, the full path to the program invoked when the Job Router finishes managing the job. <Keyword> is the hook keyword defined by JOB_ROUTER_HOOK_KEYWORD to identify the hooks.

5.5.28 Configuration File Entries Relating to Daemon ClassAd Hooks: Startd Cron and Schedd Cron

The following macros describe the daemon ClassAd hooks which run startd cron and schedd cron. These run executables or scripts directly from the *condor_startd* and *condor_schedd* daemons. The output is merged into the ClassAd generated by the respective daemon. The mechanism is described in *Startd Cron and Schedd Cron*.

These macros all include CRON because the default mode for a daemon ClassAd hook is to run periodically. A specific daemon ClassAd hook is called a JOB.

To define a job:

- Add a JobName to . (If you want to define a benchmark, or a daemon ClassAd hook in the schedd, use BENCHMARK or SCHEDD in the macro name instead.) A JobName identifies a specific job and must be unique. In the rest of this section, where <JobName> appears in a macro name, it means to replace <JobName> with one of the names .
- You must set , and you'll probably want to set as well. These macros tell HTCondor how to actually run the job.
- You must also decide when your job will run. By default, a job runs every seconds after the daemon starts up. You may set to change to this to continuously (WaitForExit); on start-up (OneShot) and optionally, when the daemon is reconfigured; or as a benchmark (OnDemand). If you do not select OneShot, you must set .

All the other job-specific macros are optional, of which and are probably the most common.

STARTD_CRON_AUTOPUBLISH
Optional setting that determines if the *condor_startd* should automatically publish a new update to the *condor_collector* after any of the jobs produce output. Beware that enabling this setting can greatly increase the network traffic in an HTCondor pool, especially when many modules are executed, or if the period in which they run is short. There are three possible (case insensitive) values for this variable:

`Never`
This default value causes the *condor_startd* to not automatically publish updates based on any jobs. Instead, updates rely on the usual behavior for sending updates, which is periodic, based on the UPDATE_INTERVAL configuration variable, or whenever a given slot changes state.

`Always`
Causes the *condor_startd* to always send a new update to the *condor_collector* whenever any job exits.

`If_Changed`
Causes the *condor_startd* to only send a new update to the *condor_collector* if the output produced by a given job is different than the previous output of the same job. The only exception is the LastUpdate attribute, which is automatically set for all jobs to be the timestamp when the job last ran. It is ignored when STARTD_CRON_AUTOPUBLISH is set to If_Changed.

STARTD_CRON_CONFIG_VAL and SCHEDD_CRON_CONFIG_VAL and BENCHMARKS_CONFIG_VAL
This configuration variable can be used to specify the path and executable name of the *condor_config_val* program which the jobs (hooks) should use to get configuration information from the daemon. If defined, an environment variable by the same name with the same value will be passed to all jobs.

STARTD_CRON_JOBLIST and SCHEDD_CRON_JOBLIST and BENCHMARKS_JOBLIST
These configuration variables are defined by a comma and/or white space separated list of job names to run. Each is the logical name of a job. This name must be unique; no two jobs may have the same name. The *condor_startd* reads this configuration variable on startup and on reconfig. The *condor_schedd* reads this variable and other SCHEDD_CRON_* variables only on startup.

STARTD_CRON_MAX_JOB_LOAD and SCHEDD_CRON_MAX_JOB_LOAD and BENCHMARKS_MAX_JOB_LOAD
A floating point value representing a threshold for CPU load, such that if starting another job would cause the sum of assumed loads for all running jobs to exceed this value, no further jobs will be started. The default value for STARTD_CRON or a SCHEDD_CRON hook managers is 0.1. This implies that a maximum of 10 jobs (using their default, assumed load) could be concurrently running. The default value for the BENCHMARKS hook manager is 1.0. This implies that only 1 BENCHMARKS job (at the default, assumed load) may be running.

STARTD_CRON_LOG_NON_ZERO_EXIT and SCHEDD_CRON_LOG_NON_ZERO_EXIT
If true, each time a cron job returns a non-zero exit code, the corresponding daemon will log the cron job's exit code and output. There is no default value, so no logging will occur by default.

STARTD_CRON_<JobName>_ARGS and SCHEDD_CRON_<JobName>_ARGS and BENCHMARKS_<JobName>_ARGS
The command line arguments to pass to the job as it is invoked. The first argument will be <JobName>.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_CONDITION
A ClassAd expression evaluated each time the job might otherwise be started. If this macro is set, but the expression does not evaluate to True, the job will not be started. The expression is evaluated in a context similar to a slot ad, but without any slot-specific attributes.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST.

STARTD_CRON_<JobName>_CWD and SCHEDD_CRON_<JobName>_CWD and BENCHMARKS_<JobName>_CWD
The working directory in which to start the job.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_ENV and SCHEDD_CRON_<JobName>_ENV and BENCHMARKS_<JobName>_ENV
The environment string to pass to the job. The syntax is the same as that of <DaemonName>_ENVIRONMENT as defined at *condor_master Configuration File Macros*.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_EXECUTABLE and SCHEDD_CRON_<JobName>_EXECUTABLE and BENCHMARKS_<JobName>_EXECUTABLE
The full path and executable to run for this job. Note that multiple jobs may specify the same executable, although the jobs need to have different logical names.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_JOB_LOAD and SCHEDD_CRON_<JobName>_JOB_LOAD and BENCHMARKS_<JobName>_JOB_LOAD
A floating point value that represents the assumed and therefore expected CPU load that a job induces on the system. This job load is then used to limit the total number of jobs that run concurrently, by not starting new jobs if the assumed total load from all jobs is over a set threshold. The default value for each individual STARTD_CRON or a SCHEDD_CRON job is 0.01. The default value for each individual BENCHMARKS job is 1.0.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_KILL and SCHEDD_CRON_<JobName>_KILL and BENCHMARKS_<JobName>_KILL
A boolean value applicable only for jobs with a MODE of anything other than WaitForExit. The default value is False.

This variable controls the behavior of the daemon hook manager when it detects that an instance of the job's executable is still running as it is time to invoke the job again. If True, the daemon hook manager will kill the currently running job and then invoke an new instance of the job. If False, the existing job invocation is allowed to continue running.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_METRICS
A space or comma -separated list. Each element in the list is a metric type, either SUM or PEAK; a colon; and a metric name.

An attribute preceded by SUM is a metric which accumulates over time. The canonical example is seconds of CPU usage.

An attribute preceded by PEAK is a metric which instead records the largest value reported over the period of use. The canonical example is megabytes of memory usage.

A job with STARTD_CRON_<JobName>_METRICS set is a custom machine resource monitor (CMRM), and its output is handled differently than a normal job's. A CMRM should output one ad per custom machine resource instance and use SlotMergeConstraints (see *Startd Cron and Schedd Cron*) to specify the instance to which it applies.

The ad corresponding to each custom machine resource instance should have an attribute for each metric named in the configuration. For SUM metrics, the attribute should be Uptime<MetricName>Seconds; for PEAK metrics, the attribute should be Uptime<MetricName>PeakUsage.

Each value should be the value of the metric since the last time the job reported. The reported value may therefore go up or down; HTCondor will record either the the sum or the peak value, as appropriate, for the duration of the job running in a slot assigned resources of the corresponding type.

For example, if your custom resources are SQUIDs, and you detected four of them, your monitor might output the following:

```
SlotMergeConstraint = StringListMember( "SQUID0", AssignedSQUIDs )
UptimeSQUIDsSeconds = 5.0
UptimeSQUIDsMemoryPeakUsage = 50
- SQUIDsReport0
SlotMergeConstraint = StringListMember( "SQUID1", AssignedSQUIDs )
UptimeSQUIDsSeconds = 1.0
UptimeSQUIDsMemoryPeakUsage = 10
- SQUIDsReport1
SlotMergeConstraint = StringListMember( "SQUID2", AssignedSQUIDs )
UptimeSQUIDsSeconds = 9.0
```

(continues on next page)

(continued from previous page)

```
UptimeSQUIDsMemoryPeakUsage = 90
- SQUIDsReport2
SlotMergeConstraint = StringListMember( "SQUID3", AssignedSQUIDs )
UptimeSQUIDsSeconds = 4.0
UptimeSQUIDsMemoryPeakUsage = 40
- SQUIDsReport3
```

The names ('SQUIDsReport0') may be anything, but must be consistent from report to report and the ClassAd for each report must have a distinct name.

You might specify the monitor in the example above as follows:

```
MACHINE_RESOURCE_INVENTORY_SQUIDs = /usr/local/bin/cmr-squid-discovery

STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) SQUIDs_MONITOR
STARTD_CRON_SQUIDs_MONITOR_MODE = Periodic
STARTD_CRON_SQUIDs_MONITOR_PERIOD = 10
STARTD_CRON_SQUIDs_MONITOR_EXECUTABLE = /usr/local/bin/cmr-squid-monitor
STARTD_CRON_SQUIDs_MONITOR_METRICS = SUM:SQUIDs, PEAK:SQUIDsMemory
```

STARTD_CRON_<JobName>_MODE and SCHEDD_CRON_<JobName>_MODE and BENCHMARKS_<JobName>_MODE
A string that specifies a mode within which the job operates. Legal values are

- `Periodic`, which is the default.
- `WaitForExit`
- `OneShot`
- `OnDemand`

<JobName> is the logical name assigned for a job as defined by configuration variable `STARTD_CRON_JOBLIST`, `SCHEDD_CRON_JOBLIST`, or `BENCHMARKS_JOBLIST`.

The default `Periodic` mode is used for most jobs. In this mode, the job is expected to be started by the *condor_startd* daemon, gather and publish its data, and then exit.

In `WaitForExit` mode the *condor_startd* daemon interprets the period as defined by `STARTD_CRON_<JobName>_PERIOD` differently. In this case, it refers to the amount of time to wait after the job exits before restarting it. With a value of 1, the job is kept running nearly continuously. In general, `WaitForExit` mode is for jobs that produce a periodic stream of updated data, but it can be used for other purposes, as well. The output data from the job is accumulated into a temporary ClassAd until the job exits or until it writes a line starting with dash (-) character. At that point, the temporary ClassAd replaces the active ClassAd for the job. The active ClassAd for the job is merged into the appropriate slot ClassAds whenever the slot ClassAds are published.

The `OneShot` mode is used for jobs that are run once at the start of the daemon. If the `reconfig_rerun` option is specified, the job will be run again after any reconfiguration.

The `OnDemand` mode is used only by the `BENCHMARKS` mechanism. All benchmark jobs must be be `OnDemand` jobs. Any other jobs specified as `OnDemand` will never run. Additional future features may allow for other `OnDemand` job uses.

STARTD_CRON_<JobName>_PERIOD and SCHEDD_CRON_<JobName>_PERIOD and BENCHMARKS_<JobName>_PERIOD
The period specifies time intervals at which the job should be run. For periodic jobs, this is the time interval that passes between starting the execution of the job. The value may be specified in seconds, minutes, or hours. Specify this time by appending the character `s`, `m`, or `h` to the value. As an example, 5m starts the execution of the

job every five minutes. If no character is appended to the value, seconds are used as a default. In `WaitForExit` mode, the value has a different meaning: the period specifies the length of time after the job ceases execution and before it is restarted. The minimum valid value of the period is 1 second.

<JobName> is the logical name assigned for a job as defined by configuration variable `STARTD_CRON_JOBLIST`, `SCHEDD_CRON_JOBLIST`, or `BENCHMARKS_JOBLIST`.

STARTD_CRON_<JobName>_PREFIX and SCHEDD_CRON_<JobName>_PREFIX and BENCHMARKS_<JobName>_PREFIX
Specifies a string which is prepended by HTCondor to all attribute names that the job generates. The use of prefixes avoids the conflicts that would be caused by attributes of the same name generated and utilized by different jobs. For example, if a module prefix is `xyz_`, and an individual attribute is named `abc`, then the resulting attribute name will be `xyz_abc`. Due to restrictions on ClassAd names, a prefix is only permitted to contain alpha-numeric characters and the underscore character.

<JobName> is the logical name assigned for a job as defined by configuration variable `STARTD_CRON_JOBLIST`, `SCHEDD_CRON_JOBLIST`, or `BENCHMARKS_JOBLIST`.

STARTD_CRON_<JobName>_RECONFIG and SCHEDD_CRON_<JobName>_RECONFIG
A boolean value that when `True`, causes the daemon to send an HUP signal to the job when the daemon is reconfigured. The job is expected to reread its configuration at that time.

<JobName> is the logical name assigned for a job as defined by configuration variable `STARTD_CRON_JOBLIST` or `SCHEDD_CRON_JOBLIST`.

STARTD_CRON_<JobName>_RECONFIG_RERUN and SCHEDD_CRON_<JobName>_RECONFIG_RERUN
A boolean value that when `True`, causes the daemon ClassAd hook mechanism to re-run the specified job when the daemon is reconfigured via *condor_reconfig*. The default value is `False`.

<JobName> is the logical name assigned for a job as defined by configuration variable `STARTD_CRON_JOBLIST` or `SCHEDD_CRON_JOBLIST`.

STARTD_CRON_<JobName>_SLOTS and BENCHMARKS_<JobName>_SLOTS
Only the slots specified in this comma-separated list may incorporate the output of the job specified by <JobName>. If the list is not specified, any slot may. Whether or not a specific slot actually incorporates the output depends on the output; see *Startd Cron and Schedd Cron*.

<JobName> is the logical name assigned for a job as defined by configuration variable `STARTD_CRON_JOBLIST` or `BENCHMARKS_JOBLIST`.

5.5.29 Configuration File Entries Only for Windows Platforms

These macros are utilized only on Windows platforms.

WINDOWS_RMDIR
The complete path and executable name of the HTCondor version of the built-in *rmdir* program. The HTCondor version will not fail when the directory contains files that have ACLs that deny the SYSTEM process delete access. If not defined, the built-in Windows *rmdir* program is invoked, and a value defined for `WINDOWS_RMDIR_OPTIONS` is ignored.

WINDOWS_RMDIR_OPTIONS
Command line options to be specified when configuration variable `WINDOWS_RMDIR` is defined. Defaults to **/S /C** when configuration variable `WINDOWS_RMDIR` is defined and its definition contains the string `"condor_rmdir.exe"`.

5.5.30 condor_defrag Configuration File Macros

These configuration variables affect the *condor_defrag* daemon. A general discussion of *condor_defrag* may be found in *condor_startd Policy Configuration*.

DEFRAG_NAME
Used to give an prefix value to the `Name` attribute in the *condor_defrag* daemon's ClassAd. Defaults to the name given in the :macro:`DAEMON_LIST`. This esoteric configuration macro might be used in the situation where there are two *condor_defrag* daemons running on one machine, and each reports to the same *condor_collector*. Different names will distinguish the two daemons. See the description of `MASTER_NAME` in *condor_master Configuration File Macros* for defaults and composition of valid HTCondor daemon names.

DEFRAG_DRAINING_MACHINES_PER_HOUR
A floating point number that specifies how many machines should be drained per hour. The default is 0, so no draining will happen unless this setting is changed. Each *condor_startd* is considered to be one machine. The actual number of machines drained per hour may be less than this if draining is halted by one of the other defragmentation policy controls. The granularity in timing of draining initiation is controlled by `DEFRAG_INTERVAL` . The lowest rate of draining that is supported is one machine per day or one machine per `DEFRAG_INTERVAL` , whichever is lower. A fractional number of machines contributing to the value of `DEFRAG_DRAINING_MACHINES_PER_HOUR` is rounded to the nearest whole number of machines on a per day basis.

DEFRAG_DRAINING_START_EXPR
A ClassAd expression that replaces the machine's `START` expression while it's draining. Slots which accepted a job after the machine began draining set the machine ad attribute `AcceptedWhileDraining` to `true`. When the last job which was not accepted while draining exits, all other jobs are immediately evicted with a `MaxJobRetirementTime` of 0; job vacate times are still respected. While the jobs which were accepted while draining are vacating, the `START` expression is `false`. Using `$(START)` in this expression is usually a mistake: it will be replaced by the defrag daemon's `START` expression, not the value of the target machine's `START` expression (and especially not the value of its `START` expression at the time draining begins).

DEFRAG_REQUIREMENTS
An expression that narrows the selection of which machines to drain. By default *condor_defrag* will drain all machines that are drainable. A machine, meaning a *condor_startd*, is matched if any of its partitionable slots match this expression. Machines are automatically excluded if they cannot be drained, are already draining, or if they match `DEFRAG_WHOLE_MACHINE_EXPR` .

The *condor_defrag* daemon will always add the following requirements to `DEFRAG_REQUIREMENTS`

```
PartitionableSlot && Offline =!= true && Draining =!= true
```

DEFRAG_CANCEL_REQUIREMENTS
An expression that is periodically evaluated against machines that are draining. When this expression evaluates to `True`, draining will be cancelled. This defaults to `$(DEFRAG_WHOLE_MACHINE_EXPR)` . This could be used to drain partial rather than whole machines. Beginning with version 8.9.11, only machines that have no `DrainReason` or a value of `"Defrag"` for `DrainReason` will be checked to see if draining should be cancelled. Beginning with 10.7.0 The Defrag daemon will also check for its own name in the `DrainReason`.

DEFRAG_RANK
An expression that specifies which machines are more desirable to drain. The expression should evaluate to a number for each candidate machine to be drained. If the number of machines to be drained is less than the number of candidates, the machines with higher rank will be chosen. The rank of a machine, meaning a *condor_startd*, is the rank of its highest ranked slot. The default rank is `-ExpectedMachineGracefulDrainingBadput`.

DEFRAG_WHOLE_MACHINE_EXPR
An expression that specifies which machines are already operating as whole machines. The default is

```
Cpus == TotalSlotCpus
```

A machine is matched if any Partitionable slot on the machine matches this expression and the machine is not draining or was drained by *condor_defrag*. Each *condor_startd* is considered to be one machine. Whole machines are excluded when selecting machines to drain. They are also counted against `DEFRAG_MAX_WHOLE_MACHINES`.

DEFRAG_MAX_WHOLE_MACHINES
An integer that specifies the maximum number of whole machines. When the number of whole machines is greater than or equal to this, no new machines will be selected for draining. Each *condor_startd* is counted as one machine. The special value -1 indicates that there is no limit. The default is -1.

DEFRAG_MAX_CONCURRENT_DRAINING
An integer that specifies the maximum number of draining machines. When the number of machines that are draining is greater than or equal to this, no new machines will be selected for draining. Each draining *condor_startd* is counted as one machine. The special value -1 indicates that there is no limit. The default is -1.

DEFRAG_INTERVAL
An integer that specifies the number of seconds between evaluations of the defragmentation policy. In each cycle, the state of the pool is observed and machines are drained, if specified by the policy. The default is 600 seconds. Very small intervals could create excessive load on the *condor_collector*.

DEFRAG_UPDATE_INTERVAL
An integer that specifies the number of seconds between times that the *condor_defrag* daemon sends updates to the collector. (See *Defrag ClassAd Attributes* for information about the attributes in these updates.) The default is 300 seconds.

DEFRAG_SCHEDULE
A setting that specifies the draining schedule to use when draining machines. Possible values are `graceful`, `quick`, and `fast`. The default is `graceful`.

graceful
Initiate a graceful eviction of the job. This means all promises that have been made to the job are honored, including `MaxJobRetirementTime`. The eviction of jobs is coordinated to reduce idle time. This means that if one slot has a job with a long retirement time and the other slots have jobs with shorter retirement times, the effective retirement time for all of the jobs is the longer one.

quick
`MaxJobRetirementTime` is not honored. Eviction of jobs is immediately initiated. Jobs are given time to shut down according to the usual policy, as given by `MachineMaxVacateTime`.

fast
Jobs are immediately hard-killed, with no chance to gracefully shut down.

DEFRAG_LOG
The path to the *condor_defrag* daemon's log file. The default log location is `$(LOG)/DefragLog`.

5.5.31 *condor_gangliad* Configuration File Macros

condor_gangliad is an optional daemon responsible for publishing information about HTCondor daemons to the Ganglia™ monitoring system. The Ganglia monitoring system must be installed and configured separately. In the typical case, a single instance of the *condor_gangliad* daemon is run per pool. A default set of metrics are sent. Additional metrics may be defined, in order to publish any information available in ClassAds that the *condor_collector* daemon has.

GANGLIAD_INTERVAL
The integer number of seconds between consecutive sending of metrics to Ganglia. Daemons update the *condor_collector* every 300 seconds, and the Ganglia heartbeat interval is 20 seconds. Therefore, multiples of 20 between 20 and 300 makes sense for this value. Negative values inhibit sending data to Ganglia. The default value is 60.

GANGLIAD_MIN_METRIC_LIFETIME
An integer value representing the minimum DMAX value for all metrics. Where DMAX is the number number of seconds without updating that a metric will be kept before deletion. This value defaults to `86400` which is equivalent to 1 day. This value will be overridden by a specific metric defined `Lifetime` value.

GANGLIAD_VERBOSITY
An integer that specifies the maximum verbosity level of metrics to be published to Ganglia. Basic metrics have a verbosity level of 0, which is the default. Additional metrics can be enabled by increasing the verbosity to 1. In the default configuration, there are no metrics with verbosity levels higher than 1. Some metrics depend on attributes that are not published to the *condor_collector* when using the default value of `STATISTICS_TO_PUBLISH` . For example, per-user file transfer statistics will only be published to Ganglia if `GANGLIA_VERBOSITY` is set to 1 or higher in the *condor_gangliad* configuration and `STATISTICS_TO_PUBLISH` in the *condor_schedd* configuration contains `TRANSFER:2`, or if the `STATISTICS_TO_PUBLISH_LIST` contains the desired attributes explicitly.

GANGLIAD_REQUIREMENTS
An optional boolean ClassAd expression that may restrict the set of daemon ClassAds to be monitored. This could be used to monitor a subset of a pool's daemons or machines. The default is an empty expression, which has the effect of placing no restriction on the monitored ClassAds. Keep in mind that this expression is applied to all types of monitored ClassAds, not just machine ClassAds.

GANGLIAD_PER_EXECUTE_NODE_METRICS
A boolean value that, when `False`, causes metrics from execute node daemons to not be published. Aggregate values from these machines will still be published. The default value is `True`. This option is useful for pools such that use glidein, in which it is not desired to record metrics for individual execute nodes.

GANGLIA_CONFIG
The path and file name of the Ganglia configuration file. The default is `/etc/ganglia/gmond.conf`.

GANGLIA_GMETRIC
The full path of the *gmetric* executable to use. If none is specified, `libganglia` will be used instead when possible, because the library interface is more efficient than invoking *gmetric*. Some versions of `libganglia` are not compatible. When a failure to use `libganglia` is detected, *gmetric* will be used, if *gmetric* can be found in HTCondor's `PATH` environment variable.

GANGLIA_GSTAT_COMMAND
The full *gstat* command used to determine which hosts are monitored by Ganglia. For a *condor_gangliad* running on a host whose local *gmond* does not know the list of monitored hosts, change `localhost` to be the appropriate host name or IP address within this default string:

```
gstat --all --mpifile --gmond_ip=localhost --gmond_port=8649
```

GANGLIA_SEND_DATA_FOR_ALL_HOSTS
A boolean value that when `True` causes data to be sent to Ganglia for hosts that it is not currently monitoring.

The default is `False`.

GANGLIA_LIB
The full path and file name of the `libganglia` shared library to use. If none is specified, and if configuration variable `GANGLIA_GMETRIC` is also not specified, then a search for `libganglia` will be performed in the directories listed in configuration variable `GANGLIA_LIB_PATH` or `GANGLIA_LIB64_PATH` . The special value `NOOP` indicates that *condor_gangliad* should not publish statistics to Ganglia, but should otherwise go through all the motions it normally does.

GANGLIA_LIB_PATH
A comma-separated list of directories within which to search for the `libganglia` executable, if `GANGLIA_LIB` is not configured. This is used in 32-bit versions of HTCondor.

GANGLIA_LIB64_PATH
A comma-separated list of directories within which to search for the `libganglia` executable, if `GANGLIA_LIB` is not configured. This is used in 64-bit versions of HTCondor.

GANGLIAD_DEFAULT_CLUSTER
An expression specifying the default name of the Ganglia cluster for all metrics. The expression may refer to attributes of the machine.

GANGLIAD_DEFAULT_MACHINE
An expression specifying the default machine name of Ganglia metrics. The expression may refer to attributes of the machine.

GANGLIAD_DEFAULT_IP
An expression specifying the default IP address of Ganglia metrics. The expression may refer to attributes of the machine.

GANGLIAD_LOG
The path and file name of the *condor_gangliad* daemon's log file. The default log is `$(LOG)/GangliadLog`.

GANGLIAD_METRICS_CONFIG_DIR
Path to the directory containing files which define Ganglia metrics in terms of HTCondor ClassAd attributes to be published. All files in this directory are read, to define the metrics. The default directory `/etc/condor/ganglia.d/` is used when not specified.

5.5.32 *condor_annex* Configuration File Macros

See *HTCondor Annex Configuration* for *condor_annex* configuration file macros.

5.6 User Priorities and Negotiation

HTCondor uses priorities to determine machine allocation for jobs. This section details the priorities and the allocation of machines (negotiation).

For accounting purposes, each user is identified by username@uid_domain. Each user is assigned a priority value even if submitting jobs from different machines in the same domain, or even if submitting from multiple machines in the different domains.

The numerical priority value assigned to a user is inversely related to the goodness of the priority. A user with a numerical priority of 5 gets more resources than a user with a numerical priority of 50. There are two priority values assigned to HTCondor users:

- Real User Priority (RUP), which measures resource usage of the user.
- Effective User Priority (EUP), which determines the number of resources the user can get.

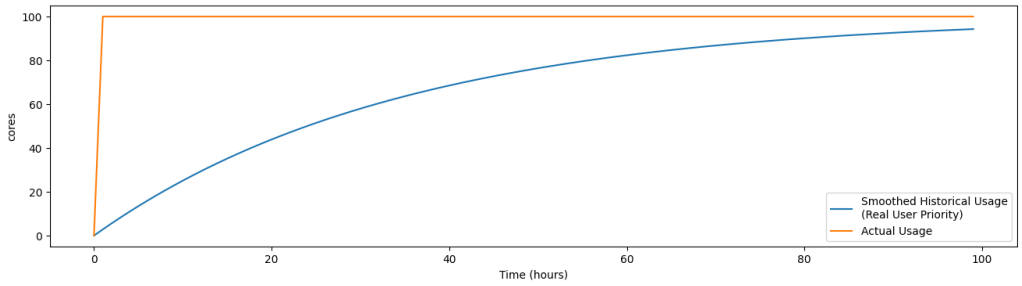
This section describes these two priorities and how they affect resource allocations in HTCondor. Documentation on configuring and controlling priorities may be found in the *condor_negotiator Configuration File Entries* section.

5.6.1 Real User Priority (RUP)

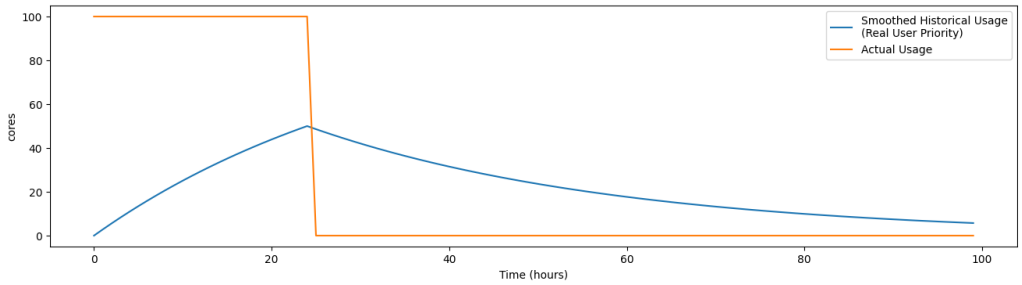
A user's RUP reports a smoothed average of the number of cores a user has used over some recent period of time. Every user begins with a RUP of one half (0.5), which is the lowest possible value. At steady state, the RUP of a user equilibrates to the number of cores currently used. So, if a specific user continuously uses exactly ten cores for a long period of time, the RUP of that user asymptotically approaches ten.

However, if the user decreases the number of cores used, the RUP asymptotically lowers to the new value. The rate at which the priority value decays can be set by the macro , a time period defined in seconds. Intuitively, if the in a pool is set to the default of 86400 seconds (one day), and a user with a RUP of 10 has no running jobs, that user's RUP would be 5 one day later, 2.5 two days later, and so on.

For example, if a new user has no historical usage, their RUP will start at 0.5 If that user then has 100 cores running, their RUP will grow as the graph below show:



Or, if a new user with no historical usage has 100 cores running for 24 hours, then removes all the jobs, so has no cores running, their RUP will grow and shrink as shown below:

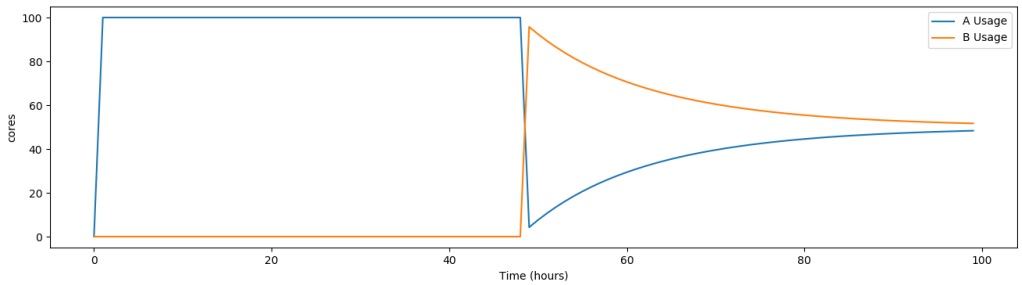


5.6.2 Effective User Priority (EUP)

The effective user priority (EUP) of a user is used to determine how many cores a user should receive. The EUP is simply the RUP multiplied by a priority factor the administrator can set per-user. The default initial priority factor for all new users as they first submit jobs is set by the configuration variable , and defaults to 1000.0. An administrator can change this priority factor using the *condor_userprio* command. For example, setting the priority factor of some user to 2,000 will grant that user twice as many cores as a user with the default priority factor of 1,000, assuming they both have the same historical usage.

The number of resources that a user may receive is inversely related to the ratio between the EUPs of submitting users. User A with EUP=5 will receive twice as many resources as user B with EUP=10 and four times as many resources as user C with EUP=20. However, if A does not use the full number of resources that A may be given, the available resources are repartitioned and distributed among remaining users according to the inverse ratio rule.

Assume two users with no history, named A and B, using a pool with 100 cores. To simplify the math, also assume both users have an equal priority factor of 1.0. User A submits a very large number of short-running jobs at time t = 0 zero. User B waits until 48 hours later, and also submits an infinite number of short jobs. At the beginning, the EUP doesn't matter, as there is only one user with jobs, and so user A gets the whole pool. At the 48 hour mark, both users compete for the pool. Assuming the default of 24 hours, user A's RUP should be about 75.0 at the 48 hour mark, and User B will still be the minimum of .5. At that instance, User B deserves 150 times User A. However, this ratio will decay quickly. User A's share of the pool will drop from all 100 cores to less than one core immediately, but will quickly rebound to a handful of cores, and will asymptotically approach half of the pool as User B gets the inverse. A graph of these two users might look like this:



HTCondor supplies mechanisms to directly support two policies in which EUP may be useful:

Nice users
: A job may be submitted with the submit command **nice_user** set to `True`. This nice user job will have its RUP boosted by the priority factor specified in the configuration, leading to a very large EUP. This corresponds to a low priority for resources, therefore using resources not used by other HTCondor users.

Remote Users
: HTCondor's flocking feature (see the *Connecting HTCondor Pools with Flocking* section) allows jobs to run in a pool other than the local one. In addition, the submit-only feature allows a user to submit jobs to another pool. In such situations, submitters from other domains can submit to the local pool. It may be desirable to have HTCondor treat local users preferentially over these remote users. If configured, HTCondor will boost the RUPs of remote users by specified in the configuration, thereby lowering their priority for resources.

The priority boost factors for individual users can be set with the **setfactor** option of *condor_userprio*. Details may be found in the *condor_userprio* manual page.

5.6.3 Priorities in Negotiation and Preemption

Priorities are used to ensure that users get their fair share of resources. The priority values are used at allocation time, meaning during negotiation and matchmaking. Therefore, there are ClassAd attributes that take on defined values only during negotiation, making them ephemeral. In addition to allocation, HTCondor may preempt a machine claim and reallocate it when conditions change.

Too many preemptions lead to thrashing, a condition in which negotiation for a machine identifies a new job with a better priority most every cycle. Each job is, in turn, preempted, and no job finishes. To avoid this situation, the configuration variable is defined for and used only by the *condor_negotiator* daemon to specify the conditions that must be met for a preemption to occur. When preemption is enabled, it is usually defined to deny preemption if a current running job has been running for a relatively short period of time. This effectively limits the number of preemptions per resource per time interval. Note that only applies to preemptions due to user priority. It does not have any effect if the machine's expression prefers a different job, or if the machine's policy causes the job to vacate due to other activity on the machine. See the *condor_startd Policy Configuration* section for the current default policy on preemption.

The following ephemeral attributes may be used within policy definitions. Care should be taken when using these attributes, due to their ephemeral nature; they are not always defined, so the usage of an expression to check if defined such as

```
(RemoteUserPrio =?= UNDEFINED)
```

is likely necessary.

Within these attributes, those with names that contain the string `Submitter` refer to characteristics about the candidate job's user; those with names that contain the string `Remote` refer to characteristics about the user currently using the resource. Further, those with names that end with the string `ResourcesInUse` have values that may change within the time period associated with a single negotiation cycle. Therefore, the configuration variables and exist to inform the *condor_negotiator* daemon that values may change. See the *condor_negotiator Configuration File Entries* section for definitions of these configuration variables.

SubmitterUserPrio
A floating point value representing the user priority of the candidate job.

SubmitterUserResourcesInUse
The integer number of slots currently utilized by the user submitting the candidate job.

RemoteUserPrio
A floating point value representing the user priority of the job currently running on the machine. This version of the attribute, with no slot represented in the attribute name, refers to the current slot being evaluated.

Slot<N>_RemoteUserPrio
A floating point value representing the user priority of the job currently running on the particular slot represented by <N> on the machine.

RemoteUserResourcesInUse
The integer number of slots currently utilized by the user of the job currently running on the machine.

SubmitterGroupResourcesInUse
If the owner of the candidate job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots currently utilized by the group.

SubmitterGroup
The accounting group name of the requesting submitter.

SubmitterGroupQuota
If the owner of the candidate job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots defined as the group's quota.

RemoteGroupResourcesInUse
If the owner of the currently running job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots currently utilized by the group.

RemoteGroup
The accounting group name of the owner of the currently running job.

RemoteGroupQuota
If the owner of the currently running job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots defined as the group's quota.

SubmitterNegotiatingGroup
The accounting group name that the candidate job is negotiating under.

RemoteNegotiatingGroup
The accounting group name that the currently running job negotiated under.

SubmitterAutoregroup
Boolean attribute is `True` if candidate job is negotiated via autoregroup.

RemoteAutoregroup
Boolean attribute is `True` if currently running job negotiated via autoregroup.

5.6.4 Priority Calculation

This section may be skipped if the reader so feels, but for the curious, here is HTCondor's priority calculation algorithm.

The RUP of a user u at time t, $\pi_r(u,t)$, is calculated every time interval δt using the formula

$$\pi_r(u,t) = \beta\pi_r(u,t-\delta t) + (1-\beta)\rho(u,t)$$

where $\rho(u,t)$ is the number of resources used by user u at time t, and $\beta = 0.5^{\delta t/h}$. h is the half life period set by .

The EUP of user u at time t, $\pi_e(u,t)$ is calculated by

$$\pi_e(u,t) = \pi_r(u,t) \times f(u,t)$$

where $f(u,t)$ is the priority boost factor for user u at time t.

As mentioned previously, the RUP calculation is designed so that at steady state, each user's RUP stabilizes at the number of resources used by that user. The definition of β ensures that the calculation of $\pi_r(u,t)$ can be calculated over non-uniform time intervals δt without affecting the calculation. The time interval δt varies due to events internal to the system, but HTCondor guarantees that unless the central manager machine is down, no matches will be unaccounted for due to this variance.

5.6.5 Negotiation

Negotiation is the method HTCondor undergoes periodically to match queued jobs with resources capable of running jobs. The *condor_negotiator* daemon is responsible for negotiation.

During a negotiation cycle, the *condor_negotiator* daemon accomplishes the following ordered list of items.

1. Build a list of all possible resources, regardless of the state of those resources.
2. Obtain a list of all job submitters (for the entire pool).
3. Sort the list of all job submitters based on EUP (see *The Layperson's Description of the Pie Spin and Pie Slice* for an explanation of EUP). The submitter with the best priority is first within the sorted list.

4. Iterate until there are either no more resources to match, or no more jobs to match.

   For each submitter (in EUP order):

   For each submitter, get each job. Since jobs may be submitted from more than one machine (hence to more than one *condor_schedd* daemon), here is a further definition of the ordering of these jobs. With jobs from a single *condor_schedd* daemon, jobs are typically returned in job priority order. When more than one *condor_schedd* daemon is involved, they are contacted in an undefined order. All jobs from a single *condor_schedd* daemon are considered before moving on to the next. For each job:

   - For each machine in the pool that can execute jobs:
     1. If `machine.requirements` evaluates to `False` or `job.requirements` evaluates to `False`, skip this machine
     2. If the machine is in the Claimed state, but not running a job, skip this machine.
     3. If this machine is not running a job, add it to the potential match list by reason of No Preemption.
     4. If the machine is running a job
        - If the `machine.RANK` on this job is better than the running job, add this machine to the potential match list by reason of Rank.
        - If the EUP of this job is better than the EUP of the currently running job, and is `True`, and the `machine.RANK` on this job is not worse than the currently running job, add this machine to the potential match list by reason of Priority. See example below.
   - Of machines in the potential match list, sort by , `job.RANK`, , Reason for claim (No Preemption, then Rank, then Priority),
   - The job is assigned to the top machine on the potential match list. The machine is removed from the list of resources to match (on this negotiation cycle).

As described above, the *condor_negotiator* tries to match each job to all slots in the pool. Assume that five slots match one request for three jobs, and that their , `Job.Rank`, and expressions evaluate (in the context of both the slot ad and the job ad) to the following values.

Slot Name	NEGOTIATOR_PRE_JOB_RANK	Job.Rank	NEGOTIATOR_POST_JOB_RANK
slot1	100	1	10
slot2	100	2	20
slot3	100	2	30
slot4	0	1	40
slot5	200	1	50

Table 3.1: Example of slots before sorting

These slots would be sorted first on , then sorting all ties based on `Job.Rank` and any remaining ties sorted by . After that, the first three slots would be handed to the *condor_schedd*. This means that is very strong, and overrides any ranking expression by the submitter of the job. After sorting, the slots would look like this, and the schedd would be given slot5, slot3 and slot2:

Slot Name	NEGOTIATOR_PRE_JOB_RANK	Job.Rank	NEGOTIATOR_POST_JOB_RANK
slot5	200	1	50
slot3	100	2	30
slot2	100	2	20
slot1	100	1	10
slot4	0	1	40

Table 3.2: Example of slots after sorting

The *condor_negotiator* asks the *condor_schedd* for the "next job" from a given submitter/user. Typically, the *condor_schedd* returns jobs in the order of job priority. If priorities are the same, job submission time is used; older jobs go first. If a cluster has multiple procs in it and one of the jobs cannot be matched, the *condor_schedd* will not return any more jobs in that cluster on that negotiation pass. This is an optimization based on the theory that the cluster jobs are similar. The configuration variable disables the cluster-skipping optimization. Use of the configuration variable will change the definition of what the *condor_schedd* considers a cluster from the default definition of all jobs that share the same `ClusterId`.

5.6.6 The Layperson's Description of the Pie Spin and Pie Slice

The negotiator first finds all users who have submitted jobs and calculates their priority. Then, it totals the SlotWeight (by default, cores) of all currently available slots, and using the ratios of the user priorities, it calculates the number of cores each user could get. This is their pie slice. (See: SLOT_WEIGHT in *condor_startd Configuration File Macros*)

If any users have a floor defined via *condor_userprio* -set-floor , and their current allocation of cores is below the floor, a special round of the below-floor users goes first, attempting to allocate up to the defined number of cores for their floor level. These users are negotiated for in user priority order. This allows an admin to give users some "guaranteed" minimum number of cores, no matter what their previous usage or priority is.

After the below-floor users are negotiated for, all users are negotiated for, in user priority order. The *condor_negotiator* contacts each schedd where the user's job lives, and asks for job information. The *condor_schedd* daemon (on behalf of a user) tells the matchmaker about a job, and the matchmaker looks at available slots to create a list that match the requirements expression. It then sorts the matching slots by the rank expressions within ClassAds. If a slot prefers a job via the slot RANK expression, the job is assigned to that slot, potentially preempting an already running job. Otherwise, give the slot to the job that the job ranks highest. If the highest ranked slot is already running a job, the negotiator may preempt the running job for the new job.

This matchmaking cycle continues until the user has received all of the machines in their pie slice. If there is a per-user ceiling defined with the *condor_userprio* -setceil command, and this ceiling is smaller than the pie slice, the user gets only up to their ceiling number of cores. The matchmaker then contacts the next highest priority user and offers that user their pie slice worth of machines. After contacting all users, the cycle is repeated with any still available resources and recomputed pie slices. The matchmaker continues spinning the pie until it runs out of machines or all the *condor_schedd* daemons say they have no more jobs.

5.6.7 Group Accounting

By default, HTCondor does all accounting on a per-user basis. This means that HTCondor keeps track of the historical usage per-user, calculates a priority and fair-share per user, and allows the administrator to change this fair-share per user. In HTCondor terminology, the accounting principal is called the submitter.

The name of this submitter is, by default, the name the schedd authenticated when the job was first submitted to the schedd. Usually, this is the operating system username. However, the submitter can override the username selected by setting the submit file option

```
accounting_group_user = ishmael
```

This means this job should be treated, for accounting purposes only, as "ishamel", but "ishmael" will not be the operating system id the shadow or job uses. Note that HTCondor trusts the user to set this to a valid value. The administrator can use schedd requirements or transforms to validate such settings, if desired. accounting_group_user is frequently used in web portals, where one trusted operating system process submits jobs on behalf of different users.

Note that if many people submit jobs with identical accounting_group_user values, HTCondor treats them as one set of jobs for accounting purposes. So, if Alice submits 100 jobs as accounting_group_user ishmael, and so does Bob a moment later, HTCondor will not try to fair-share between them, as it would do if they had not set accounting_group_user. If all these jobs have identical requirements, they will be run First-In, First-Out, so whoever submitted first makes the subsequent jobs wait until the last one of the first submit is finished.

5.6.8 Accounting Groups with Hierarchical Group Quotas

With additional configuration, it is possible to create accounting groups, where the submitters within the group maintain their distinct identity, and fair-share still happens within members of that group.

An upper limit on the number of slots allocated to a group of users can be specified with group quotas.

Consider an example pool with thirty slots: twenty slots are owned by the physics group and ten are owned by the chemistry group. The desired policy is that no more than twenty concurrent jobs are ever running from the physicists, and only ten from the chemists. These machines are otherwise identical, so it does not matter which machines run which group's jobs. It only matters that the proportions of allocated slots are correct.

Group quotas may implement this policy. Define the groups and set their quotas in the configuration of the central manager:

```
GROUP_NAMES = group_physics, group_chemistry
GROUP_QUOTA_group_physics =   20
GROUP_QUOTA_group_chemistry = 10
```

The implementation of quotas is hierarchical, such that quotas may be described for the tree of groups, subgroups, sub subgroups, etc. Group names identify the groups, such that the configuration can define the quotas in terms of limiting the number of cores allocated for a group or subgroup. Group names do not need to begin with `"group_"`, but that is the convention, which helps to avoid naming conflicts between groups and subgroups. The hierarchy is identified by using the period ('.') character to separate a group name from a subgroup name from a sub subgroup name, etc. Group names are case-insensitive for negotiation.

At the root of the tree that defines the hierarchical groups is the "<none>" group. The implied quota of the "<none>" group will be all available slots. This string will appear in the output of *condor_status*.

If the sum of the child quotas exceeds the parent, then the child quotas are scaled down in proportion to their relative sizes. For the given example, there were 30 original slots at the root of the tree. If a power failure removed half of the

original 30, leaving fifteen slots, physics would be scaled back to a quota of ten, and chemistry to five. This scaling can be disabled by setting the *condor_negotiator* configuration variable to `True`. If the sum of the child quotas is less than that of the parent, the child quotas remain intact; they are not scaled up. That is, if somehow the number of slots doubled from thirty to sixty, physics would still be limited to 20 slots, and chemistry would be limited to 10. This example in which the quota is defined by absolute values is called a static quota.

Each job must state which group it belongs to. By default, this is opt-in, and the system trusts each user to put the correct group in the submit description file. See "Setting Accounting Groups Automatically below" to configure the system to set them without user input and to prevent users from opting into the wrong groups. Jobs that do not identify themselves as a group member are negotiated for as part of the "<none>" group. Note that this requirement is per job, not per user. A given user may be a member of many groups. Jobs identify which group they are in by setting the **accounting_group** and **accounting_group_user** commands within the submit description file, as specified in the *Group Accounting* section. For example:

```
accounting_group = group_physics
accounting_group_user = einstein
```

The size of the quotas may instead be expressed as a proportion. This is then referred to as a dynamic group quota, because the size of the quota is dynamically recalculated every negotiation cycle, based on the total available size of the pool. Instead of using static quotas, this example can be recast using dynamic quotas, with one-third of the pool allocated to chemistry and two-thirds to physics. The quotas maintain this ratio even as the size of the pool changes, perhaps because of machine failures, because of the arrival of new machines within the pool, or because of other reasons. The job submit description files remain the same. Configuration on the central manager becomes:

```
GROUP_NAMES = group_physics, group_chemistry
GROUP_QUOTA_DYNAMIC_group_chemistry = 0.33
GROUP_QUOTA_DYNAMIC_group_physics =   0.66
```

The values of the quotas must be less than 1.0, indicating fractions of the pool's machines. As with static quota specification, if the sum of the children exceeds one, they are scaled down proportionally so that their sum does equal 1.0. If their sum is less than one, they are not changed.

Extending this example to incorporate subgroups, assume that the physics group consists of high-energy (hep) and low-energy (lep) subgroups. The high-energy sub-group owns fifteen of the twenty physics slots, and the low-energy group owns the remainder. Groups are distinguished from subgroups by an intervening period character (.) in the group's name. Static quotas for these subgroups extend the example configuration:

```
GROUP_NAMES = group_physics, group_physics.hep, group_physics.lep, group_chemistry
GROUP_QUOTA_group_physics     =   20
GROUP_QUOTA_group_physics.hep =   15
GROUP_QUOTA_group_physics.lep =    5
GROUP_QUOTA_group_chemistry   =   10
```

This hierarchy may be more useful when dynamic quotas are used. Here is the example, using dynamic quotas:

```
GROUP_NAMES = group_physics, group_physics.hep, group_physics.lep, group_chemistry
GROUP_QUOTA_DYNAMIC_group_chemistry   =   0.33334
GROUP_QUOTA_DYNAMIC_group_physics     =   0.66667
GROUP_QUOTA_DYNAMIC_group_physics.hep =   0.75
GROUP_QUOTA_DYNAMIC_group_physics.lep =   0.25
```

The fraction of a subgroup's quota is expressed with respect to its parent group's quota. That is, the high-energy physics subgroup is allocated 75% of the 66% that physics gets of the entire pool, however many that might be. If there are 30 machines in the pool, that would be the same 15 machines as specified in the static quota example.

High-energy physics users indicate which group their jobs should go in with the submit description file identification:

```
accounting_group = group_physics.hep
accounting_group_user = higgs
```

In all these examples so far, the hierarchy is merely a notational convenience. Each of the examples could be implemented with a flat structure, although it might be more confusing for the administrator. Surplus is the concept that creates a true hierarchy.

If a given group or sub-group accepts surplus, then that given group is allowed to exceed its configured quota, by using the leftover, unused quota of other groups. Surplus is disabled for all groups by default. Accepting surplus may be enabled for all groups by setting to `True`. Surplus may be enabled for individual groups by setting to `True`. Consider the following example:

```
GROUP_NAMES = group_physics, group_physics.hep, group_physics.lep, group_chemistry
GROUP_QUOTA_group_physics     =   20
GROUP_QUOTA_group_physics.hep =   15
GROUP_QUOTA_group_physics.lep =    5
GROUP_QUOTA_group_chemistry   =   10
GROUP_ACCEPT_SURPLUS = false
GROUP_ACCEPT_SURPLUS_group_physics = false
GROUP_ACCEPT_SURPLUS_group_physics.lep = true
GROUP_ACCEPT_SURPLUS_group_physics.hep = true
```

This configuration is the same as above for the chemistry users. However, is set to `False` globally, `False` for the physics parent group, and `True` for the subgroups group_physics.lep and group_physics.lep. This means that group_physics.lep and group_physics.hep are allowed to exceed their quota of 15 and 5, but their sum cannot exceed 20, for that is their parent's quota. If the group_physics had set to `True`, then either group_physics.lep and group_physics.hep would not be limited by quota.

Surplus slots are distributed bottom-up from within the quota tree. That is, any leaf nodes of this tree with excess quota will share it with any peers which accept surplus. Any subsequent excess will then be passed up to the parent node and over to all of its children, recursively. Any node that does not accept surplus implements a hard cap on the number of slots that the sum of it's children use.

After the *condor_negotiator* calculates the quota assigned to each group, possibly adding in surplus, it then negotiates with the *condor_schedd* daemons in the system to try to match jobs to each group. It does this one group at a time. By default, it goes in "starvation group order." That is, the group whose current usage is the smallest fraction of its quota goes first, then the next, and so on. The "<none>" group implicitly at the root of the tree goes last. This ordering can be replaced by defining configuration variable . The *condor_negotiator* evaluates this ClassAd expression for each group ClassAd, sorts the groups by the floating point result, and then negotiates with the smallest positive value going first. Available attributes for sorting with include:

Attribute Name	Description
AccountingGroup	A string containing the group name
GroupQuota	The computed limit for this group
GroupResourcesInUse	The total slot weight used by this group
GroupResourcesAllocated	Quota allocated this cycle

Table 3.3: Attributes visible to GROUP_SORT_EXPR

One possible group quota policy is strict priority. For example, a site prefers physics users to match as many slots as they can, and only when all the physics jobs are running, and idle slots remain, are chemistry jobs allowed to run. The default "starvation group order" can be used to implement this. By setting configuration variable to `True`, and setting the physics quota to a number so large that it cannot ever be met, such as one million, the physics group will always be the "most starving" group, will always negotiate first, and will always be unable to meet the quota. Only when all the

physics jobs are running will the chemistry jobs then run. If the chemistry quota is set to a value smaller than physics, but still larger than the pool, this policy can support a third, even lower priority group, and so on.

The *condor_userprio* command can show the current quotas in effect, and the current usage by group. For example:

```
$ condor_userprio -quotas
Last Priority Update: 11/12 15:18
Group                    Effective  Config     Use    Subtree  Requested
Name                       Quota     Quota   Surplus   Quota   Resources
------------------------ --------- --------- ------- --------- ----------
group_physics.hep            15.00     15.00 no           15.00         60
group_physics.lep             5.00      5.00 no            5.00         60
------------------------ --------- --------- ------- --------- ----------
Number of users: 2                                ByQuota
```

This shows that there are two groups, each with 60 jobs in the queue. group_physics.hep has a quota of 15 machines, and group_physics.lep has 5 machines. Other options to *condor_userprio*, such as **-most** will also show the number of resources in use.

5.6.9 Setting Accounting Group automatically per user

By default, any user can put the jobs into any accounting group by setting parameters in the submit file. This can be useful if a person is a member of multiple groups. However, many sites want to force all jobs submitted by a given user into one accounting group, and forbid the user to submit to any other group. An HTCondor metaknob makes this easy. By adding to the access point's configuration, the setting

```
USE Feature: AssignAccountingGroup(file_name_of_map)
```

The admin can create a file that maps the users into their required accounting groups, and makes the attributes immutable, so they can't be changed. The format of this map file is like other classad map files: Lines of three columns. The first should be an asterisk *. The second column is the name of the user, and the final is the accounting group that user should always submit to. For example,

```
* Alice     group_physics
* Bob       group_atlas
* Carol group_physics
* /^student_.*/     group_students
```

The second field can be a regular expression, if enclosed in //. Note that this is on the submit side, and the administrator will still need to create these group names and give them a quota on the central manager machine. This file is re-read on a *condor_reconfig*. The third field can also be a comma-separated list. If so, it represents the set of valid accounting groups a user can opt into. If the user does not set an accounting group in the submit file the first entry in the list will be used.

5.6.10 Running Multiple Negotiators in One Pool

Usually, a single HTCondor pool will have a single *condor_collector* instance running and a single *condor_negotiator* instance running. However, there are special situation where you may want to run more than one *condor_negotiator* against a *condor_collector*, and still consider it one pool.

In such a scenario, each *condor_negotiator* is responsible for some non-overlapping partition of the slots in the pool. This might be for performance – if you have more than 100,000 slots in the pool, you may need to shard this pool into several smaller sections in order to lower the time each negotiator spends. Because accounting is done at the the negotiator level, you may want to do this to have separate accounting and distinct fair share between different kinds of machines in your pool. For example, let's say you have some GPU machines and non-GPU machines, and you want usage of the non-GPU machine to not "count" against the fair-share usage of GPU machines. One way to do this would be to have a separate negotiator for the GPU machines vs the non-GPU machines. At Wisconsin, we have a separate, small subset of our pool for quick-starting interactive jobs. By allocating a negotiator to only negotiate for these few machines, we can speed up the time to match these machines to interactive users who submit with *condor_submit -i*.

Sharding the negotiator is straightforward. Simply add the NEGOTIATOR entry to the on an additional machine. While is is possible to run multiple negotiators on one machine, we may not want to, if we are trying to improve performance. Then, in each negotiator, set to only match those slots this negotiator should use.

Running with multiple negotiators also means you need to be careful with the *condor_userprio* command. As there is no default negotiator, you should always name the specific negotiator you want to *condor_userprio* to talk to with the *-name* option.

5.7 Policy Configuration for Execution Points and for Access Points

Note: Configuration templates make it easier to implement certain policies; see information on policy templates here: *Available Configuration Templates*.

5.7.1 *condor_startd* Policy Configuration

This section describes the configuration of machines, such that they, through the *condor_startd* daemon, implement a desired policy for when remote jobs should start, be suspended, (possibly) resumed, vacate or be killed. This policy is the heart of HTCondor's balancing act between the needs and wishes of resource owners (machine owners) and resource users (people submitting their jobs to HTCondor). Please read this section carefully before changing any of the settings described here, as a wrong setting can have a severe impact on either the owners of machines in the pool or the users of the pool.

condor_startd Terminology

Understanding the configuration requires an understanding of ClassAd expressions, which are detailed in the *HTCondor's ClassAd Mechanism* section.

Each machine runs one *condor_startd* daemon. Each machine may contain one or more cores (or CPUs). The HTCondor construct of a slot describes the unit which is matched to a job. Each slot may contain one or more integer number of cores. Each slot is represented by its own machine ClassAd, distinguished by the machine ClassAd attribute `Name`, which is of the form `slot<N>@hostname`. The value for `<N>` will also be defined with machine ClassAd attribute `SlotID`.

Each slot has its own machine ClassAd, and within that ClassAd, its own state and activity. Other policy expressions are propagated or inherited from the machine configuration by the *condor_startd* daemon, such that all slots have the same policy from the machine configuration. This requires configuration expressions to incorporate the `SlotID` attribute when policy is intended to be individualized based on a slot. So, in this discussion of policy expressions, where a machine is referenced, the policy can equally be applied to a slot.

The *condor_startd* daemon represents the machine on which it is running to the HTCondor pool. The daemon publishes characteristics about the machine in the machine's ClassAd to aid matchmaking with resource requests. The values of these attributes may be listed by using the command:

```
$ condor_status -l hostname
```

The `START` Expression

The most important expression to the *condor_startd* is the expression. This expression describes the conditions that must be met for a machine or slot to run a job. This expression can reference attributes in the machine's ClassAd (such as `KeyboardIdle` and `LoadAvg`) and attributes in a job ClassAd (such as `Owner`, `Imagesize`, and `Cmd`, the name of the executable the job will run). The value of the `START` expression plays a crucial role in determining the state and activity of a machine.

The `Requirements` expression is used for matching machines with jobs.

In situations where a machine wants to make itself unavailable for further matches, the `Requirements` expression is set to `False`. When the `START` expression locally evaluates to `True`, the machine advertises the `Requirements` expression as `True` and does not publish the `START` expression.

Normally, the expressions in the machine ClassAd are evaluated against certain request ClassAds in the *condor_negotiator* to see if there is a match, or against whatever request ClassAd currently has claimed the machine. However, by locally evaluating an expression, the machine only evaluates the expression against its own ClassAd. If an expression cannot be locally evaluated (because it references other expressions that are only found in a request ClassAd, such as `Owner` or `Imagesize`), the expression is (usually) undefined. See theh *HTCondor's ClassAd Mechanism* section for specifics on how undefined terms are handled in ClassAd expression evaluation.

A note of caution is in order when modifying the `START` expression to reference job ClassAd attributes. When using the `POLICY : Desktop` configuration template, the `IS_OWNER` expression is a function of the `START` expression:

```
START =?= FALSE
```

See a detailed discussion of the `IS_OWNER` expression in *condor_startd Policy Configuration*. However, the machine locally evaluates the `IS_OWNER` expression to determine if it is capable of running jobs for HTCondor. Any job ClassAd attributes appearing in the `START` expression, and hence in the `IS_OWNER` expression, are undefined in this context, and may lead to unexpected behavior. Whenever the `START` expression is modified to reference job ClassAd attributes, the `IS_OWNER` expression should also be modified to reference only machine ClassAd attributes.

Note: If you have machines with lots of real memory and swap space such that the only scarce resource is CPU time, consider defining so that HTCondor starts jobs on the machine with low priority. Then, further configure to set up the machines with:

```
START = True
SUSPEND = False
PREEMPT = False
KILL = False
```

In this way, HTCondor jobs always run and can never be kicked off from activity on the machine. However, because they would run with the low priority, interactive response on the machines will not suffer. A machine user probably

would not notice that HTCondor was running the jobs, assuming you had enough free memory for the HTCondor jobs such that there was little swapping.

The RANK Expression

A machine may be configured to prefer certain jobs over others using the RANK expression. It is an expression, like any other in a machine ClassAd. It can reference any attribute found in either the machine ClassAd or a job ClassAd. The most common use of this expression is likely to configure a machine to prefer to run jobs from the owner of that machine, or by extension, a group of machines to prefer jobs from the owners of those machines.

For example, imagine there is a small research group with 4 machines called tenorsax, piano, bass, and drums. These machines are owned by the 4 users coltrane, tyner, garrison, and jones, respectively.

Assume that there is a large HTCondor pool in the department, and this small research group has spent a lot of money on really fast machines for the group. As part of the larger pool, but to implement a policy that gives priority on the fast machines to anyone in the small research group, set the RANK expression on the machines to reference the Owner attribute and prefer requests where that attribute matches one of the people in the group as in

```
RANK = Owner == "coltrane" || Owner == "tyner" \
    || Owner == "garrison" || Owner == "jones"
```

The RANK expression is evaluated as a floating point number. However, like in C, boolean expressions evaluate to either 1 or 0 depending on if they are True or False. So, if this expression evaluated to 1, because the remote job was owned by one of the preferred users, it would be a larger value than any other user for whom the expression would evaluate to 0.

A more complex RANK expression has the same basic set up, where anyone from the group has priority on their fast machines. Its difference is that the machine owner has better priority on their own machine. To set this up for Garrison's machine (bass), place the following entry in the local configuration file of machine bass:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
    + ((Owner == "garrison") * 10) + (Owner == "jones")
```

Note that the parentheses in this expression are important, because the + operator has higher default precedence than ==.

The use of + instead of || allows us to distinguish which terms matched and which ones did not. If anyone not in the research group quartet was running a job on the machine called bass, the RANK would evaluate numerically to 0, since none of the boolean terms evaluates to 1, and 0+0+0+0 still equals 0.

Suppose Elvin Jones submits a job. His job would match the bass machine, assuming START evaluated to True for him at that time. The RANK would numerically evaluate to 1. Therefore, the Elvin Jones job could preempt the HTCondor job currently running. Further assume that later Jimmy Garrison submits a job. The RANK evaluates to 10 on machine bass, since the boolean that matches gets multiplied by 10. Due to this, Jimmy Garrison's job could preempt Elvin Jones' job on the bass machine where Jimmy Garrison's jobs are preferred.

The RANK expression is not required to reference the Owner of the jobs. Perhaps there is one machine with an enormous amount of memory, and others with not much at all. Perhaps configure this large-memory machine to prefer to run jobs with larger memory requirements:

```
RANK = ImageSize
```

That's all there is to it. The bigger the job, the more this machine wants to run it. It is an altruistic preference, always servicing the largest of jobs, no matter who submitted them. A little less altruistic is the RANK on Coltrane's machine that prefers John Coltrane's jobs over those with the largest Imagesize:

```
RANK = (Owner == "coltrane" * 1000000000000) + Imagesize
```

This RANK does not work if a job is submitted with an image size of more 10^{12} Kbytes. However, with that size, this RANK expression preferring that job would not be HTCondor's only problem!

Machine States

A machine is assigned a state by HTCondor. The state depends on whether or not the machine is available to run HTCondor jobs, and if so, what point in the negotiations has been reached. The possible states are

Owner
The machine is being used by the machine owner, and/or is not available to run HTCondor jobs. When the machine first starts up, it begins in this state.

Unclaimed
The machine is available to run HTCondor jobs, but it is not currently doing so.

Matched
The machine is available to run jobs, and it has been matched by the negotiator with a specific schedd. That schedd just has not yet claimed this machine. In this state, the machine is unavailable for further matches.

Claimed
The machine has been claimed by a schedd.

Preempting
The machine was claimed by a schedd, but is now preempting that claim for one of the following reasons.

1. the owner of the machine came back
2. another user with higher priority has jobs waiting to run
3. another request that this resource would rather serve was found

Backfill
The machine is running a backfill computation while waiting for either the machine owner to come back or to be matched with an HTCondor job. This state is only entered if the machine is specifically configured to enable backfill jobs.

Drained
The machine is not running jobs, because it is being drained. One reason a machine may be drained is to consolidate resources that have been divided in a partitionable slot. Consolidating the resources gives large jobs a chance to run.

Fig. 4: Machine states and the possible transitions between the states

Each transition is labeled with a letter. The cause of each transition is described below.

- Transitions out of the Owner state

  A
  The machine switches from Owner to Unclaimed whenever the START expression no longer locally evaluates to FALSE. This indicates that the machine is potentially available to run an HTCondor job.

N

The machine switches from the Owner to the Drained state whenever draining of the machine is initiated, for example by *condor_drain* or by the *condor_defrag* daemon.

- Transitions out of the Unclaimed state

B

The machine switches from Unclaimed back to Owner whenever the `START` expression locally evaluates to FALSE. This indicates that the machine is unavailable to run an HTCondor job and is in use by the resource owner.

C

The transition from Unclaimed to Matched happens whenever the *condor_negotiator* matches this resource with an HTCondor job.

D

The transition from Unclaimed directly to Claimed also happens if the *condor_negotiator* matches this resource with an HTCondor job. In this case the *condor_schedd* receives the match and initiates the claiming protocol with the machine before the *condor_startd* receives the match notification from the *condor_negotiator*.

E

The transition from Unclaimed to Backfill happens if the machine is configured to run backfill computations (see the *Setting Up for Special Environments* section) and the `START_BACKFILL` expression evaluates to TRUE.

P

The transition from Unclaimed to Drained happens if draining of the machine is initiated, for example by *condor_drain* or by the *condor_defrag* daemon.

- Transitions out of the Matched state

F

The machine moves from Matched to Owner if either the `START` expression locally evaluates to FALSE, or if the timer expires. This timeout is used to ensure that if a machine is matched with a given *condor_schedd*, but that *condor_schedd* does not contact the *condor_startd* to claim it, that the machine will give up on the match and become available to be matched again. In this case, since the `START` expression does not locally evaluate to FALSE, as soon as transition **F** is complete, the machine will immediately enter the Unclaimed state again (via transition **A**). The machine might also go from Matched to Owner if the *condor_schedd* attempts to perform the claiming protocol but encounters some sort of error. Finally, the machine will move into the Owner state if the *condor_startd* receives a *condor_vacate* command while it is in the Matched state.

G

The transition from Matched to Claimed occurs when the *condor_schedd* successfully completes the claiming protocol with the *condor_startd*.

- Transitions out of the Claimed state

H

From the Claimed state, the only possible destination is the Preempting state. This transition can be caused by many reasons:

  - The *condor_schedd* that has claimed the machine has no more work to perform and releases the claim
  - The expression evaluates to `True` (which usually means the resource owner has started using the machine again and is now using the keyboard, mouse, CPU, etc.)
  - The *condor_startd* receives a *condor_vacate* command

  - The *condor_startd* is told to shutdown (either via a signal or a *condor_off* command)
  - The resource is matched to a job with a better priority (either a better user priority, or one where the machine rank is higher)

- Transitions out of the Preempting state

  I

  The resource will move from Preempting back to Claimed if the resource was matched to a job with a better priority.

  J

  The resource will move from Preempting to Owner if the `PREEMPT` expression had evaluated to TRUE, if *condor_vacate* was used, or if the `START` expression locally evaluates to FALSE when the *condor_startd* has finished evicting whatever job it was running when it entered the Preempting state.

- Transitions out of the Backfill state

  K

  The resource will move from Backfill to Owner for the following reasons:

  - The expression evaluates to TRUE
  - The *condor_startd* receives a *condor_vacate* command
  - The *condor_startd* is being shutdown

  L

  The transition from Backfill to Matched occurs whenever a resource running a backfill computation is matched with a *condor_schedd* that wants to run an HTCondor job.

  M

  The transition from Backfill directly to Claimed is similar to the transition from Unclaimed directly to Claimed. It only occurs if the *condor_schedd* completes the claiming protocol before the *condor_startd* receives the match notification from the *condor_negotiator*.

- Transitions out of the Drained state

  O

  The transition from Drained to Owner state happens when draining is finalized or is canceled. When a draining request is made, the request either asks for the machine to stay in a Drained state until canceled, or it asks for draining to be automatically finalized once all slots have finished draining.

The Claimed State and Leases

When a *condor_schedd* claims a *condor_startd*, there is a claim lease. So long as the keep alive updates from the *condor_schedd* to the *condor_startd* continue to arrive, the lease is reset. If the lease duration passes with no updates, the *condor_startd* drops the claim and evicts any jobs the *condor_schedd* sent over.

The alive interval is the amount of time between, or the frequency at which the *condor_schedd* sends keep alive updates to all *condor_schedd* daemons. An alive update resets the claim lease at the *condor_startd*. Updates are UDP packets.

Initially, as when the *condor_schedd* starts up, the alive interval starts at the value set by the configuration variable . It may be modified when a job is started. The job's ClassAd attribute `JobLeaseDuration` is checked. If the value of `JobLeaseDuration/3` is less than the current alive interval, then the alive interval is set to either this lower value or the imposed lowest limit on the alive interval of 10 seconds. Thus, the alive interval starts at `ALIVE_INTERVAL` and goes down, never up.

If a claim lease expires, the *condor_startd* will drop the claim. The length of the claim lease is the job's ClassAd attribute `JobLeaseDuration`. `JobLeaseDuration` defaults to 40 minutes time, except when explicitly set within the job's submit description file. If `JobLeaseDuration` is explicitly set to 0, or it is not set as may be the case for a Web Services job that does not define the attribute, then `JobLeaseDuration` is given the Undefined value. Further, when undefined, the claim lease duration is calculated with `MAX_CLAIM_ALIVES_MISSED * alive interval`. The alive interval is the current value, as sent by the *condor_schedd*. If the *condor_schedd* reduces the current alive interval, it does not update the *condor_startd*.

Machine Activities

Within some machine states, activities of the machine are defined. The state has meaning regardless of activity. Differences between activities are significant. Therefore, a "state/activity" pair describes a machine. The following list describes all the possible state/activity pairs.

- Owner

  Idle
  This is the only activity for Owner state. As far as HTCondor is concerned the machine is Idle, since it is not doing anything for HTCondor.

- Unclaimed

  Idle
  This is the normal activity of Unclaimed machines. The machine is still Idle in that the machine owner is willing to let HTCondor jobs run, but HTCondor is not using the machine for anything.

  Benchmarking
  The machine is running benchmarks to determine the speed on this machine. This activity only occurs in the Unclaimed state. How often the activity occurs is determined by the expression.

- Matched

  Idle
  When Matched, the machine is still Idle to HTCondor.

- Claimed

  Idle
  In this activity, the machine has been claimed, but the schedd that claimed it has yet to activate the claim by requesting a *condor_starter* to be spawned to service a job. The machine returns to this state (usually briefly) when jobs (and therefore *condor_starter*) finish.

  Busy
  Once a *condor_starter* has been started and the claim is active, the machine moves to the Busy activity to signify that it is doing something as far as HTCondor is concerned.

  Suspended
  If the job is suspended by HTCondor, the machine goes into the Suspended activity. The match between the schedd and machine has not been broken (the claim is still valid), but the job is not making any progress and HTCondor is no longer generating a load on the machine.

  Retiring
  When an active claim is about to be preempted for any reason, it enters retirement, while it waits for the current job to finish. The expression determines how long to wait (counting since the

time the job started). Once the job finishes or the retirement time expires, the Preempting state is entered.

- Preempting The Preempting state is used for evicting an HTCondor job from a given machine. When the machine enters the Preempting state, it checks the expression to determine its activity.

  Vacating
  In the Vacating activity, the job is given a chance to exit cleanly. This may include uploading intermediate files. As soon as the job finishes exiting, the machine moves into either the Owner state or the Claimed state, depending on the reason for its preemption.

  Killing
  Killing means that the machine has requested the running job to exit the machine immediately.

- Backfill

  Idle
  The machine is configured to run backfill jobs and is ready to do so, but it has not yet had a chance to spawn a backfill manager (for example, the BOINC client).

  Busy
  The machine is performing a backfill computation.

  Killing
  The machine was running a backfill computation, but it is now killing the job to either return resources to the machine owner, or to make room for a regular HTCondor job.

- Drained

  Idle
  All slots have been drained.

  Retiring
  This slot has been drained. It is waiting for other slots to finish draining.

The following diagram gives the overall view of all machine states and activities and shows the possible transitions from one to another within the HTCondor system. Each transition is labeled with a number on the diagram, and transition numbers referred to in this manual will be **bold**.

Various expressions are used to determine when and if many of these state and activity transitions occur. Other transitions are initiated by parts of the HTCondor protocol (such as when the *condor_negotiator* matches a machine with a schedd). The following section describes the conditions that lead to the various state and activity transitions.

State and Activity Transitions

This section traces through all possible state and activity transitions within a machine and describes the conditions under which each one occurs. Whenever a transition occurs, HTCondor records when the machine entered its new activity and/or new state. These times are often used to write expressions that determine when further transitions occurred. For example, enter the Killing activity if a machine has been in the Vacating activity longer than a specified amount of time.

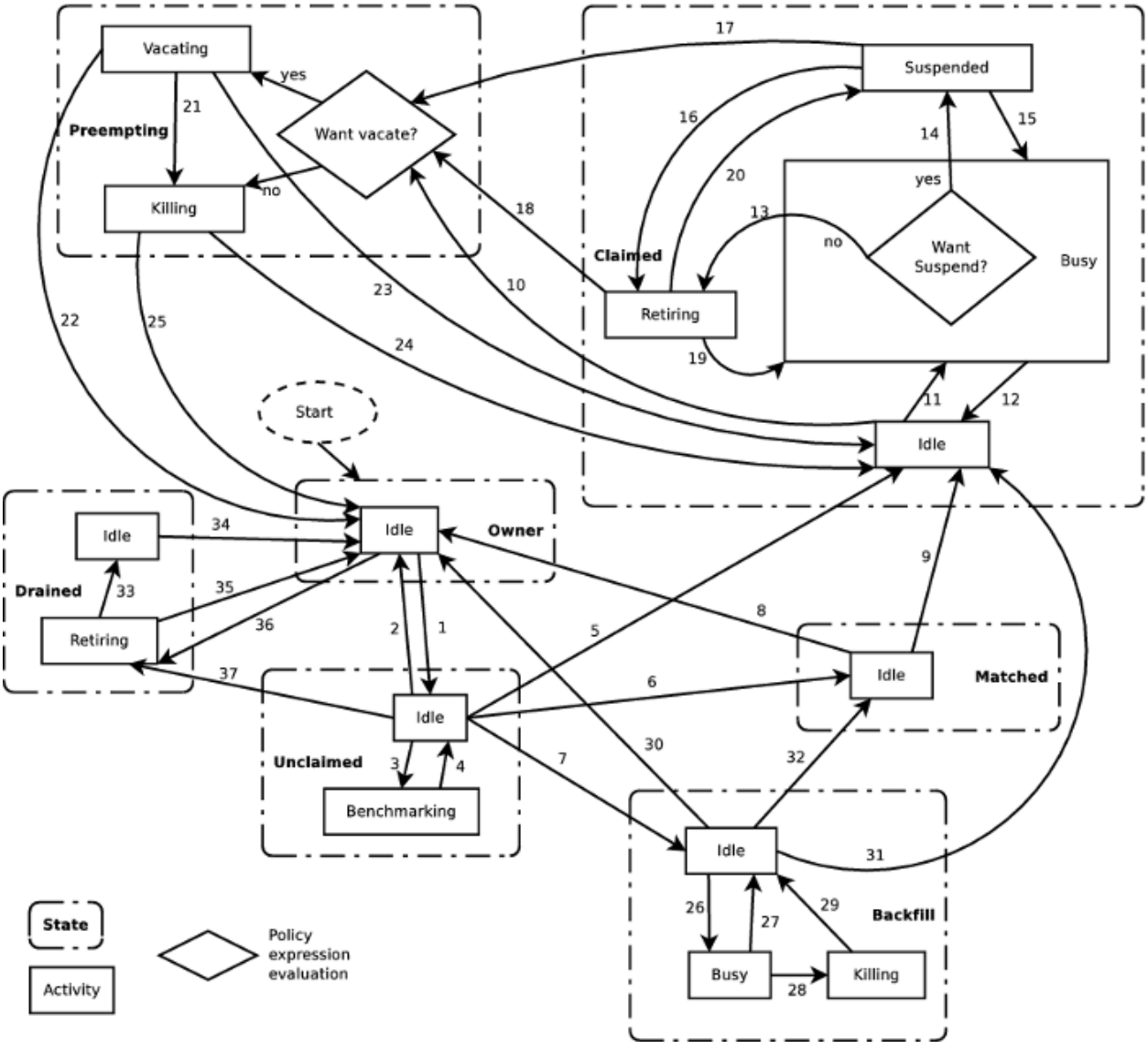


Fig. 5: Machine States and Activities

Owner State

When the startd is first spawned, the machine it represents enters the Owner state. The machine remains in the Owner state while the expression evaluates to TRUE. If the IS_OWNER expression evaluates to FALSE, then the machine transitions to the Unclaimed state. The default value of IS_OWNER is FALSE, which is intended for dedicated resources. But when the POLICY : Desktop configuration template is used, the IS_OWNER expression is optimized for a shared resource

```
START =?= FALSE
```

So, the machine will remain in the Owner state as long as the START expression locally evaluates to FALSE. The *condor_startd Policy Configuration* section provides more detail on the START expression. If the START locally evaluates to TRUE or cannot be locally evaluated (it evaluates to UNDEFINED), transition **1** occurs and the machine enters the Unclaimed state. The IS_OWNER expression is locally evaluated by the machine, and should not reference job ClassAd attributes, which would be UNDEFINED.

The Owner state represents a resource that is in use by its interactive owner (for example, if the keyboard is being used). The Unclaimed state represents a resource that is neither in use by its interactive user, nor the HTCondor system. From HTCondor's point of view, there is little difference between the Owner and Unclaimed states. In both cases, the resource is not currently in use by the HTCondor system. However, if a job matches the resource's START expression, the resource is available to run a job, regardless of if it is in the Owner or Unclaimed state. The only differences between the two states are how the resource shows up in *condor_status* and other reporting tools, and the fact that HTCondor will not run benchmarking on a resource in the Owner state. As long as the IS_OWNER expression is TRUE, the machine is in the Owner State. When the IS_OWNER expression is FALSE, the machine goes into the Unclaimed State.

Here is an example that assumes that the POLICY : Desktop configuration template is in use. If the START expression is

```
START = KeyboardIdle > 15 * $(MINUTE) && Owner == "coltrane"
```

and if KeyboardIdle is 34 seconds, then the machine would remain in the Owner state. Owner is undefined, and anything && FALSE is FALSE.

If, however, the START expression is

```
START = KeyboardIdle > 15 * $(MINUTE) || Owner == "coltrane"
```

and KeyboardIdle is 34 seconds, then the machine leaves the Owner state and becomes Unclaimed. This is because FALSE || UNDEFINED is UNDEFINED. So, while this machine is not available to just anybody, if user coltrane has jobs submitted, the machine is willing to run them. Any other user's jobs have to wait until KeyboardIdle exceeds 15 minutes. However, since coltrane might claim this resource, but has not yet, the machine goes to the Unclaimed state.

While in the Owner state, the startd polls the status of the machine every to see if anything has changed that would lead it to a different state. This minimizes the impact on the Owner while the Owner is using the machine. Frequently waking up, computing load averages, checking the access times on files, computing free swap space take time, and there is nothing time critical that the startd needs to be sure to notice as soon as it happens. If the START expression evaluates to TRUE and five minutes pass before the startd notices, that's a drop in the bucket of high-throughput computing.

The machine can only transition to the Unclaimed state from the Owner state. It does so when the IS_OWNER expression no longer evaluates to TRUE. With the POLICY : Desktop configuration template, that happens when START no longer locally evaluates to FALSE.

Whenever the machine is not actively running a job, it will transition back to the Owner state if IS_OWNER evaluates to TRUE. Once a job is started, the value of IS_OWNER does not matter; the job either runs to completion or is preempted. Therefore, you must configure the preemption policy if you want to transition back to the Owner state from Claimed Busy.

If draining of the machine is initiated while in the Owner state, the slot transitions to Drained/Retiring (transition **36**).

Unclaimed State

If the IS_OWNER expression becomes TRUE, then the machine returns to the Owner state. If the IS_OWNER expression becomes FALSE, then the machine remains in the Unclaimed state. The default value of IS_OWNER is FALSE (never enter Owner state). If the POLICY : Desktop configuration template is used, then the IS_OWNER expression is changed to

```
START =?= FALSE
```

so that while in the Unclaimed state, if the START expression locally evaluates to FALSE, the machine returns to the Owner state by transition **2**.

When in the Unclaimed state, the RUNBENCHMARKS expression is relevant. If RUNBENCHMARKS evaluates to TRUE while the machine is in the Unclaimed state, then the machine will transition from the Idle activity to the Benchmarking activity (transition **3**) and perform benchmarks to determine MIPS and KFLOPS. When the benchmarks complete, the machine returns to the Idle activity (transition **4**).

The startd automatically inserts an attribute, LastBenchmark, whenever it runs benchmarks, so commonly RunBenchmarks is defined in terms of this attribute, for example:

```
RunBenchmarks = (time() - LastBenchmark) >= (4 * $(HOUR))
```

This macro calculates the time since the last benchmark, so when this time exceeds 4 hours, we run the benchmarks again. The startd keeps a weighted average of these benchmarking results to try to get the most accurate numbers possible. This is why it is desirable for the startd to run them more than once in its lifetime.

Note: LastBenchmark is initialized to 0 before benchmarks have ever been run. To have the *condor_startd* run benchmarks as soon as the machine is Unclaimed (if it has not done so already), include a term using LastBenchmark as in the example above.

Note: If RUNBENCHMARKS is defined and set to something other than FALSE, the startd will automatically run one set of benchmarks when it first starts up. To disable benchmarks, both at startup and at any time thereafter, set RUNBENCHMARKS to FALSE or comment it out of the configuration file.

From the Unclaimed state, the machine can go to four other possible states: Owner (transition **2**), Backfill/Idle, Matched, or Claimed/Idle.

Once the *condor_negotiator* matches an Unclaimed machine with a requester at a given schedd, the negotiator sends a command to both parties, notifying them of the match. If the schedd receives that notification and initiates the claiming procedure with the machine before the negotiator's message gets to the machine, the Match state is skipped, and the machine goes directly to the Claimed/Idle state (transition **5**). However, normally the machine will enter the Matched state (transition **6**), even if it is only for a brief period of time.

If the machine has been configured to perform backfill jobs (see the *Setting Up for Special Environments* section), while it is in Unclaimed/Idle it will evaluate the expression. Once START_BACKFILL evaluates to TRUE, the machine will enter the Backfill/Idle state (transition **7**) to begin the process of running backfill jobs.

If draining of the machine is initiated while in the Unclaimed state, the slot transitions to Drained/Retiring (transition **37**).

Matched State

The Matched state is not very interesting to HTCondor. Noteworthy in this state is that the machine lies about its `START` expression while in this state and says that `Requirements` are `False` to prevent being matched again before it has been claimed. Also interesting is that the startd starts a timer to make sure it does not stay in the Matched state too long. The timer is set with the configuration file macro. It is specified in seconds and defaults to 120 (2 minutes). If the schedd that was matched with this machine does not claim it within this period of time, the machine gives up, and goes back into the Owner state via transition **8**. It will probably leave the Owner state right away for the Unclaimed state again and wait for another match.

At any time while the machine is in the Matched state, if the `START` expression locally evaluates to FALSE, the machine enters the Owner state directly (transition **8**).

If the schedd that was matched with the machine claims it before the `MATCH_TIMEOUT` expires, the machine goes into the Claimed/Idle state (transition **9**).

Claimed State

The Claimed state is certainly the most complex state. It has the most possible activities and the most expressions that determine its next activities. In addition, the *condor_vacate* command affects the machine when it is in the Claimed state.

In general, there are two sets of expressions that might take effect, depending on the universe of the job running on the claim: vanilla, and all others. The normal expressions look like the following:

```
WANT_SUSPEND            = True
WANT_VACATE             = $(ActivationTimer) > 10 * $(MINUTE)
SUSPEND                 = $(KeyboardBusy) || $(CPUBusy)
...
```

The vanilla expressions have the string"_VANILLA" appended to their names. For example:

```
WANT_SUSPEND_VANILLA    = True
WANT_VACATE_VANILLA     = True
SUSPEND_VANILLA         = $(KeyboardBusy) || $(CPUBusy)
...
```

Without specific vanilla versions, the normal versions will be used for all jobs, including vanilla jobs. In this manual, the normal expressions are referenced.

While Claimed, the takes effect, and the startd polls the machine much more frequently to evaluate its state.

If the machine owner starts typing on the console again, it is best to notice this as soon as possible to be able to start doing whatever the machine owner wants at that point. For multi-core machines, if any slot is in the Claimed state, the startd polls the machine frequently. If already polling one slot, it does not cost much to evaluate the state of all the slots at the same time.

There are a variety of events that may cause the startd to try to get rid of or temporarily suspend a running job. Activity on the machine's console, load from other jobs, or shutdown of the startd via an administrative command are all possible sources of interference. Another one is the appearance of a higher priority claim to the machine by a different HTCondor user.

Depending on the configuration, the startd may respond quite differently to activity on the machine, such as keyboard activity or demand for the cpu from processes that are not managed by HTCondor. The startd can be configured to

completely ignore such activity or to suspend the job or even to kill it. A standard configuration for a desktop machine might be to go through successive levels of getting the job out of the way. The first and least costly to the job is suspending it. If suspending the job for a short while does not satisfy the machine owner (the owner is still using the machine after a specific period of time), the startd moves on to vacating the job. Vanilla jobs are sent a soft kill signal so that they can gracefully shut down if necessary; the default is SIGTERM. If vacating does not satisfy the machine owner (usually because it is taking too long and the owner wants their machine back now), the final, most drastic stage is reached: killing. Killing is a quick death to the job, using a hard-kill signal that cannot be intercepted by the application. For vanilla jobs that do no special signal handling, vacating and killing are equivalent.

The `WANT_SUSPEND` expression determines if the machine will evaluate the `SUSPEND` expression to consider entering the Suspended activity. The `WANT_VACATE` expression determines what happens when the machine enters the Preempting state. It will go to the Vacating activity or directly to Killing. If one or both of these expressions evaluates to FALSE, the machine will skip that stage of getting rid of the job and proceed directly to the more drastic stages.

When the machine first enters the Claimed state, it goes to the Idle activity. From there, it has two options. It can enter the Preempting state via transition **10** (if a *condor_vacate* arrives, or if the `START` expression locally evaluates to FALSE), or it can enter the Busy activity (transition **11**) if the schedd that has claimed the machine decides to activate the claim and start a job.

From Claimed/Busy, the machine can transition to three other state/activity pairs. The startd evaluates the `WANT_SUSPEND` expression to decide which other expressions to evaluate. If `WANT_SUSPEND` is TRUE, then the startd evaluates the `SUSPEND` expression. If `WANT_SUSPEND` is any value other than TRUE, then the startd will evaluate the `PREEMPT` expression and skip the Suspended activity entirely. By transition, the possible state/activity destinations from Claimed/Busy:

Claimed/Idle
: If the starter that is serving a given job exits (for example because the jobs completes), the machine will go to Claimed/Idle (transition **12**). Claimed/Retiring If `WANT_SUSPEND` is FALSE and the `PREEMPT` expression is `True`, the machine enters the Retiring activity (transition **13**). From there, it waits for a configurable amount of time for the job to finish before moving on to preemption.

  Another reason the machine would go from Claimed/Busy to Claimed/Retiring is if the *condor_negotiator* matched the machine with a "better" match. This better match could either be from the machine's perspective using the startd `RANK` expression, or it could be from the negotiator's perspective due to a job with a higher user priority.

  Another case resulting in a transition to Claimed/Retiring is when the startd is being shut down. The only exception is a "fast" shutdown, which bypasses retirement completely.

Claimed/Suspended
: If both the `WANT_SUSPEND` and `SUSPEND` expressions evaluate to TRUE, the machine suspends the job (transition **14**).

From the Claimed/Suspended state, the following transitions may occur:

Claimed/Busy
: If the `CONTINUE` expression evaluates to TRUE, the machine resumes the job and enters the Claimed/Busy state (transition **15**) or the Claimed/Retiring state (transition **16**), depending on whether the claim has been preempted.

Claimed/Retiring
: If the `PREEMPT` expression is TRUE, the machine will enter the Claimed/Retiring activity (transition **16**).

Preempting
: If the claim is in suspended retirement and the retirement time expires, the job enters the Preempting state (transition **17**). This is only possible if `MaxJobRetirementTime` decreases during the suspension.

For the Claimed/Retiring state, the following transitions may occur:

Preempting
: If the job finishes or the job's run time exceeds the value defined for the job ClassAd attribute

`MaxJobRetirementTime`, the Preempting state is entered (transition **18**). The run time is computed from the time when the job was started by the startd minus any suspension time. When retiring due to *condor_startd* daemon shutdown or restart, it is possible for the administrator to issue a peaceful shutdown command, which causes `MaxJobRetirementTime` to effectively be infinite, avoiding any killing of jobs. It is also possible for the administrator to issue a fast shutdown command, which causes `MaxJobRetirementTime` to be effectively 0.

Claimed/Busy
If the startd was retiring because of a preempting claim only and the preempting claim goes away, the normal Claimed/Busy state is resumed (transition **19**). If instead the retirement is due to owner activity (`PREEMPT`) or the startd is being shut down, no unretirement is possible.

Claimed/Suspended
In exactly the same way that suspension may happen from the Claimed/Busy state, it may also happen during the Claimed/Retiring state (transition **20**). In this case, when the job continues from suspension, it moves back into Claimed/Retiring (transition **16**) instead of Claimed/Busy (transition **15**).

Preempting State

The Preempting state is less complex than the Claimed state. There are two activities. Depending on the value of `WANT_VACATE`, a machine will be in the Vacating activity (if `True`) or the Killing activity (if `False`).

While in the Preempting state (regardless of activity) the machine advertises its `Requirements` expression as `False` to signify that it is not available for further matches, either because it is about to transition to the Owner state, or because it has already been matched with one preempting match, and further preempting matches are disallowed until the machine has been claimed by the new match.

The main function of the Preempting state is to get rid of the *condor_starter* associated with the resource. If the *condor_starter* associated with a given claim exits while the machine is still in the Vacating activity, then the job successfully completed a graceful shutdown. For other jobs, this means the application was given an opportunity to do a graceful shutdown, by intercepting the soft kill signal.

If the machine is in the Vacating activity, it keeps evaluating the `KILL` expression. As soon as this expression evaluates to TRUE, the machine enters the Killing activity (transition **21**). If the Vacating activity lasts for as long as the maximum vacating time, then the machine also enters the Killing activity. The maximum vacating time is determined by the configuration variable . This may be adjusted by the setting of the job ClassAd attribute `JobMaxVacateTime`.

When the starter exits, or if there was no starter running when the machine enters the Preempting state (transition **10**), the other purpose of the Preempting state is completed: notifying the schedd that had claimed this machine that the claim is broken.

At this point, the machine enters either the Owner state by transition **22** (if the job was preempted because the machine owner came back) or the Claimed/Idle state by transition **23** (if the job was preempted because a better match was found).

If the machine enters the Killing activity, (because either `WANT_VACATE` was `False` or the `KILL` expression evaluated to `True`), it attempts to force the *condor_starter* to immediately kill the underlying HTCondor job. Once the machine has begun to hard kill the HTCondor job, the *condor_startd* starts a timer, the length of which is defined by the macro (*condor_startd Configuration File Macros*). This macro is defined in seconds and defaults to 30. If this timer expires and the machine is still in the Killing activity, something has gone seriously wrong with the *condor_starter* and the startd tries to vacate the job immediately by sending SIGKILL to all of the *condor_starter* 's children, and then to the *condor_starter* itself.

Once the *condor_starter* has killed off all the processes associated with the job and exited, and once the schedd that had claimed the machine is notified that the claim is broken, the machine will leave the Preempting/Killing state. If the job was preempted because a better match was found, the machine will enter Claimed/Idle (transition **24**). If the

preemption was caused by the machine owner (the PREEMPT expression evaluated to TRUE, *condor_vacate* was used, etc), the machine will enter the Owner state (transition **25**).

Backfill State

The Backfill state is used whenever the machine is performing low priority background tasks to keep itself busy. For more information about backfill support in HTCondor, see the *Configuring HTCondor for Running Backfill Jobs* section. This state is only used if the machine has been configured to enable backfill computation, if a specific backfill manager has been installed and configured, and if the machine is otherwise idle (not being used interactively or for regular HTCondor computations). If the machine meets all these requirements, and the START_BACKFILL expression evaluates to TRUE, the machine will move from the Unclaimed/Idle state to Backfill/Idle (transition **7**).

Once a machine is in Backfill/Idle, it will immediately attempt to spawn whatever backfill manager it has been configured to use (currently, only the BOINC client is supported as a backfill manager in HTCondor). Once the BOINC client is running, the machine will enter Backfill/Busy (transition **26**) to indicate that it is now performing a backfill computation.

Note: On multi-core machines, the *condor_startd* will only spawn a single instance of the BOINC client, even if multiple slots are available to run backfill jobs. Therefore, only the first machine to enter Backfill/Idle will cause a copy of the BOINC client to start running. If a given slot on a multi-core enters the Backfill state and a BOINC client is already running under this *condor_startd*, the slot will immediately enter Backfill/Busy without waiting to spawn another copy of the BOINC client.

If the BOINC client ever exits on its own (which normally wouldn't happen), the machine will go back to Backfill/Idle (transition **27**) where it will immediately attempt to respawn the BOINC client (and return to Backfill/Busy via transition **26**).

As the BOINC client is running a backfill computation, a number of events can occur that will drive the machine out of the Backfill state. The machine can get matched or claimed for an HTCondor job, interactive users can start using the machine again, the machine might be evicted with *condor_vacate*, or the *condor_startd* might be shutdown. All of these events cause the *condor_startd* to kill the BOINC client and all its descendants, and enter the Backfill/Killing state (transition **28**).

Once the BOINC client and all its children have exited the system, the machine will enter the Backfill/Idle state to indicate that the BOINC client is now gone (transition **29**). As soon as it enters Backfill/Idle after the BOINC client exits, the machine will go into another state, depending on what caused the BOINC client to be killed in the first place.

If the EVICT_BACKFILL expression evaluates to TRUE while a machine is in Backfill/Busy, after the BOINC client is gone, the machine will go back into the Owner/Idle state (transition **30**). The machine will also return to the Owner/Idle state after the BOINC client exits if *condor_vacate* was used, or if the *condor_startd* is being shutdown.

When a machine running backfill jobs is matched with a requester that wants to run an HTCondor job, the machine will either enter the Matched state, or go directly into Claimed/Idle. As with the case of a machine in Unclaimed/Idle (described above), the *condor_negotiator* informs both the *condor_startd* and the *condor_schedd* of the match, and the exact state transitions at the machine depend on what order the various entities initiate communication with each other. If the *condor_schedd* is notified of the match and sends a request to claim the *condor_startd* before the *condor_negotiator* has a chance to notify the *condor_startd*, once the BOINC client exits, the machine will immediately enter Claimed/Idle (transition **31**). Normally, the notification from the *condor_negotiator* will reach the *condor_startd* before the *condor_schedd* attempts to claim it. In this case, once the BOINC client exits, the machine will enter Matched/Idle (transition **32**).

Drained State

The Drained state is used when the machine is being drained, for example by *condor_drain* or by the *condor_defrag* daemon, and the slot has finished running jobs and is no longer willing to run new jobs.

Slots initially enter the Drained/Retiring state. Once all slots have been drained, the slots transition to the Idle activity (transition **33**).

If draining is finalized or canceled, the slot transitions to Owner/Idle (transitions **34** and **35**).

State/Activity Transition Expression Summary

This section is a summary of the information from the previous sections. It serves as a quick reference.

START
When TRUE, the machine is willing to spawn a remote HTCondor job.

RUNBENCHMARKS
While in the Unclaimed state, the machine will run benchmarks whenever TRUE.

MATCH_TIMEOUT
If the machine has been in the Matched state longer than this value, it will transition to the Owner state.

WANT_SUSPEND
If `True`, the machine evaluates the `SUSPEND` expression to see if it should transition to the Suspended activity. If any value other than `True`, the machine will look at the `PREEMPT` expression.

SUSPEND
If `WANT_SUSPEND` is `True`, and the machine is in the Claimed/Busy state, it enters the Suspended activity if `SUSPEND` is `True`.

CONTINUE
If the machine is in the Claimed/Suspended state, it enter the Busy activity if `CONTINUE` is `True`.

PREEMPT
If the machine is either in the Claimed/Suspended activity, or is in the Claimed/Busy activity and `WANT_SUSPEND` is FALSE, the machine enters the Claimed/Retiring state whenever `PREEMPT` is TRUE.

CLAIM_WORKLIFE
This expression specifies the number of seconds after which a claim will stop accepting additional jobs. This configuration macro is fully documented here: *condor_startd Configuration File Macros*.

MachineMaxVacateTime
When the machine enters the Preempting/Vacating state, this expression specifies the maximum time in seconds that the *condor_startd* will wait for the job to finish. The job may adjust the wait time by setting `JobMaxVacateTime`. If the job's setting is less than the machine's, the job's is used. If the job's setting is larger than the machine's, the result depends on whether the job has any excess retirement time. If the job has more retirement time left than the machine's maximum vacate time setting, then retirement time will be converted into vacating time, up to the amount of `JobMaxVacateTime`. Once the vacating time expires, the job is hard-killed. The `KILL` expression may be used to abort the graceful shutdown of the job at any time.

MAXJOBRETIREMENTTIME
If the machine is in the Claimed/Retiring state, jobs which have run for less than the number of seconds specified by this expression will not be hard-killed. The *condor_startd* will wait for the job to finish or to exceed this amount of time, whichever comes sooner. Time spent in suspension does not count against the job. If the job vacating policy grants the job X seconds of vacating time, a preempted job will be soft-killed X seconds before the end of

its retirement time, so that hard-killing of the job will not happen until the end of the retirement time if the job does not finish shutting down before then. The job may provide its own expression for `MaxJobRetirementTime`, but this can only be used to take less than the time granted by the *condor_startd*, never more. For convenience, nice_user jobs are submitted with a default retirement time of 0, so they will never wait in retirement unless the user overrides the default.

The machine enters the Preempting state with the goal of finishing shutting down the job by the end of the retirement time. If the job vacating policy grants the job X seconds of vacating time, the transition to the Preempting state will happen X seconds before the end of the retirement time, so that the hard-killing of the job will not happen until the end of the retirement time, if the job does not finish shutting down before then.

This expression is evaluated in the context of the job ClassAd, so it may refer to attributes of the current job as well as machine attributes.

By default the *condor_negotiator* will not match jobs to a slot with retirement time remaining. This behavior is controlled by `NEGOTIATOR_CONSIDER_EARLY_PREEMPTION` .

WANT_VACATE
This is checked only when the `PREEMPT` expression is `True` and the machine enters the Preempting state. If `WANT_VACATE` is `True`, the machine enters the Vacating activity. If it is `False`, the machine will proceed directly to the Killing activity.

KILL
If the machine is in the Preempting/Vacating state, it enters Preempting/Killing whenever `KILL` is `True`.

KILLING_TIMEOUT
If the machine is in the Preempting/Killing state for longer than `KILLING_TIMEOUT` seconds, the *condor_startd* sends a SIGKILL to the *condor_starter* and all its children to try to kill the job as quickly as possible.

RANK
If this expression evaluates to a higher number for a pending resource request than it does for the current request, the machine may preempt the current request (enters the Preempting/Vacating state). When the preemption is complete, the machine enters the Claimed/Idle state with the new resource request claiming it.

START_BACKFILL
When TRUE, if the machine is otherwise idle, it will enter the Backfill state and spawn a backfill computation (using BOINC).

EVICT_BACKFILL
When TRUE, if the machine is currently running a backfill computation, it will kill the BOINC client and return to the Owner/Idle state.

Examples of Policy Configuration

This section describes various policy configurations, including the default policy.

Default Policy

These settings are the default as shipped with HTCondor. They have been used for many years with no problems. The vanilla expressions are identical to the regular ones. (They are not listed here. If not defined, the standard expressions are used for vanilla jobs as well).

The following are macros to help write the expressions clearly.

StateTimer
Amount of time in seconds in the current state.

ActivityTimer
Amount of time in seconds in the current activity.

ActivationTimer
Amount of time in seconds that the job has been running on this machine.

NonCondorLoadAvg
The difference between the system load and the HTCondor load (the load generated by everything but HTCondor).

BackgroundLoad
Amount of background load permitted on the machine and still start an HTCondor job.

HighLoad
If the `$(NonCondorLoadAvg)` goes over this, the CPU is considered too busy, and eviction of the HTCondor job should start.

StartIdleTime
Amount of time the keyboard must to be idle before HTCondor will start a job.

ContinueIdleTime
Amount of time the keyboard must to be idle before resumption of a suspended job.

MaxSuspendTime
Amount of time a job may be suspended before more drastic measures are taken.

KeyboardBusy
A boolean expression that evaluates to TRUE when the keyboard is being used.

CPUIdle
A boolean expression that evaluates to TRUE when the CPU is idle.

CPUBusy
A boolean expression that evaluates to TRUE when the CPU is busy.

MachineBusy
The CPU or the Keyboard is busy.

CPUIsBusy
A boolean value set to the same value as `CPUBusy`.

CPUBusyTime
The value 0 if `CPUBusy` is False; the time in seconds since `CPUBusy` became True.

These variable definitions exist in the example configuration file in order to help write legible expressions. They are not required, and perhaps will go unused by many configurations.

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (time() - EnteredCurrentState)
ActivityTimer   = (time() - EnteredCurrentActivity)
ActivationTimer = (time() - JobStart)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
```

(continues on next page)

(continued from previous page)

```
CPUBusy                = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)
```

Preemption is disabled as a default. Always desire to start jobs.

```
WANT_SUSPEND          = False
WANT_VACATE           = False
START                 = True
SUSPEND               = False
CONTINUE              = True
PREEMPT               = False
# Kill jobs that take too long leaving gracefully.
MachineMaxVacateTime = 10 * $(MINUTE)
KILL                  = False
```

Test-job Policy Example

This example shows how the default macros can be used to set up a machine for running test jobs from a specific user. Suppose we want the machine to behave normally, except if user coltrane submits a job. In that case, we want that job to start regardless of what is happening on the machine. We do not want the job suspended, vacated or killed. This is reasonable if we know coltrane is submitting very short running programs for testing purposes. The jobs should be executed right away. This works with any machine (or the whole pool, for that matter) by adding the following 5 expressions to the existing configuration:

```
START      = ($(START)) || Owner == "coltrane"
SUSPEND    = ($(SUSPEND)) && Owner != "coltrane"
CONTINUE   = $(CONTINUE)
PREEMPT    = ($(PREEMPT)) && Owner != "coltrane"
KILL       = $(KILL)
```

Notice that there is nothing special in either the `CONTINUE` or `KILL` expressions. If Coltrane's jobs never suspend, they never look at `CONTINUE`. Similarly, if they never preempt, they never look at `KILL`.

Time of Day Policy

HTCondor can be configured to only run jobs at certain times of the day. In general, we discourage configuring a system like this, since there will often be lots of good cycles on machines, even when their owners say "I'm always using my machine during the day." However, if you submit mostly jobs that cannot produce checkpoints, it might be a good idea to only allow the jobs to run when you know the machines will be idle and when they will not be interrupted.

To configure this kind of policy, use the `ClockMin` and `ClockDay` attributes. These are special attributes which are automatically inserted by the *condor_startd* into its ClassAd, so you can always reference them in your policy expressions. `ClockMin` defines the number of minutes that have passed since midnight. For example, 8:00am is 8 hours after midnight, or 8 * 60 minutes, or 480. 5:00pm is 17 hours after midnight, or 17 * 60, or 1020. `ClockDay` defines the day of the week, Sunday = 0, Monday = 1, and so on.

To make the policy expressions easy to read, we recommend using macros to define the time periods when you want jobs to run or not run. For example, assume regular work hours at your site are from 8:00am until 5:00pm, Monday through Friday:

```
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
              (ClockDay > 0 && ClockDay < 6) )
```

(continues on next page)

(continued from previous page)

```
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
               (ClockDay == 0 || ClockDay == 6) )
```

Of course, you can fine-tune these settings by changing the definition of `AfterHours` and `WorkHours` for your site.

To force HTCondor jobs to stay off of your machines during work hours:

```
# Only start jobs after hours.
START = $(AfterHours)

# Consider the machine busy during work hours, or if the keyboard or
# CPU are busy.
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
```

This `MachineBusy` macro is convenient if other than the default `SUSPEND` and `PREEMPT` expressions are used.

Desktop/Non-Desktop Policy

Suppose you have two classes of machines in your pool: desktop machines and dedicated cluster machines. In this case, you might not want keyboard activity to have any effect on the dedicated machines. For example, when you log into these machines to debug some problem, you probably do not want a running job to suddenly be killed. Desktop machines, on the other hand, should do whatever is necessary to remain responsive to the user.

There are many ways to achieve the desired behavior. One way is to make a standard desktop policy and a standard non-desktop policy and to copy the desired one into the local configuration file for each machine. Another way is to define one standard policy (in the global configuration file) with a simple toggle that can be set in the local configuration file. The following example illustrates the latter approach.

For ease of use, an entire policy is included in this example. Some of the expressions are just the usual default settings.

```
# If "IsDesktop" is configured, make it an attribute of the machine ClassAd.
STARTD_ATTRS = IsDesktop

# Only consider starting jobs if:
# 1) the load average is low enough OR the machine is currently
#    running an HTCondor job
# 2) AND the user is not active (if a desktop)
START = ( ($(CPUIdle) || (State != "Unclaimed" && State != "Owner")) \
          && (IsDesktop =!= True || (KeyboardIdle > $(StartIdleTime))) )

# Suspend (instead of vacating/killing) for the following cases:
WANT_SUSPEND = ( $(SmallJob) || $(JustCpu) \
                 || $(IsVanilla) )

# When preempting, vacate (instead of killing) in the following cases:
WANT_VACATE  = ( $(ActivationTimer) > 10 * $(MINUTE) \
                 || $(IsVanilla) )

# Suspend jobs if:
# 1) The CPU has been busy for more than 2 minutes, AND
# 2) the job has been running for more than 90 seconds
# 3) OR suspend if this is a desktop and the user is active
SUSPEND = ( ((CpuBusyTime > 2 * $(MINUTE)) && ($(ActivationTimer) > 90)) \
            || ( IsDesktop =?= True && $(KeyboardBusy) ) )
```

(continues on next page)

(continued from previous page)

```
# Continue jobs if:
# 1) the CPU is idle, AND
# 2) we've been suspended more than 5 minutes AND
# 3) the keyboard has been idle for long enough (if this is a desktop)
CONTINUE = ( $(CPUIdle) && ($(ActivityTimer) > 300) \
             && (IsDesktop =!= True || (KeyboardIdle > $(ContinueIdleTime))) )

# Preempt jobs if:
# 1) The job is suspended and has been suspended longer than we want
# 2) OR, we don't want to suspend this job, but the conditions to
#    suspend jobs have been met (someone is using the machine)
PREEMPT = ( ((Activity == "Suspended") && \
            ($(ActivityTimer) > $(MaxSuspendTime))) \
           || (SUSPEND && (WANT_SUSPEND == False)) )

# Replace 0 in the following expression with whatever amount of
# retirement time you want dedicated machines to provide.  The other part
# of the expression forces the whole expression to 0 on desktop
# machines.
MAXJOBRETIREMENTTIME = (IsDesktop =!= True) * 0

# Kill jobs if they have taken too long to vacate gracefully
MachineMaxVacateTime = 10 * $(MINUTE)
KILL = False
```

With this policy in the global configuration, the local configuration files for desktops can be easily configured with the following line:

```
IsDesktop = True
```

In all other cases, the default policy described above will ignore keyboard activity.

Disabling and Enabling Preemption

Preemption causes a running job to be suspended or killed, such that another job can run. As of HTCondor version 8.1.5, preemption is disabled by the default configuration. Previous versions of HTCondor had configuration that enabled preemption. Upon upgrade, the previous behavior will continue, if the previous configuration files are used. New configuration file examples disable preemption, but contain directions for enabling preemption.

Job Suspension

As new jobs are submitted that receive a higher priority than currently executing jobs, the executing jobs may be preempted. If the preempted jobs are not capable of writing checkpoints, they lose whatever forward progress they have made, and are sent back to the job queue to await starting over again as another machine becomes available. An alternative to this is to use suspension to freeze the job while some other task runs, and then unfreeze it so that it can continue on from where it left off. This does not require any special handling in the job, unlike most strategies that take checkpoints. However, it does require a special configuration of HTCondor. This example implements a policy that allows the job to decide whether it should be evicted or suspended. The jobs announce their choice through the use of the invented job ClassAd attribute `IsSuspendableJob`, that is also utilized in the configuration.

The implementation of this policy utilizes two categories of slots, identified as suspendable or nonsuspendable. A job identifies which category of slot it wishes to run on. This affects two aspects of the policy:

- Of two jobs that might run on a slot, which job is chosen. The four cases that may occur depend on whether the currently running job identifies itself as suspendable or nonsuspendable, and whether the potentially running job identifies itself as suspendable or nonsuspendable.

1. If the currently running job is one that identifies itself as suspendable, and the potentially running job identifies itself as nonsuspendable, the currently running job is suspended, in favor of running the nonsuspendable one. This occurs independent of the user priority of the two jobs.
2. If both the currently running job and the potentially running job identify themselves as suspendable, then the relative priorities of the users and the preemption policy determines whether the new job will replace the existing job.
3. If both the currently running job and the potentially running job identify themselves as nonsuspendable, then the relative priorities of the users and the preemption policy determines whether the new job will replace the existing job.
4. If the currently running job is one that identifies itself as nonsuspendable, and the potentially running job identifies itself as suspendable, the currently running job continues running.

- What happens to a currently running job that is preempted. A job that identifies itself as suspendable will be suspended, which means it is frozen in place, and will later be unfrozen when the preempting job is finished. A job that identifies itself as nonsuspendable is evicted, giving it a chance to write a checkpoint, and then is killed. The job will return to the idle state in the job queue, and it can try to run again in the future.

```
# Lie to HTCondor, to achieve 2 slots for each real slot
NUM_CPUS = $(DETECTED_CORES)*2
# There is no good way to tell HTCondor that the two slots should be treated
# as though they share the same real memory, so lie about how much
# memory we have.
MEMORY = $(DETECTED_MEMORY)*2

# Slots 1 through DETECTED_CORES are nonsuspendable and the rest are
# suspendable
IsSuspendableSlot = SlotID > $(DETECTED_CORES)

# If I am a suspendable slot, my corresponding nonsuspendable slot is
# my SlotID plus $(DETECTED_CORES)
NonSuspendableSlotState = eval(strcat("slot",SlotID-$(DETECTED_CORES),"_State")

# The above expression looks at slotX_State, so we need to add
# State to the list of slot attributes to advertise.
STARTD_SLOT_ATTRS = $(STARTD_SLOT_ATTRS) State

# For convenience, advertise these expressions in the machine ad.
STARTD_ATTRS = $(STARTD_ATTRS) IsSuspendableSlot NonSuspendableSlotState

MyNonSuspendableSlotIsIdle = \
  (NonSuspendableSlotState =!= "Claimed" && NonSuspendableSlotState =!= "Preempting")

# NonSuspendable slots are always willing to start jobs.
# Suspendable slots are only willing to start if the NonSuspendable slot is idle.
START = \
  IsSuspendableSlot!=True && IsSuspendableJob=!=True || \
  IsSuspendableSlot && IsSuspendableJob==True && $(MyNonSuspendableSlotIsIdle)

# Suspend the suspendable slot if the other slot is busy.
SUSPEND = \
  IsSuspendableSlot && $(MyNonSuspendableSlotIsIdle)!=True
```

(continues on next page)

(continued from previous page)

```
WANT_SUSPEND = $(SUSPEND)

CONTINUE = ($(SUSPEND)) != True
```

Note that in this example, the job ClassAd attribute `IsSuspendableJob` has no special meaning to HTCondor. It is an invented name chosen for this example. To take advantage of the policy, a job that wishes to be suspended must submit the job so that this attribute is defined. The following line should be placed in the job's submit description file:

```
+IsSuspendableJob = True
```

Configuration for Interactive Jobs

Policy may be set based on whether a job is an interactive one or not. Each interactive job has the job ClassAd attribute

```
InteractiveJob = True
```

and this may be used to identify interactive jobs, distinguishing them from all other jobs.

As an example, presume that slot 1 prefers interactive jobs. Set the machine's `RANK` to show the preference:

```
RANK = ( (MY.SlotID == 1) && (TARGET.InteractiveJob =?= True) )
```

Or, if slot 1 should be reserved for interactive jobs:

```
START = ( (MY.SlotID == 1) && (TARGET.InteractiveJob =?= True) )
```

Multi-Core Machine Terminology

Machines with more than one CPU or core may be configured to run more than one job at a time. As always, owners of the resources have great flexibility in defining the policy under which multiple jobs may run, suspend, vacate, etc.

Multi-core machines are represented to the HTCondor system as shared resources broken up into individual slots. Each slot can be matched and claimed by users for jobs. Each slot is represented by an individual machine ClassAd. In this way, each multi-core machine will appear to the HTCondor system as a collection of separate slots. As an example, a multi-core machine named `vulture.cs.wisc.edu` would appear to HTCondor as the multiple machines, named `slot1@vulture.cs.wisc.edu`, `slot2@vulture.cs.wisc.edu`, `slot3@vulture.cs.wisc.edu`, and so on.

The way that the *condor_startd* breaks up the shared system resources into the different slots is configurable. All shared system resources, such as RAM, disk space, and swap space, can be divided evenly among all the slots, with each slot assigned one core. Alternatively, slot types are defined by configuration, so that resources can be unevenly divided. Regardless of the scheme used, it is important to remember that the goal is to create a representative slot ClassAd, to be used for matchmaking with jobs.

HTCondor does not directly enforce slot shared resource allocations, and jobs are free to over subscribe to shared resources. Consider an example where two slots are each defined with 50% of available RAM. The resultant ClassAd for each slot will advertise one half the available RAM. Users may submit jobs with RAM requirements that match these slots. However, jobs run on either slot are free to consume more than 50% of available RAM. HTCondor will not directly enforce a RAM utilization limit on either slot. If a shared resource enforcement capability is needed, it is possible to write a policy that will evict a job that over subscribes to shared resources, as described in *condor_startd Policy Configuration*.

Dividing System Resources in Multi-core Machines

Within a machine the shared system resources of cores, RAM, swap space and disk space will be divided for use by the slots. There are two main ways to go about dividing the resources of a multi-core machine:

Evenly divide all resources.
Prior to HTCondor 23.0 the *condor_startd* will automatically divide the machine into multiple slots by default, placing one core in each slot, and evenly dividing all shared resources among the slots. Beginning with HTCondor 23.0 the *condor_startd* will create a single partitionable slot by default.

In HTCondor 23.0 you can use the configuration template `use FEATURE : StaticSlots` to configure a number of static slots. If used without arguments this configuration template will define a number of single core static slots equal to the number of detected cpu cores.

To simply configure static slots in any version, configure to the integer number of slots desired. `NUM_SLOTS` may not be used to make HTCondor advertise more slots than there are cores on the machine. The number of cores is defined by .

Define slot types.
Instead of the default slot configuration, the machine may have definitions of slot types, where each type is provided with a fraction of shared system resources. Given the slot type definition, control how many of each type are reported at any given time with further configuration.

Configuration variables define the slot types, as well as variables that list how much of each system resource goes to each slot type.

Configuration variable , where <N> is an integer (for example, `SLOT_TYPE_1`) defines the slot type. Note that there may be multiple slots of each type. The number of slots created of a given type is configured with `NUM_SLOTS_TYPE_<N>`.

The type can be defined by:

- A simple fraction, such as 1/4
- A simple percentage, such as 25%
- A comma-separated list of attributes, with a percentage, fraction, numerical value, or `auto` for each one.
- A comma-separated list that includes a blanket value that serves as a default for any resources not explicitly specified in the list.

A simple fraction or percentage describes the allocation of the total system resources, including the number of CPUS or cores. A comma separated list allows a fine tuning of the amounts for specific resources.

The number of CPUs and the total amount of RAM in the machine do not change over time. For these attributes, specify either absolute values or percentages of the total available amount (or `auto`). For example, in a machine with 128 Mbytes of RAM, all the following definitions result in the same allocation amount.

```
SLOT_TYPE_1 = mem=64

SLOT_TYPE_1 = mem=1/2

SLOT_TYPE_1 = mem=50%

SLOT_TYPE_1 = mem=auto
```

Amounts of disk space and swap space are dynamic, as they change over time. For these, specify a percentage or fraction of the total value that is allocated to each slot, instead of specifying absolute values. As the total values of these resources change on the machine, each slot will take its fraction of the total and report that as its available amount.

The disk space allocated to each slot is taken from the disk partition containing the slot's or directory. If every slot is in a different partition, then each one may be defined with up to 100% for its disk share. If some slots are in the same partition, then their total is not allowed to exceed 100%.

The four predefined attribute names are case insensitive when defining slot types. The first letter of the attribute name distinguishes between these attributes. The four attributes, with several examples of acceptable names for each:

- Cpus, C, c, cpu
- ram, RAM, MEMORY, memory, Mem, R, r, M, m
- disk, Disk, D, d
- swap, SWAP, S, s, VirtualMemory, V, v

As an example, consider a machine with 4 cores and 256 Mbytes of RAM. Here are valid example slot type definitions. Types 1-3 are all equivalent to each other, as are types 4-6. Note that in a real configuration, all of these slot types would not be used together, because they add up to more than 100% of the various system resources. This configuration example also omits definitions of `NUM_SLOTS_TYPE_<N>`, to define the number of each slot type.

```
SLOT_TYPE_1 = cpus=2, ram=128, swap=25%, disk=1/2

SLOT_TYPE_2 = cpus=1/2, memory=128, virt=25%, disk=50%

SLOT_TYPE_3 = c=1/2, m=50%, v=1/4, disk=1/2

SLOT_TYPE_4 = c=25%, m=64, v=1/4, d=25%

SLOT_TYPE_5 = 25%

SLOT_TYPE_6 = 1/4
```

The default value for each resource share is `auto`. The share may also be explicitly set to `auto`. All slots with the value `auto` for a given type of resource will evenly divide whatever remains, after subtracting out explicitly allocated resources given in other slot definitions. For example, if one slot is defined to use 10% of the memory and the rest define it as `auto` (or leave it undefined), then the rest of the slots will evenly divide 90% of the memory between themselves.

In both of the following examples, the disk share is set to `auto`, number of cores is 1, and everything else is 50%:

```
SLOT_TYPE_1 = cpus=1, ram=1/2, swap=50%

SLOT_TYPE_1 = cpus=1, disk=auto, 50%
```

Note that it is possible to set the configuration variables such that they specify an impossible configuration. If this occurs, the *condor_startd* daemon fails after writing a message to its log attempting to indicate the configuration requirements that it could not implement.

In addition to the standard resources of CPUs, memory, disk, and swap, the administrator may also define custom resources on a localized per-machine basis. In addition to GPUs (see *Configuring GPUs*.) the administrator can define other types of custom resources.

The resource names and quantities of available resources are defined using configuration variables of the form , as shown in this example:

```
MACHINE_RESOURCE_Cogs = 16
MACHINE_RESOURCE_actuator = 8
```

If the configuration uses the optional configuration variable to enable and disable local machine resources, also add the resource names to this variable. For example:

```
if defined MACHINE_RESOURCE_NAMES
  MACHINE_RESOURCE_NAMES = $(MACHINE_RESOURCE_NAMES) Cogs actuator
endif
```

Local machine resource names defined in this way may now be used in conjunction with , using all the same syntax described earlier in this section. The following example demonstrates the definition of static and partitionable slot types with local machine resources:

```
# declare one partitionable slot with half of the Cogs, 6 actuators, and
# 50% of all other resources:
SLOT_TYPE_1 = cogs=50%,actuator=6,50%
SLOT_TYPE_1_PARTITIONABLE = TRUE
NUM_SLOTS_TYPE_1 = 1

# declare two static slots, each with 25% of the Cogs, 1 actuator, and
# 25% of all other resources:
SLOT_TYPE_2 = cogs=25%,actuator=1,25%
SLOT_TYPE_2_PARTITIONABLE = FALSE
NUM_SLOTS_TYPE_2 = 2
```

A job may request these local machine resources using the syntax **request_<name>** , as described in *condor_startd Policy Configuration*. This example shows a portion of a submit description file that requests cogs and an actuator:

```
universe = vanilla

# request two cogs and one actuator:
request_cogs = 2
request_actuator = 1

queue
```

The slot ClassAd will represent each local machine resource with the following attributes:

> `Total<name>`: the total quantity of the resource identified by <name> `Detected<name>`: the quantity detected of the resource identified by <name>; this attribute is currently equivalent to `Total<name>` `TotalSlot<name>`: the quantity of the resource identified by <name> allocated to this slot <name>: the amount of the resource identified by <name> available to be used on this slot

From the example given, the `Cogs` resource would be represented by the ClassAd attributes `TotalCogs`, `DetectedCogs`, `TotalSlotCogs`, and `Cogs`. In the job ClassAd, the amount of the requested machine resource appears in a job ClassAd attribute named `Request<name>`. For this example, the two attributes will be `RequestCogs` and `RequestActuator`.

The number of each type and the definitions for the types themselves cannot be changed with reconfiguration. To change any slot type definitions, use *condor_restart*

```
$ condor_restart -startd
```

for that change to take effect.

Configuration Specific to Multi-core Machines

Each slot within a multi-core machine is treated as an independent machine, each with its own view of its state as represented by the machine ClassAd attribute `State`. The policy expressions for the multi-core machine as a whole are propagated from the *condor_startd* to the slot's machine ClassAd. This policy may consider a slot state(s) in its expressions. This makes some policies easy to set, but it makes other policies difficult or impossible to set.

An easy policy to set configures how many of the slots notice console or tty activity on the multi-core machine as a whole. Slots that are not configured to notice any activity will report `ConsoleIdle` and `KeyboardIdle` times from when the *condor_startd* daemon was started, plus a configurable number of seconds. A multi-core machine with the default policy settings can add the keyboard and console to be noticed by only one slot. Assuming a reasonable load average, only the one slot will suspend or vacate its job when the owner starts typing at their machine again. The rest of the slots could be matched with jobs and continue running them, even while the user was interactively using the machine. If the default policy is used, all slots notice tty and console activity and currently running jobs would suspend.

This example policy is controlled with the following configuration variables.

- `SLOTS_CONNECTED_TO_CONSOLE` , with definition at the *condor_startd Configuration File Macros* section
- `SLOTS_CONNECTED_TO_KEYBOARD` , with definition at the *condor_startd Configuration File Macros* section
- `DISCONNECTED_KEYBOARD_IDLE_BOOST` , with definition at the *condor_startd Configuration File Macros* section

Each slot has its own machine ClassAd. Yet, the policy expressions for the multi-core machine are propagated and inherited from configuration of the *condor_startd*. Therefore, the policy expressions for each slot are the same. This makes the implementation of certain types of policies impossible, because while evaluating the state of one slot within the multi-core machine, the state of other slots are not available. Decisions for one slot cannot be based on what other slots are doing.

Specifically, the evaluation of a slot policy expression works in the following way.

1. The configuration file specifies policy expressions that are shared by all of the slots on the machine.
2. Each slot reads the configuration file and sets up its own machine ClassAd.
3. Each slot is now separate from the others. It has a different ClassAd attribute `State`, a different machine ClassAd, and if there is a job running, a separate job ClassAd. Each slot periodically evaluates the policy expressions, changing its own state as necessary. This occurs independently of the other slots on the machine. So, if the *condor_startd* daemon is evaluating a policy expression on a specific slot, and the policy expression refers to `ProcID`, `Owner`, or any attribute from a job ClassAd, it always refers to the ClassAd of the job running on the specific slot.

To set a different policy for the slots within a machine, incorporate the slot-specific machine ClassAd attribute `SlotID`. A `SUSPEND` policy that is different for each of the two slots will be of the form

```
SUSPEND = ( (SlotID == 1) && (PolicyForSlot1) ) || \
          ( (SlotID == 2) && (PolicyForSlot2) )
```

where (PolicyForSlot1) and (PolicyForSlot2) are the desired expressions for each slot.

Load Average for Multi-core Machines

Most operating systems define the load average for a multi-core machine as the total load on all cores. For example, a 4-core machine with 3 CPU-bound processes running at the same time will have a load of 3.0. In HTCondor, we maintain this view of the total load average and publish it in all resource ClassAds as `TotalLoadAvg`.

HTCondor also provides a per-core load average for multi-core machines. This nicely represents the model that each node on a multi-core machine is a slot, separate from the other nodes. All of the default, single-core policy expressions can be used directly on multi-core machines, without modification, since the `LoadAvg` and `CondorLoadAvg` attributes are the per-slot versions, not the total, multi-core wide versions.

The per-core load average on multi-core machines is an HTCondor invention. No system call exists to ask the operating system for this value. HTCondor already computes the load average generated by HTCondor on each slot. It does this by close monitoring of all processes spawned by any of the HTCondor daemons, even ones that are orphaned and then inherited by *init*. This HTCondor load average per slot is reported as the attribute `CondorLoadAvg` in all resource ClassAds, and the total HTCondor load average for the entire machine is reported as `TotalCondorLoadAvg`. The total, system-wide load average for the entire machine is reported as `TotalLoadAvg`. Basically, HTCondor walks through all the slots and assigns out portions of the total load average to each one. First, HTCondor assigns the known HTCondor load average to each node that is generating load. If there is any load average left in the total system load, it is considered an owner load. Any slots HTCondor believes are in the Owner state, such as ones that have keyboard activity, are the first to get assigned this owner load. HTCondor hands out owner load in increments of at most 1.0, so generally speaking, no slot has a load average above 1.0. If HTCondor runs out of total load average before it runs out of slots, all the remaining machines believe that they have no load average at all. If, instead, HTCondor runs out of slots and it still has owner load remaining, HTCondor starts assigning that load to HTCondor nodes as well, giving individual nodes with a load average higher than 1.0.

Debug Logging in the Multi-Core *condor_startd* Daemon

This section describes how the *condor_startd* daemon handles its debugging messages for multi-core machines. In general, a given log message will either be something that is machine-wide, such as reporting the total system load average, or it will be specific to a given slot. Any log entries specific to a slot have an extra word printed out in the entry with the slot number. So, for example, here's the output about system resources that are being gathered (with `D_FULLDEBUG` and `D_LOAD` turned on) on a 2-core machine with no HTCondor activity, and the keyboard connected to both slots:

```
11/25 18:15 Swap space: 131064
11/25 18:15 number of Kbytes available for (/home/condor/execute): 1345063
11/25 18:15 Looking up RESERVED_DISK parameter
11/25 18:15 Reserving 5120 Kbytes for file system
11/25 18:15 Disk space: 1339943
11/25 18:15 Load avg: 0.340000 0.800000 1.170000
11/25 18:15 Idle Time: user= 0 , console= 4 seconds
11/25 18:15 SystemLoad: 0.340   TotalCondorLoad: 0.000  TotalOwnerLoad: 0.340
11/25 18:15 slot1: Idle time: Keyboard: 0        Console: 4
11/25 18:15 slot1: SystemLoad: 0.340  CondorLoad: 0.000  OwnerLoad: 0.340
11/25 18:15 slot2: Idle time: Keyboard: 0        Console: 4
11/25 18:15 slot2: SystemLoad: 0.000  CondorLoad: 0.000  OwnerLoad: 0.000
11/25 18:15 slot1: State: Owner          Activity: Idle
11/25 18:15 slot2: State: Owner          Activity: Idle
```

If, on the other hand, this machine only had one slot connected to the keyboard and console, and the other slot was running a job, it might look something like this:

```
11/25 18:19 Load avg: 1.250000 0.910000 1.090000
11/25 18:19 Idle Time: user= 0 , console= 0 seconds
11/25 18:19 SystemLoad: 1.250   TotalCondorLoad: 0.996  TotalOwnerLoad: 0.254
11/25 18:19 slot1: Idle time: Keyboard: 0        Console: 0
11/25 18:19 slot1: SystemLoad: 0.254  CondorLoad: 0.000  OwnerLoad: 0.254
11/25 18:19 slot2: Idle time: Keyboard: 1496     Console: 1496
11/25 18:19 slot2: SystemLoad: 0.996  CondorLoad: 0.996  OwnerLoad: 0.000
11/25 18:19 slot1: State: Owner           Activity: Idle
11/25 18:19 slot2: State: Claimed         Activity: Busy
```

Shared system resources are printed without the header, such as total swap space, and slot-specific messages, such as the load average or state of each slot, get the slot number appended.

Configuring GPUs

HTCondor supports incorporating GPU resources and making them available for jobs. First, GPUs must be detected as available resources. Then, machine ClassAd attributes advertise this availability. Both detection and advertisement are accomplished by having this configuration for each execute machine that has GPUs:

```
use feature : GPUs
```

Use of this configuration template invokes the *condor_gpu_discovery* tool to create a custom resource, with a custom resource name of `GPUs`, and it generates the ClassAd attributes needed to advertise the GPUs. *condor_gpu_discovery* is invoked in a mode that discovers and advertises both CUDA and OpenCL GPUs.

This configuration template refers to macro , which can be used to define additional command line arguments for the *condor_gpu_discovery* tool. For example, setting

```
use feature : GPUs
GPU_DISCOVERY_EXTRA = -extra
```

causes the *condor_gpu_discovery* tool to output more attributes that describe the detected GPUs on the machine.

Prior to HTCondor version 9.11 *condor_gpu_discovery* would publish GPU properties using attributes with a name prefix that indicated which GPU the property referred to. Beginning with version 9.11, discovery would default to using nested ClassAds for GPU properties. The administrator can be explicit about which form to use for properties by adding either the `-nested` or `-not-nested` option to .

The format – nested or not – of GPU properties in the slot ad is the same as published by *condor_gpu_discovery*. The use of nested GPU property ads is necessary to do GPU matchmaking and to properly support heterogeneous GPUs. For pools that have execute nodes running older versions of HTCondor, you may want to config `-not-nested` on newer machines for consistency with older machines. However jobs that use the `require_gpus` keyword will never match machines that are configured to use `-not-nested` gpu discovery.

For resources like GPUs that have individual properties, when configuring slots the slot configuration can specify a constraint on those properties for the purpose of choosing which GPUs are assigned to which slots. This serves the same purpose as the `require_gpus` submit keyword, but in this case it controls the slot configuration on startup.

The resource constraint can be specified by following the resource quantity with a colon and then a constraint expression. The constraint expression can refer to resource property attributes like the GPU properties from *condor_gpu_discovery* `-nested` output. If the constraint expression is a string literal, it will be matched automatically against the resource id, otherwise it will be evaluated against each of the resource property ads.

When using resource constraints, it is recommended that you put each resource quantity on a separate line as in the following example, otherwise the constraint expression may be truncated.

```
# Assuming a machine that has two types of GPUs, 2 of which have Capability 8.0
# and the remaining GPUs are less powerful

# declare a partitionable slot that has the 2 powerful GPUs
# and 90% of the other resources:
SLOT_TYPE_1 @=slot
   GPUs = 2 : Capability >= 8.0
   90%
@slot
SLOT_TYPE_1_PARTITIONABLE = TRUE
NUM_SLOTS_TYPE_1 = 1

# declare a small static slot and assign it a specific GPU by id
SLOT_TYPE_2 @=slot
   GPUs = 1 : "GPU-6a96bd13"
   CPUs = 1
          Memory = 10
@slot
SLOT_TYPE_2_PARTITIONABLE = FALSE
NUM_SLOTS_TYPE_2 = 1

# declare two static slots that split up the remaining resources which may or␣
↪may not include GPUs
SLOT_TYPE_3 = auto
SLOT_TYPE_3_PARTITIONABLE = FALSE
NUM_SLOTS_TYPE_3 = 2
```

Configuring STARTD_ATTRS on a per-slot basis

The settings can be configured on a per-slot basis. The *condor_startd* daemon builds the list of items to advertise by combining the lists in this order:

1. STARTD_ATTRS
2. SLOT<N>_STARTD_ATTRS

For example, consider the following configuration:

```
STARTD_ATTRS = favorite_color, favorite_season
SLOT1_STARTD_ATTRS = favorite_movie
SLOT2_STARTD_ATTRS = favorite_song
```

This will result in the *condor_startd* ClassAd for slot1 defining values for `favorite_color`, `favorite_season`, and `favorite_movie`. Slot2 will have values for `favorite_color`, `favorite_season`, and `favorite_song`.

Attributes themselves in the `STARTD_ATTRS` list can also be defined on a per-slot basis. Here is another example:

```
favorite_color = "blue"
favorite_season = "spring"
STARTD_ATTRS = favorite_color, favorite_season
SLOT2_favorite_color = "green"
SLOT3_favorite_season = "summer"
```

For this example, the *condor_startd* ClassAds are

slot1:

```
favorite_color = "blue"
favorite_season = "spring"
```

slot2:

```
favorite_color = "green"
favorite_season = "spring"
```

slot3:

```
favorite_color = "blue"
favorite_season = "summer"
```

Dynamic Provisioning: Partitionable and Dynamic Slots

Dynamic provisioning, also referred to as partitionable or dynamic slots, allows HTCondor to use the resources of a slot in a dynamic way; these slots may be partitioned. This means that more than one job can occupy a single slot at any one time. Slots have a fixed set of resources which include the cores, memory and disk space. By partitioning the slot, the use of these resources becomes more flexible.

Here is an example that demonstrates how resources are divided as more than one job is or can be matched to a single slot. In this example, Slot1 is identified as a partitionable slot and has the following resources:

```
cpu = 10
memory = 10240
disk = BIG
```

Assume that JobA is allocated to this slot. JobA includes the following requirements:

```
cpu = 3
memory = 1024
disk = 10240
```

The portion of the slot that is carved out is now known as a dynamic slot. This dynamic slot has its own machine ClassAd, and its `Name` attribute distinguishes itself as a dynamic slot with incorporating the substring `Slot1_1`.

After allocation, the partitionable Slot1 advertises that it has the following resources still available:

```
cpu = 7
memory = 9216
disk = BIG-10240
```

As each new job is allocated to Slot1, it breaks into `Slot1_1`, `Slot1_2`, `Slot1_3` etc., until the entire set of Slot1's available resources have been consumed by jobs.

To enable dynamic provisioning, define a slot type. and declare at least one slot of that type. Then, identify that slot type as partitionable by setting configuration variable to `True`. The value of `<N>` within the configuration variable name is the same value as in slot type definition configuration variable `SLOT_TYPE_<N>`. For the most common cases the machine should be configured for one slot, managing all the resources on the machine. To do so, set the following configuration variables:

```
NUM_SLOTS = 1
NUM_SLOTS_TYPE_1 = 1
SLOT_TYPE_1 = 100%
SLOT_TYPE_1_PARTITIONABLE = TRUE
```

In a pool using dynamic provisioning, jobs can have extra, and desired, resources specified in the submit description file:

```
request_cpus
request_memory
request_disk (in kilobytes)
```

This example shows a portion of the job submit description file for use when submitting a job to a pool with dynamic provisioning.

```
universe = vanilla

request_cpus = 3
request_memory = 1024
request_disk = 10240

queue
```

Each partitionable slot will have the ClassAd attributes

```
PartitionableSlot = True
SlotType = "Partitionable"
```

Each dynamic slot will have the ClassAd attributes

```
DynamicSlot = True
SlotType = "Dynamic"
```

These attributes may be used in a `START` expression for the purposes of creating detailed policies.

A partitionable slot will always appear as though it is not running a job. If matched jobs consume all its resources, the partitionable slot will eventually show as having no available resources; this will prevent further matching of new jobs. The dynamic slots will show as running jobs. The dynamic slots can be preempted in the same way as all other slots.

Dynamic provisioning provides powerful configuration possibilities, and so should be used with care. Specifically, while preemption occurs for each individual dynamic slot, it cannot occur directly for the partitionable slot, or for groups of dynamic slots. For example, for a large number of jobs requiring 1GB of memory, a pool might be split up into 1GB dynamic slots. In this instance a job requiring 2GB of memory will be starved and unable to run. A partial solution to this problem is provided by defragmentation accomplished by the *condor_defrag* daemon, as discussed in *condor_startd Policy Configuration*.

Another partial solution is a new matchmaking algorithm in the negotiator, referred to as partitionable slot preemption, or pslot preemption. Without pslot preemption, when the negotiator searches for a match for a job, it looks at each slot ClassAd individually. With pslot preemption, the negotiator looks at a partitionable slot and all of its dynamic slots as a group. If the partitionable slot does not have sufficient resources (memory, cpu, and disk) to be matched with the candidate job, then the negotiator looks at all of the related dynamic slots that the candidate job might preempt (following the normal preemption rules described elsewhere). The resources of each dynamic slot are added to those of the partitionable slot, one dynamic slot at a time. Once this partial sum of resources is sufficient to enable a match, the negotiator sends the match information to the *condor_schedd*. When the *condor_schedd* claims the partitionable slot, the dynamic slots are preempted, such that their resources are returned to the partitionable slot for use by the new job.

To enable pslot preemption, the following configuration variable must be set for the *condor_negotiator*:

```
ALLOW_PSLOT_PREEMPTION = True
```

When the negotiator examines the resources of dynamic slots, it sorts the slots by their `CurrentRank` attribute, such that slots with lower values are considered first. The negotiator only examines the cpu, memory and disk resources of the dynamic slots; custom resources are ignored.

Dynamic slots that have retirement time remaining are not considered eligible for preemption, regardless of how configuration variable `NEGOTIATOR_CONSIDER_EARLY_PREEMPTION` is set.

When pslot preemption is enabled, the negotiator will not preempt dynamic slots directly. It will preempt them only as part of a match to a partitionable slot.

When multiple partitionable slots match a candidate job and the various job rank expressions are evaluated to sort the matching slots, the ClassAd of the partitionable slot is used for evaluation. This may cause unexpected results for some expressions, as attributes such as `RemoteOwner` will not be present in a partitionable slot that matches with preemption of some of its dynamic slots.

Defaults for Partitionable Slot Sizes

If a job does not specify the required number of CPUs, amount of memory, or disk space, there are ways for the administrator to set default values for all of these parameters.

First, if any of these attributes are not set in the submit description file, there are three variables in the configuration file that condor_submit will use to fill in default values. These are

- `JOB_DEFAULT_REQUESTCPUS`
- `JOB_DEFAULT_REQUESTMEMORY`
- `JOB_DEFAULT_REQUESTDISK`

The value of these variables can be ClassAd expressions. The default values for these variables, should they not be set are

```
JOB_DEFAULT_REQUESTCPUS = 1
JOB_DEFAULT_REQUESTMEMORY = \
    ifThenElse(MemoryUsage =!= UNDEFINED, MemoryUsage, 1)
JOB_DEFAULT_REQUESTDISK = DiskUsage
```

Note that these default values are chosen such that jobs matched to partitionable slots function similar to static slots. These variables do not apply to **batch** grid universe jobs.

Once the job has been matched, and has made it to the execute machine, the *condor_startd* has the ability to modify these resource requests before using them to size the actual dynamic slots carved out of the partitionable slot. Clearly, for the job to work, the *condor_startd* daemon must create slots with at least as many resources as the job needs. However, it may be valuable to create dynamic slots somewhat bigger than the job's request, as subsequent jobs may be more likely to reuse the newly created slot when the initial job is done using it.

The *condor_startd* configuration variables which control this and their defaults are

```
MODIFY_REQUEST_EXPR_REQUESTCPUS = quantize(RequestCpus, {1})
MODIFY_REQUEST_EXPR_REQUESTMEMORY = quantize(RequestMemory, {128})
MODIFY_REQUEST_EXPR_REQUESTDISK = quantize(RequestDisk, {1024})
```

Enforcing scratch disk usage with on-the-fly, HTCondor managed, per-job scratch filesystems.

Warning: The per job filesystem feature is a work in progress and not currently supported.

On Linux systems, when HTCondor is started as root, it optionally has the ability to create a custom filesystem for the job's scratch directory. This allows HTCondor to prevent the job from using more scratch space than provisioned. This also requires that the disk is managed with the LVM disk management system. Three HTCondor configuration knobs need to be set for this to work, in addition to the above requirements:

```
THINPOOL_VOLUME_GROUP_NAME = vgname
THINPOOL_NAME = htcondor
STARTD_ENFORCE_DISK_LIMITS = true
```

THINPOOL_VOLUME_GROUP_NAME is the name of an existing LVM volume group, with enough disk space to provision all the scratch directories for all running jobs on a worker node. THINPOOL_NAME is the name of the logical volume that the scratch directory filesystems will be created on in the volume group. Finally, STARTD_ENFORCE_DISK_LIMITS is a boolean. When true, if a job fills up the filesystem created for it, the starter will put the job on hold with the out of resources hold code (34). This is the recommended value. If false, should the job fill the filesystem, writes will fail with ENOSPC, and it is up to the job to handle these errors and exit with an appropriate code in every part of the job that writes to the filesystem, including third party libraries.

Note that the ephemeral filesystem created for the job is private to the job, so the contents of that filesystem are not visible outside the process hierarchy. The administrator can use the nsenter command to enter this namespace, if they need to inspect the job's sandbox. As this filesystem will never live through a system reboot, it is mounted with mount options that optimize for performance, not reliability, and may improve performance for I/O heavy jobs.

condor_negotiator-Side Resource Consumption Policies

For partitionable slots, the specification of a consumption policy permits matchmaking at the negotiator. A dynamic slot carved from the partitionable slot acquires the required quantities of resources, leaving the partitionable slot with the remainder. This differs from scheduler matchmaking in that multiple jobs can match with the partitionable slot during a single negotiation cycle.

All specification of the resources available is done by configuration of the partitionable slot. The machine is identified as having a resource consumption policy enabled with

```
CONSUMPTION_POLICY = True
```

A defined slot type that is partitionable may override the machine value with

```
SLOT_TYPE_<N>_CONSUMPTION_POLICY = True
```

A job seeking a match may always request a specific number of cores, amount of memory, and amount of disk space. Availability of these three resources on a machine and within the partitionable slot is always defined and have these default values:

```
CONSUMPTION_CPUS = quantize(target.RequestCpus,{1})
CONSUMPTION_MEMORY = quantize(target.RequestMemory,{128})
CONSUMPTION_DISK = quantize(target.RequestDisk,{1024})
```

Here is an example-driven definition of a consumption policy. Assume a single partitionable slot type on a multi-core machine with 8 cores, and that the resource this policy cares about allocating are the cores. Configuration for the machine includes the definition of the slot type and that it is partitionable.

```
SLOT_TYPE_1 = cpus=8
SLOT_TYPE_1_PARTITIONABLE = True
NUM_SLOTS_TYPE_1 = 1
```

Enable use of the *condor_negotiator*-side resource consumption policy, allocating the job-requested number of cores to the dynamic slot, and use to assess the user usage that will affect user priority by the number of cores allocated. Note that the only attributes valid within the expression are Cpus, Memory, and disk. This must the set to the same value on all machines in the pool.

```
SLOT_TYPE_1_CONSUMPTION_POLICY = True
SLOT_TYPE_1_CONSUMPTION_CPUS = TARGET.RequestCpus
SLOT_WEIGHT = Cpus
```

If custom resources are available within the partitionable slot, they may be used in a consumption policy, by specifying the resource. Using a machine with 4 GPUs as an example custom resource, define the resource and include it in the definition of the partitionable slot:

```
MACHINE_RESOURCE_NAMES = gpus
MACHINE_RESOURCE_gpus = 4
SLOT_TYPE_2 = cpus=8, gpus=4
SLOT_TYPE_2_PARTITIONABLE = True
NUM_SLOTS_TYPE_2 = 1
```

Add the consumption policy to incorporate availability of the GPUs:

```
SLOT_TYPE_2_CONSUMPTION_POLICY = True
SLOT_TYPE_2_CONSUMPTION_gpus = TARGET.RequestGpu
SLOT_WEIGHT = Cpus
```

Defragmenting Dynamic Slots

When partitionable slots are used, some attention must be given to the problem of the starvation of large jobs due to the fragmentation of resources. The problem is that over time the machine resources may become partitioned into slots suitable only for running small jobs. If a sufficient number of these slots do not happen to become idle at the same time on a machine, then a large job will not be able to claim that machine, even if the large job has a better priority than the small jobs.

One way of addressing the partitionable slot fragmentation problem is to periodically drain all jobs from fragmented machines so that they become defragmented. The *condor_defrag* daemon implements a configurable policy for doing that. Its implementation is targeted at machines configured to run whole-machine jobs and at machines that only have partitionable slots. The draining of a machine configured to have both partitionable slots and static slots would have a negative impact on single slot jobs running in static slots.

To use this daemon, DEFRAG must be added to , and the defragmentation policy must be configured. Typically, only one instance of the *condor_defrag* daemon would be run per pool. It is a lightweight daemon that should not require a lot of system resources.

Here is an example configuration that puts the *condor_defrag* daemon to work:

```
DAEMON_LIST = $(DAEMON_LIST) DEFRAG
DEFRAG_INTERVAL = 3600
DEFRAG_DRAINING_MACHINES_PER_HOUR = 1.0
DEFRAG_MAX_WHOLE_MACHINES = 20
DEFRAG_MAX_CONCURRENT_DRAINING = 10
```

This example policy tells *condor_defrag* to initiate draining jobs from 1 machine per hour, but to avoid initiating new draining if there are 20 completely defragmented machines or 10 machines in a draining state. A full description of each configuration variable used by the *condor_defrag* daemon may be found in the *condor_defrag Configuration File Macros* section.

By default, when a machine is drained, existing jobs are gracefully evicted. This means that each job will be allowed to use the remaining time promised to it by `MaxJobRetirementTime`. If the job has not finished when the retirement time runs out, the job will be killed with a soft kill signal, so that it has an opportunity to save a checkpoint (if the job supports this).

By default, no new jobs will be allowed to start while the machine is draining. To reduce unused time on the machine caused by some jobs having longer retirement time than others, the eviction of jobs with shorter retirement time is delayed until the job with the longest retirement time needs to be evicted.

There is a trade off between reduced starvation and throughput. Frequent draining of machines reduces the chance of starvation of large jobs. However, frequent draining reduces total throughput. Some of the machine's resources may go unused during draining, if some jobs finish before others. If jobs that cannot produce checkpoints are killed because they run past the end of their retirement time during draining, this also adds to the cost of draining.

To reduce these costs, you may set the configuration macro . If draining gracefully, the defrag daemon will set the expression for the machine to this value expression. Do not set this to your usual `START` expression; jobs accepted while draining will not be given their `MaxRetirementTime`. Instead, when the last retiring job finishes (either terminates or runs out of retirement time), all other jobs on machine will be evicted with a retirement time of 0. (Those jobs will be given their `MaxVacateTime`, as usual.) The machine's `START` expression will become `FALSE` and stay that way until - as usual - the machine exits the draining state.

We recommend that you allow only interruptible jobs to start on draining machines. Different pools may have different ways of denoting interruptible, but a `MaxJobRetirementTime` of 0 is probably a good sign. You may also want to restrict the interruptible jobs' `MaxVacateTime` to ensure that the machine will complete draining quickly.

To help gauge the costs of draining, the *condor_startd* advertises the accumulated time that was unused due to draining and the time spent by jobs that were killed due to draining. These are advertised respectively in the attributes `TotalMachineDrainingUnclaimedTime` and `TotalMachineDrainingBadput`. The *condor_defrag* daemon averages these values across the pool and advertises the result in its daemon ClassAd in the attributes `AvgDrainingBadput` and `AvgDrainingUnclaimed`. Details of all attributes published by the *condor_defrag* daemon are described in the *Defrag ClassAd Attributes* section.

The following command may be used to view the *condor_defrag* daemon ClassAd:

```
$ condor_status -l -any -constraint 'MyType == "Defrag"'
```

5.7.2 *condor_schedd* Policy Configuration

There are two types of schedd policy: job transforms (which change the ClassAd of a job at submission) and submit requirements (which prevent some jobs from entering the queue). These policies are explained below.

Job Transforms

The *condor_schedd* can transform jobs as they are submitted. Transformations can be used to guarantee the presence of required job attributes, to set defaults for job attributes the user does not supply, or to modify job attributes so that they conform to schedd policy; an example of this might be to automatically set accounting attributes based on the owner of the job while letting the job owner indicate a preference.

There can be multiple job transforms. Each transform can have a Requirements expression to indicate which jobs it should transform and which it should ignore. Transforms without a Requirements expression apply to all jobs. Job transforms are applied in order. The set of transforms and their order are configured using the Configuration variable .

For each entry in this list there must be a corresponding configuration variable that specifies the transform rules. Transforms can use the same syntax as *condor_job_router* transforms; although unlike the *condor_job_router* there is no default transform, and all matching transforms are applied - not just the first one. (See the *The HTCondor Job Router* section for information on the *condor_job_router*.)

Beginning with HTCondor 9.4.0, when a submission is a late materialization job factory, transforms that would match the first factory job will be applied to the Cluster ad at submit time. When job ads are later materialized, attribute values set by the transform will override values set by the job factory for those attributes. Prior to this version transforms were applied to late materialization jobs only after submit time.

The following example shows a set of two transforms: one that automatically assigns an accounting group to jobs based on the submitting user, and one that shows one possible way to transform Vanilla jobs to Docker jobs.

```
JOB_TRANSFORM_NAMES = AssignGroup, SL6ToDocker

JOB_TRANSFORM_AssignGroup @=end
   # map Owner to group using the existing accounting group attribute as requested group
   EVALSET AcctGroup = userMap("Groups",Owner,AcctGroup)
   EVALSET AccountingGroup = join(".",AcctGroup,Owner)
@end

JOB_TRANSFORM_SL6ToDocker @=end
   # match only vanilla jobs that have WantSL6 and do not already have a DockerImage
   REQUIREMENTS JobUniverse==5 && WantSL6 && DockerImage =?= undefined
   SET  WantDocker = true
   SET  DockerImage = "SL6"
   SET  Requirements = TARGET.HasDocker && $(MY.Requirements)
@end
```

The AssignGroup transform above assumes that a mapfile that can map an owner to one or more accounting groups has been configured via , and given the name “Groups”.

The SL6ToDocker transform above is most likely incomplete, as it assumes a custom attribute (`WantSL6`) that your pool may or may not use.

Submit Requirements

The *condor_schedd* may reject job submissions, such that rejected jobs never enter the queue. Rejection may be best for the case in which there are jobs that will never be able to run; for instance, a job specifying an obsolete universe, like standard. Another appropriate example might be to reject all jobs that do not request a minimum amount of memory. Or, it may be appropriate to prevent certain users from using a specific submit host.

Rejection criteria are configured. Configuration variable lists criteria, where each criterion is given a name. The chosen name is a major component of the default error message output if a user attempts to submit a job which fails to meet the requirements. Therefore, choose a descriptive name. For the three example submit requirements described:

```
SUBMIT_REQUIREMENT_NAMES = NotStandardUniverse, MinimalRequestMemory, NotChris
```

The criterion for each submit requirement is then specified in configuration variable , where <Name> matches the chosen name listed in SUBMIT_REQUIREMENT_NAMES. The value is a boolean ClassAd expression. The three example criterion result in these configuration variable definitions:

```
SUBMIT_REQUIREMENT_NotStandardUniverse = JobUniverse != 1
SUBMIT_REQUIREMENT_MinimalRequestMemory = RequestMemory > 512
SUBMIT_REQUIREMENT_NotChris = Owner != "chris"
```

Submit requirements are evaluated in the listed order; the first requirement that evaluates to False causes rejection of the job, terminates further evaluation of other submit requirements, and is the only requirement reported. Each submit requirement is evaluated in the context of the *condor_schedd* ClassAd, which is the MY. name space and the job ClassAd, which is the TARGET. name space. Note that JobUniverse and RequestMemory are both job ClassAd attributes.

Further configuration may associate a rejection reason with a submit requirement with the .

```
SUBMIT_REQUIREMENT_NotStandardUniverse_REASON = "This pool does not accept standard␣
↪universe jobs."
SUBMIT_REQUIREMENT_MinimalRequestMemory_REASON = strcat( "The job only requested ", \
  RequestMemory, " Megabytes.  If that small amount is really enough, please contact ...
↪" )
SUBMIT_REQUIREMENT_NotChris_REASON = "Chris, you may only submit jobs to the␣
↪instructional pool."
```

The value must be a ClassAd expression which evaluates to a string. Thus, double quotes were required to make strings for both SUBMIT_REQUIREMENT_NotStandardUniverse_REASON and SUBMIT_REQUIREMENT_NotChris_REASON. The ClassAd function strcat() produces a string in the definition of SUBMIT_REQUIREMENT_MinimalRequestMemory_REASON.

Rejection reasons are sent back to the submitting program and will typically be immediately presented to the user. If an optional is not defined, a default reason will include the <Name> chosen for the submit requirement. Completing the presentation of the example submit requirements, upon an attempt to submit a standard universe job, *condor_submit* would print

```
Submitting job(s).
ERROR: Failed to commit job submission into the queue.
ERROR: This pool does not accept standard universe jobs.
```

Where there are multiple jobs in a cluster, if any job within the cluster is rejected due to a submit requirement, the entire cluster of jobs will be rejected.

Submit Warnings

Starting in HTCondor 8.7.4, you may instead configure submit warnings. A submit warning is a submit requirement for which is true. A submit warning does not cause the submission to fail; instead, it returns a warning to the user's console (when triggered via *condor_submit*) or writes a message to the user log (always). Submit warnings are intended to allow HTCondor administrators to provide their users with advance warning of new submit requirements. For example, if you want to increase the minimum request memory, you could use the following configuration.

```
SUBMIT_REQUIREMENT_NAMES = OneGig $(SUBMIT_REQUIREMENT_NAMES)
SUBMIT_REQUIREMENT_OneGig = RequestMemory > 1024
SUBMIT_REQUIREMENT_OneGig_REASON = "As of <date>, the minimum requested memory will be␣
↪1024."
SUBMIT_REQUIREMENT_OneGig_IS_WARNING = TRUE
```

When a user runs *condor_submit* to submit a job with `RequestMemory` between 512 and 1024, they will see (something like) the following, assuming that the job meets all the other requirements.

```
Submitting job(s).
WARNING: Committed job submission into the queue with the following warning:
WARNING: As of <date>, the minimum requested memory will be 1024.

1 job(s) submitted to cluster 452.
```

The job will contain (something like) the following:

```
000 (452.000.000) 10/06 13:40:45 Job submitted from host: <128.105.136.53:37317?
↪addrs=128.105.136.53-37317+[fc00--1]-37317&noUDP&sock=19966_e869_5>
    WARNING: Committed job submission into the queue with the following warning: As of
↪<date>, the minimum requested memory will be 1024.
...
```

Marking a submit requirement as a warning does not change when or how it is evaluated, only the result of doing so. In particular, failing a submit warning does not terminate further evaluation of the submit requirements list. Currently, only one (the most recent) problem is reported for each submit attempt. This means users will see (as they previously did) only the first failed requirement; if all requirements passed, they will see the last failed warning, if any.

5.8 Startd Cron and Schedd Cron

5.8.1 Daemon ClassAd Hooks

Overview

The Startd Cron and Schedd Cron *Daemon ClassAd Hooks* mechanism are used to run executables (called jobs) directly from the *condor_startd* and *condor_schedd* daemons. The output from these jobs is incorporated into the machine ClassAd generated by the respective daemon. This mechanism and associated jobs have been identified by various names, including the Startd Cron, dynamic attributes, and a distribution of executables collectively known as Hawkeye.

Pool management tasks can be enhanced by using a daemon's ability to periodically run executables. The executables are expected to generate ClassAd attributes as their output; these ClassAds are then incorporated into the machine ClassAd. Policy expressions can then reference dynamic attributes (created by the ClassAd hook jobs) in the machine ClassAd.

Job output

The output of the job is incorporated into one or more ClassAds when the job exits. When the job outputs the special line:

```
- update:true
```

the output of the job is merged into all proper ClassAds, and an update goes to the *condor_collector* daemon.

As of version 8.3.0, it is possible for a Startd Cron job (but not a Schedd Cron job) to define multiple ClassAds, using the mechanism defined below:

- An output line starting with `'-'` has always indicated end-of-ClassAd. The `'-'` can now be followed by a uniqueness tag to indicate the name of the ad that should be replaced by the new ad. This name is joined to the name of the Startd Cron job to produced a full name for the ad. This allows a single Startd Cron job to return multiple ads by giving each a unique name, and to replace multiple ads by using the same unique name as a previous invocation. The optional uniqueness tag can also be followed by the optional keyword `update:<bool>`, which can be used to override the Startd Cron configuration and suppress or force immediate updates.

  In other words, the syntax is:

  - [*name*] [**update:** *bool*]

- Each ad can contain one of four possible attributes to control what slot ads the ad is merged into when the *condor_startd* sends updates to the collector. These attributes are, in order of highest to lower priority (in other words, if `SlotMergeConstraint` matches, the other attributes are not considered, and so on):

  - **SlotMergeConstraint** *expression*: the current ad is merged into all slot ads for which this expression is true. The expression is evaluated with the slot ad as the TARGET ad.
  - **SlotName|Name** *string*: the current ad is merged into all slots whose `Name` attributes match the value of `SlotName` up to the length of `SlotName`.
  - **SlotTypeId** *integer*: the current ad is merged into all ads that have the same value for their `SlotTypeId` attribute.
  - **SlotId** *integer*: the current ad is merged into all ads that have the same value for their `SlotId` attribute.

For example, if the Startd Cron job returns:

```
Value=1
SlotId=1
-s1
Value=2
SlotId=2
-s2
```

(continues on next page)

(continued from previous page)

```
Value=10
- update:true
```

it will set `Value=10` for all slots except slot1 and slot2. On those slots it will set `Value=1` and `Value=2` respectively. It will also send updates to the collector immediately.

Configuration

Configuration variables related to Daemon ClassAd Hooks are defined in *Configuration File Entries Relating to Daemon ClassAd Hooks: Startd Cron and Schedd Cron*

Here is a complete configuration example. It defines all three of the available types of jobs: ones that use the *condor_startd*, benchmark jobs, and ones that use the *condor_schedd*.

```
#
# Startd Cron Stuff
#
# auxiliary variable to use in identifying locations of files
MODULES = $(ROOT)/modules

STARTD_CRON_CONFIG_VAL = $(RELEASE_DIR)/bin/condor_config_val
STARTD_CRON_MAX_JOB_LOAD = 0.2
STARTD_CRON_JOBLIST =

# Test job
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) test
STARTD_CRON_TEST_MODE = OneShot
STARTD_CRON_TEST_RECONFIG_RERUN = True
STARTD_CRON_TEST_PREFIX = test_
STARTD_CRON_TEST_EXECUTABLE = $(MODULES)/test
STARTD_CRON_TEST_KILL = True
STARTD_CRON_TEST_ARGS = abc 123
STARTD_CRON_TEST_SLOTS = 1
STARTD_CRON_TEST_JOB_LOAD = 0.01

# job 'date'
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) date
STARTD_CRON_DATE_MODE = Periodic
STARTD_CRON_DATE_EXECUTABLE = $(MODULES)/date
STARTD_CRON_DATE_PERIOD = 15s
STARTD_CRON_DATE_JOB_LOAD = 0.01

# Job 'foo'
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) foo
STARTD_CRON_FOO_EXECUTABLE = $(MODULES)/foo
STARTD_CRON_FOO_PREFIX = Foo
STARTD_CRON_FOO_MODE = Periodic
STARTD_CRON_FOO_PERIOD = 10m
STARTD_CRON_FOO_JOB_LOAD = 0.2

#
# Benchmark Stuff
```

(continues on next page)

(continued from previous page)

```
#
BENCHMARKS_JOBLIST = mips kflops

# MIPS benchmark
BENCHMARKS_MIPS_EXECUTABLE = $(LIBEXEC)/condor_mips
BENCHMARKS_MIPS_JOB_LOAD = 1.0

# KFLOPS benchmark
BENCHMARKS_KFLOPS_EXECUTABLE = $(LIBEXEC)/condor_kflops
BENCHMARKS_KFLOPS_JOB_LOAD = 1.0

#
# Schedd Cron Stuff. Unlike the Startd,
# a restart of the Schedd is required for changes to take effect
#
SCHEDD_CRON_CONFIG_VAL = $(RELEASE_DIR)/bin/condor_config_val
SCHEDD_CRON_JOBLIST =

# Test job
SCHEDD_CRON_JOBLIST = $(SCHEDD_CRON_JOBLIST) test
SCHEDD_CRON_TEST_MODE = OneShot
SCHEDD_CRON_TEST_RECONFIG_RERUN = True
SCHEDD_CRON_TEST_PREFIX = test_
SCHEDD_CRON_TEST_EXECUTABLE = $(MODULES)/test
SCHEDD_CRON_TEST_PERIOD = 5m
SCHEDD_CRON_TEST_KILL = True
SCHEDD_CRON_TEST_ARGS = abc 123
```

5.9 Security

5.9.1 Security Overview

Beginning in HTCondor version 9, a main goal is to make all condor installations easier to secure. In previous versions, a default installation typically required additional steps after setup to enable end-to-end security for all users and daemons in the system. Configuring various different types of authentication and security policy could also involve setting quite a number of different configuration parameters and a fairly deep foray into the manual to understand how they all work together.

This overview will explain the high-level concepts involved in securing an HTCondor pool. If possible, we recommend performing a clean installation "from scratch" and then migrating over pieces of your old configuration as needed. Here are some quick links for getting started if you want to jump right in:

Quick Links:

If you are upgrading an existing pool from 8.9.X to 9.0.X, please visit https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=UpgradingFromEightNineToNineZero

If you are installing a new HTCondor pool from scratch, please read about *Downloading and Installing*

General Security Flow

Establishing a secure connection in HTCondor goes through four major steps, which are very briefly enumerated here for reference.

1. Negotiation: In order for a client and server to communicate, they need to agree on which security mechanisms will be used for the connection. This includes whether or not the connection will be authenticated, which types of authentication methods can be used, whether the connection will be encrypted, and which different types of encryption algorithms can be used. The client sends its capabilities, preferences, and requirements; the server compares those against its own, decides what to do, and tells the client; if a connection is possible, they both then work to enact it. We call the decisions the server makes during negotiation the "security policy" for that connection; see *Security Negotiation* for details on policy configuration.
2. Authentication/Mapping: If the server decides to authenticate (and we strongly recommend that it almost always either do so or reject the connection), the methods allowed are tried in the order decided by the server until one of them succeeds. After a successful authentication, the server decides the canonical name of the user based on the credentials used by the client. For SSL, this involves mapping the DN to a user@domain.name format. For most other methods the result is already in user@domain.name format. For details on different types of supported authentication methods, please see *Authentication*.
3. Encryption and Integrity: If the server decided that encryption would be used, both sides now enable encryption and integrity checks using the method preferred by the server. AES is now the preferred method and enabled by default. The overhead of doing the encryption and integrity checks is minimal so we have decided to simplify configuration by requiring changes to disable it rather than enable it. For details on different types of supported authentication methods, see *Encryption*.
4. Authorization: The canonical user is now checked to see if they are allowed to send the command to the server that they wish to send. Commands are "registered" at different authorization levels, and there is an ALLOW/DENY list for each level. If the canonical user is authorized, HTCondor performs the requested action. If authorization fails, the permission is DENIED and the network connection is closed. For list of authorization levels and more information on configuring ALLOW and DENY lists, please see *Authorization*.

Highlights of New Features In Version 9.0.0

Introducing: IDTOKENS

In 9.0.0, we have introduced a new authentication mechanism called `IDTOKENS`. These tokens are easy for the administrator to issue, and in many cases users can also acquire their own tokens on a machine used to submit jobs (running the *condor_schedd*). An `IDTOKEN` is a relatively lightweight credential that can be used to prove an identity. The contents of the token are actually a JWT (https://jwt.io/) that is signed by a "Token Signing Key" that establishes the trustworthiness of the token. Typically, this signing key is something accessible only to HTCondor (and owned by the "root" user of the system) and not users, and by default lives in /etc/condor/passwords.d/POOL. To make configuration easier, this signing key is generated automatically by HTCondor if it does not exist on the machine that runs the Central Manager, or the *condor_collector* daemon in particular. So after installing the central manager and starting it up for the first time, you should as the administrator be all set to start issuing tokens. That said, you will need to copy the signing key to all other machines in your pool that you want to be able to receive and validate the `IDTOKEN` credentials that you issue.

Documentation for the command line tools used for creating and managing `IDTOKENS` is available in the *Token Authentication* section.

Introducing: AES

In version 9.0.0 we have also added support for AES, a widely-used encryption method that has hardware support in most modern CPUS. Because the overhead of encryption is so much lower, we have turned it on by default. We use AES in such a way (called AESGCM mode) that it provides integrity checks (checksums) on transmitted data, and this method is now on by default and is the preferred method to be used if both sides support it.

Types of Network Connections

We generally consider user-to-daemon and daemon-to-daemon connections distinctly. User-to-daemon connections almost always issue `READ` or `WRITE` level commands, and the vast majority of those connections are to the schedd or the collector; many of those connections will be between processes on the same machine. Conversely, daemon-to-daemon connections are typically between two different machines, and use commands registered at all levels.

User-to-Daemon Connections (User Authentication)

In order for users to submit jobs to the HTCondor system, they will need to authenticate to the *condor_schedd* daemon. They also need to authenticate to the SchedD to modify, remove, hold, or release jobs. When users are interacting with the *condor_schedd*, they issue commands that need to be authorized at either the "READ" or "WRITE" level. (Unless the user is an administrator, in which case they might also issue "ADMINISTRATOR"-level commands).

Authenticating using FS

On a Linux system this is typically done by logging into the machine that is running the *condor_schedd* daemon and authentication using a method called `FS` (on Linux see Windows note below this paragraph). `FS` stands for "File System" and the method works by having the user create a file in /tmp that the *condor_schedd* can then examine to determine who the owner is. Because this operates in /tmp, this only works for connections to daemons on the same machine. `FS` is enabled by default so the administrator does not need to do anything to allow users to interact with the job queue this way. (There are other methods, mentioned below, that can work over a network connection.)

[Windows note: HTCondor on Windows does not use `FS`, but rather a method specific to Windows called NTSSPI. See the section on *Authentication* for more more info.]

If it is necessary to do a "remote submit" – that is, run *condor_submit* on a different machine than is running the *condor_schedd* – then the administrator will need to configure another method. `FS_REMOTE` works similarly to `FS` but uses a shared directory other than /tmp. Mechanisms such as `KERBEROS`, `SSL`, and `MUNGE` can also be configured. However, with the addition of `IDTOKENS` in 9.0.0, it is easy to configure and deploy this mechanism and we would suggest you do so unless you have a specific need to use one of the alternatives.

Authenticating using IDTOKENS

If a user is able to log in to the machine running the *condor_schedd*, and the SchedD has been set up with the Token Signing Key (see above for how that is created and deployed) then the user can simply run *condor_token_fetch* and retrieve their own token. This token can then be (securely) moved to another machine and used to interact with the job queue, including submission, edits, hold, release, and removing the job.

If the user cannot log in to the machine running the *condor_schedd*, they should ask their administrator to create tokens for them using the *condor_token_create* command line tool. Once again, more info can be found in the *Token Authentication* section.

Daemon-to-Daemon Connections (Daemon Authentication)

HTCondor daemons need to trust each other to pass information security from one to the other. This information may contain important attributes about a job to run, such as which executable to run, the arguments, and which user to run the job as. Obviously, being able to tamper those could allow an impersonator to perform all sorts of nefarious tasks.

For daemons that run on the same machine, for example a *condor_master*, *condor_schedd*, and the *condor_shadow* daemons launched by the *condor_schedd*, this authentication is performed using a secret that is shared with each condor daemon when it is launched. These are called "family sessions", since the processes sharing the secret are all part of the same unix process family. This allows the HTCondor daemons to contact one another locally without having to use another type of authentication. So essentially, when we are discussing daemon-to-daemon communication, we are talking about HTCondor daemons on two different physical machines. In those cases, they need to establish trust using some mechanism that works over a network. The `FS` mechanism used for user job submission typically doesn't work here because it relies on sharing a directory between the two daemons, typically /tmp. However, `IDTOKENS` are able to work here as long as the server has a copy of the Signing Key that was used to issue the token that the client is using. The daemon will authenticate as `condor@$(TRUST_DOMAIN)` where the trust domain is the string set by the token issuer, and is usually equal to the `$(UID_DOMAIN)` setting on the central manager. (Note that setting has other consequences.)

Once HTCondor has determined the authenticate principal, it checks the authorization lists as mentioned above in *General Security Flow*. For daemon-to-daemon authorization, there are a few lists that may be consulted.

If the condor daemon receiving the connection is the *condor_collector*, it first checks to see if there are specific authorization lists for daemons advertising to the collector (i.e. joining the pool). If the incoming command is advertising a submit node (i.e. a *condor_schedd* daemon), it will check . If the incoming command is for an execute node (a *condor_startd* daemon), it will check `ALLOW_ADVERTISE_STARTD`. And if the incoming command is for a *condor_master* (which runs on all HTCondor nodes) it will check . If the list it checks is undefined, it will then check instead.

If the condor daemon receiving the connection is not a *condor_collector*, the is the only list that is looked at.

It is notable that many daemon-to-daemon connections have been optimized to not need to authenticate using one of the standard methods. Similar to the "family" sessions that work internally on one machine, there are sessions called "match" sessions that can be used internally within one POOL of machines. Here, trust is established by the negotiator when matching a job to a resource – the Negotiator takes a secret generated by the *condor_startd* and securely passes it to the *condor_schedd* when a match is made. The submit and execute machines can now use this secret to establish a secure channel. Because of this, you do not necessarily need to have authentication from one to the other configured; it is enough to have secure channels from the SchedD to the Collector and from the StartD to the collector. Likewise, a Negotiator can establish trust with a SchedD in the same way: the SchedD trusts the Collector to tell only trustworthy Negotiators its secret. However, some features such as *condor_ssh_to_job* and *condor_tail* will not work unless the access point can authenticate directly to the execute point, which is why we mentioned needing to distribute the signing key earlier – if the server does not have the signing key, it cannot directly validate the incoming `IDTOKEN` used for authentication.

5.9.2 Security Terms

Security in HTCondor is a broad issue, with many aspects to consider. Because HTCondor's main purpose is to allow users to run arbitrary code on large numbers of computers, it is important to try to limit who can access an HTCondor pool and what privileges they have when using the pool. This section covers these topics.

There is a distinction between the kinds of resource attacks HTCondor can defeat, and the kinds of attacks HTCondor cannot defeat. HTCondor cannot prevent security breaches of users that can elevate their privilege to the root or administrator account. HTCondor does not run user jobs in sandboxes (possibly excepting Docker or Singularity jobs) so HTCondor cannot defeat all malicious actions by user jobs. An example of a malicious job is one that launches a distributed denial of service attack. HTCondor assumes that users are trustworthy. HTCondor can prevent unauthorized access to the HTCondor pool, to help ensure that only trusted users have access to the pool. In addition, HTCondor

provides encryption and integrity checking, to ensure that network transmissions are not examined or tampered with while in transit.

Broadly speaking, the aspects of security in HTCondor may be categorized and described:

Users
: Authorization or capability in an operating system is based on a process owner. Both those that submit jobs and HTCondor daemons become process owners. The HTCondor system prefers that HTCondor daemons are run as the user root, while other common operations are owned by a user of HTCondor. Operations that do not belong to either root or an HTCondor user are often owned by the condor user. See *User Accounts in HTCondor on Unix Platforms* for more detail.

Authentication
: Proper identification of a user is accomplished by the process of authentication. It attempts to distinguish between real users and impostors. By default, HTCondor's authentication uses the user id (UID) to determine identity, but HTCondor can choose among a variety of authentication mechanisms, including the stronger authentication methods Kerberos and SSL.

Authorization
: Authorization specifies who is allowed to do what. Some users are allowed to submit jobs, while other users are allowed administrative privileges over HTCondor itself. HTCondor provides authorization on either a per-user or on a per-machine basis.

Privacy
: HTCondor may encrypt data sent across the network, which prevents others from viewing the data. With persistence and sufficient computing power, decryption is possible. HTCondor can encrypt the data sent for internal communication, as well as user data, such as files and executables. Encryption operates on network transmissions: unencrypted data is stored on disk by default. However, see the setting for how to encrypt job data on the disk of an execute node.

Integrity
: The man-in-the-middle attack tampers with data without the awareness of either side of the communication. HTCondor's integrity check sends additional cryptographic data to verify that network data transmissions have not been tampered with. Note that the integrity information is only for network transmissions: data stored on disk does not have this integrity information. Also note that integrity checks are not performed upon job data files that are transferred by HTCondor via the File Transfer Mechanism described in the *Submitting a Job* section.

5.9.3 Quick Configuration of Security

Note: This method of configuring security is experimental. Many tools and daemons that send administrative commands between machines (e.g. *condor_off*, *condor_drain*, or *condor_defrag*) won't work without further setup. We plan to remove this limitation in future releases.

While pool administrators with complex configurations or application developers may need to understand the full security model described in this chapter, HTCondor strives to make it easy to enable reasonable security settings for new pools.

When installing a new pool, assuming you are on a trusted network and there are no unprivileged users logged in to the submit hosts:

1. Start HTCondor on your central manager host (containing the *condor_collector* daemon) first. For a fresh install, this will automatically generate a random key in the file specified by `SEC_TOKEN_POOL_SIGNING_KEY_FILE` (defaulting to `/etc/condor/passwords.d/POOL` on Linux and `$(RELEASE_DIR)\tokens.sk\POOL` on Windows).
2. Install an auto-approval rule on the central manager using `condor_token_request_auto_approve`. This automatically approves any daemons starting on a specified network for a fixed period of time. For example, to

auto-authorize any daemon on the network `192.168.0.0/24` for the next hour (3600 seconds), run the following command from the central manager:

```
$ condor_token_request_auto_approve -netblock 192.168.0.0/24 -lifetime 3600
```

3. Within the auto-approval rule's lifetime, start the submit and execute hosts inside the appropriate network. The token requests for the corresponding daemons (the *condor_master*, *condor_startd*, and *condor_schedd*) will be automatically approved and installed into `/etc/condor/tokens.d/`; this will authorize the daemon to advertise to the collector. By default, auto-generated tokens do not have an expiration.

This quick-configuration requires no configuration changes beyond the default settings. More complex cases, such as those where the network is not trusted, are covered in the *Token Authentication* section.

5.9.4 HTCondor's Security Model

At the heart of HTCondor's security model is the notion that communications are subject to various security checks. A request from one HTCondor daemon to another may require authentication to prevent subversion of the system. A request from a user of HTCondor may need to be denied due to the confidential nature of the request. The security model handles these example situations and many more.

Requests to HTCondor are categorized into groups of access levels, based on the type of operation requested. The user of a specific request must be authorized at the required access level. For example, executing the *condor_status* command requires the `READ` access level. Actions that accomplish management tasks, such as shutting down or restarting of a daemon require an `ADMINISTRATOR` access level. See the *Authorization* section for a full list of HTCondor's access levels and their meanings.

There are two sides to any communication or command invocation in HTCondor. One side is identified as the client, and the other side is identified as the daemon. The client is the party that initiates the command, and the daemon is the party that processes the command and responds. In some cases it is easy to distinguish the client from the daemon, while in other cases it is not as easy. HTCondor tools such as *condor_submit* and *condor_config_val* are clients. They send commands to daemons and act as clients in all their communications. For example, the *condor_submit* command communicates with the *condor_schedd*. Behind the scenes, HTCondor daemons also communicate with each other; in this case the daemon initiating the command plays the role of the client. For instance, the *condor_negotiator* daemon acts as a client when contacting the *condor_schedd* daemon to initiate matchmaking. Once a match has been found, the *condor_schedd* daemon acts as a client and contacts the *condor_startd* daemon.

HTCondor's security model is implemented using configuration. Commands in HTCondor are executed over TCP/IP network connections. While network communication enables HTCondor to manage resources that are distributed across an organization (or beyond), it also brings in security challenges. HTCondor must have ways of ensuring that communications are being sent by trustworthy users and not tampered with in transit. These issues can be addressed with HTCondor's authentication, encryption, and integrity features.

Access Level Descriptions

Authorization is granted based on specified access levels. This list describes each access level, and provides examples of their usage. The levels implement a partial hierarchy; a higher level often implies a `READ` or both a `WRITE` and a `READ` level of access as described.

READ
: This access level can obtain or read information about HTCondor. Examples that require only `READ` access are viewing the status of the pool with *condor_status*, checking a job queue with *condor_q*, or viewing user priorities with *condor_userprio*. `READ` access does not allow any changes, and it does not allow job submission.

WRITE
: This access level is required to send (write) information to HTCondor. Examples that require WRITE access are job submission with *condor_submit* and advertising a machine so it appears in the pool (this is usually done automatically by the *condor_startd* daemon). The WRITE level of access implies READ access.

ADMINISTRATOR
: This access level has additional HTCondor administrator rights to the pool. It includes the ability to change user priorities with the command *condor_userprio*, as well as the ability to turn HTCondor on and off (as with the commands *condor_on* and *condor_off*). The *condor_fetchlog* tool also requires an ADMINISTRATOR access level. The ADMINISTRATOR level of access implies both READ and WRITE access.

CONFIG
: This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, this level of access can change any configuration parameters of an HTCondor pool, except those specified in the condor_config.root configuration file. The CONFIG level of access implies READ access.

DAEMON
: This access level is used for commands that are internal to the operation of HTCondor. An example of this internal operation is when the *condor_startd* daemon sends its ClassAd updates to the *condor_collector* daemon (which may be more specifically controlled by the ADVERTISE_STARTD access level). Authorization at this access level should only be given to the user account under which the HTCondor daemons run. The DAEMON level of access implies both READ and WRITE access.

NEGOTIATOR
: This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run. The NEGOTIATOR level of access implies READ access.

ADVERTISE_MASTER
: This access level is used specifically for commands used to advertise a *condor_master* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level. The ADVERTISE_MASTER level of access implies READ access.

ADVERTISE_STARTD
: This access level is used specifically for commands used to advertise a *condor_startd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level. The ADVERTISE_STARTD level of access implies READ access.

ADVERTISE_SCHEDD
: This access level is used specifically for commands used to advertise a *condor_schedd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level. The ADVERTISE_SCHEDD level of access implies READ access.

CLIENT
: This access level is different from all the others. Whereas all of the other access levels refer to the security policy for accepting connections from others, the CLIENT access level applies when an HTCondor daemon or tool is connecting to some other HTCondor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

The following is a list of registered commands that daemons will accept. The list is ordered by daemon. For each daemon, the commands are grouped by the access level required for a daemon to accept the command from a given machine.

ALL DAEMONS:

WRITE
: The command sent as a result of *condor_reconfig* to reconfigure a daemon.

STARTD:

WRITE
All commands that relate to a *condor_schedd* daemon claiming a machine, starting jobs there, or stopping those jobs.

READ
The command that *condor_preen* sends to request the current state of the *condor_startd* daemon.

NEGOTIATOR
The command that the *condor_negotiator* daemon sends to match a machine's *condor_startd* daemon with a given *condor_schedd* daemon.

NEGOTIATOR:

WRITE
The command that initiates a new negotiation cycle. It is sent by the *condor_schedd* when new jobs are submitted or a *condor_reschedule* command is issued.

READ
The command that can retrieve the current state of user priorities in the pool, sent by the *condor_userprio* command.

ADMINISTRATOR
The command that can set the current values of user priorities, sent as a result of the *condor_userprio* command.

COLLECTOR:

ADVERTISE_MASTER
Commands that update the *condor_collector* daemon with new *condor_master* ClassAds.

ADVERTISE_SCHEDD
Commands that update the *condor_collector* daemon with new *condor_schedd* ClassAds.

ADVERTISE_STARTD
Commands that update the *condor_collector* daemon with new *condor_startd* ClassAds.

DAEMON
All other commands that update the *condor_collector* daemon with new ClassAds. Note that the specific access levels such as `ADVERTISE_STARTD` default to the `DAEMON` settings, which in turn defaults to `WRITE`.

READ
All commands that query the *condor_collector* daemon for ClassAds.

SCHEDD:

NEGOTIATOR
The command that the *condor_negotiator* sends to begin negotiating with this *condor_schedd* to match its jobs with available *condor_startds*.

WRITE
The command which *condor_reschedule* sends to the *condor_schedd* to get it to update the *condor_collector* with a current ClassAd and begin a negotiation cycle.

The commands which write information into the job queue (such as *condor_submit* and *condor_hold*). Note that for most commands which attempt to write to the job queue, HTCondor will perform an additional user-level authentication step. This additional user-level authentication prevents, for example, an ordinary user from removing a different user's jobs.

READ
The command from any tool to view the status of the job queue.

The commands that a *condor_startd* sends to the *condor_schedd* when the *condor_schedd* daemon's claim is being preempted and also when the lease on the claim is renewed. These operations only require `READ` access, rather than `DAEMON` in order to limit the level of trust that the *condor_schedd* must have for the *condor_startd*. Success of these commands is only possible if the *condor_startd* knows the secret claim id, so effectively, authorization for these commands is more specific than HTCondor's general security model implies. The *condor_schedd* automatically grants the *condor_startd* `READ` access for the duration of the claim. Therefore, if one desires to only authorize specific execute machines to run jobs, one must either limit which machines are allowed to advertise themselves to the pool (most common) or configure the *condor_schedd* 's setting to only allow connections from the *condor_schedd* to the trusted execute machines.

MASTER: All commands are registered with `ADMINISTRATOR` access:

restart
: Master restarts itself (and all its children)

off
: Master shuts down all its children

off -master
: Master shuts down all its children and exits

on
: Master spawns all the daemons it is configured to spawn

5.9.5 Security Negotiation

Because of the wide range of environments and security demands necessary, HTCondor must be flexible. Configuration provides this flexibility. The process by which HTCondor determines the security settings that will be used when a connection is established is called security negotiation. Security negotiation's primary purpose is to determine which of the features of authentication, encryption, and integrity checking will be enabled for a connection. In addition, since HTCondor supports multiple technologies for authentication and encryption, security negotiation also determines which technology is chosen for the connection.

Security negotiation is a completely separate process from matchmaking, and should not be confused with any specific function of the *condor_negotiator* daemon. Security negotiation occurs when one HTCondor daemon or tool initiates communication with another HTCondor daemon, to determine the security settings by which the communication will be ruled. The *condor_negotiator* daemon does negotiation, whereby queued jobs and available machines within a pool go through the process of matchmaking (deciding out which machines will run which jobs).

Configuration

The configuration macro names that determine what features will be used during client-daemon communication follow the pattern:

```
SEC_<context>_<feature>
```

The <feature> portion of the macro name determines which security feature's policy is being set. <feature> may be any one of

```
AUTHENTICATION
ENCRYPTION
INTEGRITY
NEGOTIATION
```

The <context> component of the security policy macros can be used to craft a fine-grained security policy based on the type of communication taking place. <context> may be any one of

```
CLIENT
READ
WRITE
ADMINISTRATOR
CONFIG
DAEMON
NEGOTIATOR
ADVERTISE_MASTER
ADVERTISE_STARTD
ADVERTISE_SCHEDD
DEFAULT
```

Any of these constructed configuration macros may be set to any of the following values:

```
REQUIRED
PREFERRED
OPTIONAL
NEVER
```

Security negotiation resolves various client-daemon combinations of desired security features in order to set a policy.

As an example, consider Frida the scientist. Frida wants to avoid authentication when possible. She sets

```
SEC_DEFAULT_AUTHENTICATION = OPTIONAL
```

The machine running the *condor_schedd* to which Frida will remotely submit jobs, however, is operated by a security-conscious system administrator who dutifully sets:

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

When Frida submits her jobs, HTCondor's security negotiation determines that authentication will be used, and allows the command to continue. This example illustrates the point that the most restrictive security policy sets the levels of security enforced. There is actually more to the understanding of this scenario. Some HTCondor commands, such as the use of *condor_submit* to submit jobs always require authentication of the submitter, no matter what the policy says. This is because the identity of the submitter needs to be known in order to carry out the operation. Others commands, such as *condor_q*, do not always require authentication, so in the above example, the server's policy would force Frida's *condor_q* queries to be authenticated, whereas a different policy could allow *condor_q* to happen without any authentication.

Whether or not security negotiation occurs depends on the setting at both the client and daemon side of the configuration variable(s) defined by `SEC_*_NEGOTIATION`. is a variable representing the entire set of configuration variables for `NEGOTIATION`. For the client side setting, the only definitions that make sense are `REQUIRED` and `NEVER`. For the daemon side setting, the `PREFERRED` value makes no sense. Table 3.2 shows how security negotiation resolves various client-daemon combinations of security negotiation policy settings. Within the table, Yes means the security negotiation will take place. No means it will not. Fail means that the policy settings are incompatible and the communication cannot continue.

		Daemon Setting		
		NEVER	OPTIONAL	REQUIRED
Client Setting	NEVER	No	No	Fail
	REQUIRED	Fail	Yes	Yes

Table 3.2: Resolution of security negotiation.

Enabling authentication, encryption, and integrity checks is dependent on security negotiation taking place. The enabled security negotiation further sets the policy for these other features. Table 3.3 shows how security features are resolved for client-daemon combinations of security feature policy settings. Like Table 3.2, Yes means the feature will be utilized. No means it will not. Fail implies incompatibility and the feature cannot be resolved.

		Daemon Setting			
		NEVER	OPTIONAL	PREFERRED	REQUIRED
Client Setting	NEVER	No	No	No	Fail
	OPTIONAL	No	No	Yes	Yes
	PREFERRED	No	Yes	Yes	Yes
	REQUIRED	Fail	Yes	Yes	Yes

Table 3.3: Resolution of security features.

The enabling of encryption and/or integrity checks is dependent on authentication taking place. The authentication provides a key exchange. The key is needed for both encryption and integrity checks.

Setting SEC_CLIENT_<feature> determines the policy for all outgoing commands. The policy for incoming commands (the daemon side of the communication) takes a more fine-grained approach that implements a set of access levels for the received command. For example, it is desirable to have all incoming administrative requests require authentication. Inquiries on pool status may not be so restrictive. To implement this, the administrator configures the policy:

```
SEC_ADMINISTRATOR_AUTHENTICATION = REQUIRED
SEC_READ_AUTHENTICATION          = OPTIONAL
```

The DEFAULT value for <context> provides a way to set a policy for all access levels (READ, WRITE, etc.) that do not have a specific configuration variable defined. In addition, some access levels will default to the settings specified for other access levels. For example, defaults to `DAEMON`, and `DAEMON` defaults to `WRITE`, which then defaults to the general DEFAULT setting.

Configuration for Security Methods

Authentication and encryption can each be accomplished by a variety of methods or technologies. Which method is utilized is determined during security negotiation.

The configuration macros that determine the methods to use for authentication and/or encryption are

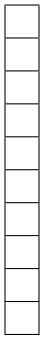
```
SEC_<context>_AUTHENTICATION_METHODS
SEC_<context>_CRYPTO_METHODS
```

These macros are defined by a comma or space delimited list of possible methods to use. The *Authentication* section lists all implemented authentication methods. The *Encryption* section lists all implemented encryption methods.

5.9.6 Authentication

The client side of any communication uses one of two macros to specify whether authentication is to occur:

For the daemon side, there are a larger number of macros to specify whether authentication is to take place, based upon the necessary access level:



As an example, the macro defined in the configuration file for a daemon as

```
SEC_WRITE_AUTHENTICATION = REQUIRED
```

signifies that the daemon must authenticate the client for any communication that requires the `WRITE` access level. If the daemon's configuration contains

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

and does not contain any other security configuration for AUTHENTICATION, then this default defines the daemon's needs for authentication over all access levels. Where a specific macro is defined, the more specific value takes precedence over the default definition.

If authentication is to be done, then the communicating parties must negotiate a mutually acceptable method of authentication to be used. A list of acceptable methods may be provided by the client, using the macros

A list of acceptable methods may be provided by the daemon, using the macros

The methods are given as a comma-separated list of acceptable values. These variables list the authentication methods that are available to be used. The ordering of the list defines preference; the first item in the list indicates the highest preference. As not all of the authentication methods work on Windows platforms, which ones do not work on Windows are indicated in the following list of defined values:

```
SSL
KERBEROS
PASSWORD
FS        (not available on Windows platforms)
FS_REMOTE (not available on Windows platforms)
```

(continues on next page)

(continued from previous page)

```
IDTOKENS
SCITOKENS
NTSSPI
MUNGE
CLAIMTOBE
ANONYMOUS
```

For example, a client may be configured with:

```
SEC_CLIENT_AUTHENTICATION_METHODS = FS, SSL
```

and a daemon the client is trying to contact with:

```
SEC_DEFAULT_AUTHENTICATION_METHODS = SSL
```

Security negotiation will determine that SSL authentication is the only compatible choice. If there are multiple compatible authentication methods, security negotiation will make a list of acceptable methods and they will be tried in order until one succeeds.

As another example, the macro

```
SEC_DEFAULT_AUTHENTICATION_METHODS = KERBEROS, NTSSPI
```

indicates that either Kerberos or Windows authentication may be used, but Kerberos is preferred over Windows. Note that if the client and daemon agree that multiple authentication methods may be used, then they are tried in turn. For instance, if they both agree that Kerberos or NTSSPI may be used, then Kerberos will be tried first, and if there is a failure for any reason, then NTSSPI will be tried.

An additional specialized method of authentication exists for communication between the *condor_schedd* and *condor_startd*, as well as communication between the *condor_schedd* and the *condor_negotiator*. It is especially useful when operating at large scale over high latency networks or in situations where it is inconvenient to set up one of the other methods of authentication between the submit and execute daemons. See the description of `SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION` in *Configuration File Entries Relating to Security* for details.

If the configuration for a machine does not define any variable for `SEC_<access-level>_AUTHENTICATION`, then HTCondor uses a default value of OPTIONAL. Authentication will be required for any operation which modifies the job queue, such as *condor_qedit* and *condor_rm*. If the configuration for a machine does not define any variable for `SEC_<access-level>_AUTHENTICATION_METHODS`, the default value for a Unix machine is FS, IDTOKENS, KERBEROS. This default value for a Windows machine is NTSSPI, IDTOKENS, KERBEROS.

SSL Authentication

SSL authentication utilizes X.509 certificates to establish trust between a client and a server.

SSL authentication may be mutual or server-only. That is, the server always needs a certificate that can be verified by the client, but a certificate for the client may be optional. Whether a client certificate is required is controlled by configuration variable , a boolean value that defaults to `False`. If the value is `False`, then the client may present a certificate to be verified by the server. If the client doesn't have a certificate, then its identity is set to `unauthenticated` by the server. If the value is `True` and the client doesn't have a certificate, then the SSL authentication fails (other authentication methods may then be tried).

The names and locations of keys and certificates for clients, servers, and the files used to specify trusted certificate authorities (CAs) are defined by settings in the configuration files. The contents of the files are identical in format and interpretation to those used by other systems which use SSL, such as Apache httpd.

The configuration variables and specify the file location for the certificate file for the initiator and recipient of connections, respectively. Similarly, the configuration variables and specify the locations for keys. If no client certificate is used, the client will authenticate as user `anonymous@ssl`.

The configuration variables and each specify a path and file name, providing the location of a file containing one or more certificates issued by trusted certificate authorities. Similarly, and each specify a directory with one or more files, each which may contain a single CA certificate. The directories must be prepared using the OpenSSL `c_rehash` utility.

Bootstrapping SSL Authentication

HTCondor daemons exposed to the Internet may utilize server certificates provided by well-known authorities; however, SSL can be difficult to bootstrap for non-public hosts.

Accordingly, on first startup, if is `True`, the *condor_collector* generates a new CA and key in the locations pointed to by and , respectively. If or do not exist, the collector will generate a host certificate and key using the generated CA and write them to the respective locations.

The first time an unknown CA is encountered by tool such as `condor_status`, the tool will prompt the user on whether it should trust the CA; the prompt looks like the following:

```
$ condor_status
The remote host collector.wisc.edu presented an untrusted CA certificate with the␣
↪following fingerprint:
SHA-256: 781b:1d:1:ca:b:f7:ab:b6:e4:a3:31:80:ae:28:9d:b0:a9:ee:1b:c1:63:8b:62:29:83:1f:
↪e7:88:29:75:6:
Subject: /O=condor/CN=hcc-briantest7.unl.edu
Would you like to trust this server for current and future communications?
Please type 'yes' or 'no':
```

The result will be persisted in a file at `.condor/known_hosts` inside the user's home directory.

Similarly, a daemon authenticating as a client against a remote server will record the result of the authentication in a system-wide trust whose location is kept in the configuration variable . Since a daemon cannot prompt the administrator for a decision, it will always deny unknown CAs _unless_ is set to `true`.

The first time any daemon is authenticated, even if it's not through SSL, it will be noted in the `known_hosts` file.

The format of the `known_hosts` file is line-oriented and has three fields,

```
HOSTNAME METHOD CERTIFICATE_DATA
```

Any blank line or line prefixed with # will be ignored. Any line prefixed with `!` will result in the CA certificate to _not_ be trusted. To easily switch an untrusted CA to be trusted, simply delete the `!` prefix.

For example, collector.wisc.edu would be trusted with this file entry using SSL:

```
collector.wisc.edu SSL␣
↪MIIBvjCCAWSgAwIBAgIJAJRheVnN5ZDyMAoGCCqGSM49BAMCMDIxDzANBgNVBAoMBmNvbmRvcjEfMB0GA1UEAwwWaGNjLWJyaWFud
↪EqHYp+wri/aAKyDrLM5R1lWX44jSykgIpTOCLJUS/
↪ajYzBhMB0GA1UdDgQWBBRBPe8Ga9Q7X3F198fWBSg6VT1DZDAfBgNVHSMEGDAWgBRBPe8Ga9Q7X3F198fWBSg6VT1DZDAPBgNVHRM
↪MA4GA1UdDwEB/wQEAwICBDAKBggqhkjOPQQDAgNIADBFAiARfW+suELxSzSdi9u20hFs/
↪aSXpd+gwJ6Ne8jjG+y/2AIhAO6f3ff9nnYRmesFbvt1lv+LosOMbeiUdVoaKFOGIyuJ
```

The following line would cause collector.wisc.edu to _not_ be trusted:

```
!collector.wisc.edu SSL␣
↪MIIBvjCCAWSgAwIBAgIJAJRheVnN5ZDyMAoGCCqGSM49BAMCMDIxDzANBgNVBAoMBmNvbmRvcjEfMB0GA1UEAwwWaGNjLWJyaWFud
↪EqHYp+wri/aAKyDrLM5R1lWX44jSykgIpTOCLJUS/
↪ajYzBhMB0GA1UdDgQWBBRBPe8Ga9Q7X3F198fWBSg6VT1DZDAfBgNVHSMEGDAWgBRBPe8Ga9Q7X3F198fWBSg6VT1DZDAPBgNVHRM
↪MA4GA1UdDwEB/wQEAwICBDAKBggqhkjOPQQDAgNIADBFAiARfW+suELxSzSdi9u20hFs/
↪aSXpd+gwJ6Ne8jjG+y/2AIhAO6f3ff9nnYRmesFbvt1lv+LosOMbeiUdVoaKFOGIyuJ
```

Kerberos Authentication

If Kerberos is used for authentication, then a mapping from a Kerberos domain (called a realm) to an HTCondor UID domain is necessary. There are two ways to accomplish this mapping. For a first way to specify the mapping, see *The Unified Map File for Authentication* to use HTCondor's unified map file. A second way to specify the mapping is to set the configuration variable to the path of an administrator-maintained Kerberos-specific map file. The configuration syntax is

```
KERBEROS_MAP_FILE = /path/to/etc/condor.kmap
```

Lines within this map file have the syntax

```
KERB.REALM = UID.domain.name
```

Here are two lines from a map file to use as an example:

```
CS.WISC.EDU   = cs.wisc.edu
ENGR.WISC.EDU = ee.wisc.edu
```

If a configuration variable is defined and set, then all permitted realms must be explicitly mapped. If no map file is specified, then HTCondor assumes that the Kerberos realm is the same as the HTCondor UID domain.

The configuration variable defines the name of a Kerberos principal, to override the default `host/<hostname>@<realm>`. A principal specifies a unique name to which a set of credentials may be assigned.

The configuration variable defines a Kerberos service to override the default `host`. HTCondor prefixes this to `/<hostname>@<realm>` to obtain the default Kerberos principal. Configuration variable `KERBEROS_SERVER_PRINCIPAL` overrides `KERBEROS_SERVER_SERVICE`.

For example, the configuration

```
KERBEROS_SERVER_SERVICE = condor-daemon
```

results in HTCondor's use of

```
condor-daemon/the.host.name@YOUR.KERB.REALM
```

as the server principal.

Here is an example of configuration settings that use Kerberos for authentication and require authentication of all communications of the write or administrator access level.

```
SEC_WRITE_AUTHENTICATION                 = REQUIRED
SEC_WRITE_AUTHENTICATION_METHODS         = KERBEROS
SEC_ADMINISTRATOR_AUTHENTICATION         = REQUIRED
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS = KERBEROS
```

Kerberos authentication on Unix platforms requires access to various files that usually are only accessible by the root user. At this time, the only supported way to use KERBEROS authentication on Unix platforms is to start daemons HTCondor as user root.

Password Authentication

The password method provides mutual authentication through the use of a shared secret. This is often a good choice when strong security is desired, but an existing Kerberos or X.509 infrastructure is not in place. Password authentication is available on both Unix and Windows. It currently can only be used for daemon-to-daemon authentication. The shared secret in this context is referred to as the pool password.

Before a daemon can use password authentication, the pool password must be stored on the daemon's local machine. On Unix, the password will be placed in a file defined by the configuration variable . This file will be accessible only by the UID that HTCondor is started as. On Windows, the same secure password store that is used for user passwords will be used for the pool password (see the *Secure Password Storage* section).

Under Unix, the password file can be generated by using the following command to write directly to the password file:

```
$ condor_store_cred -f /path/to/password/file
```

Under Windows (or under Unix), storing the pool password is done with the **-c** option when using to *condor_store_cred* **add**. Running

```
$ condor_store_cred -c add
```

prompts for the pool password and store it on the local machine, making it available for daemons to use in authentication. The *condor_master* must be running for this command to work.

In addition, storing the pool password to a given machine requires CONFIG-level access. For example, if the pool password should only be set locally, and only by root, the following would be placed in the global configuration file.

```
ALLOW_CONFIG = root@mydomain/$(IP_ADDRESS)
```

It is also possible to set the pool password remotely, but this is recommended only if it can be done over an encrypted channel. This is possible on Windows, for example, in an environment where common accounts exist across all the machines in the pool. In this case, ALLOW_CONFIG can be set to allow the HTCondor administrator (who in this example has an account condor common to all machines in the pool) to set the password from the central manager as follows.

```
ALLOW_CONFIG = condor@mydomain/$(CONDOR_HOST)
```

The HTCondor administrator then executes

```
$ condor_store_cred -c -n host.mydomain add
```

from the central manager to store the password to a given machine. Since the condor account exists on both the central manager and host.mydomain, the NTSSPI authentication method can be used to authenticate and encrypt the connection. *condor_store_cred* will warn and prompt for cancellation, if the channel is not encrypted for whatever reason (typically because common accounts do not exist or HTCondor's security is misconfigured).

When a daemon is authenticated using a pool password, its security principle is condor_pool@$(UID_DOMAIN), where $(UID_DOMAIN) is taken from the daemon's configuration. The ALLOW_DAEMON and ALLOW_NEGOTIATOR configuration variables for authorization should restrict access using this name. For example,

```
ALLOW_DAEMON = condor_pool@mydomain/*, condor@mydomain/$(IP_ADDRESS)
ALLOW_NEGOTIATOR = condor_pool@mydomain/$(CONDOR_HOST)
```

This configuration allows remote DAEMON-level and NEGOTIATOR-level access, if the pool password is known. Local daemons authenticated as condor@mydomain are also allowed access. This is done so local authentication can be done using another method such as FS.

If there is no pool password available on Linux, the *condor_collector* will automatically generate one. This is meant to ease the configuration of freshly-installed clusters; for POOL authentication, the HTCondor administrator only needs to copy this file to each host in the cluster.

Example Security Configuration Using Pool Password

The following example configuration uses pool password authentication and network message integrity checking for all communication between HTCondor daemons.

```
SEC_PASSWORD_FILE = $(LOCK)/pool_password
SEC_DAEMON_AUTHENTICATION = REQUIRED
SEC_DAEMON_INTEGRITY = REQUIRED
SEC_DAEMON_AUTHENTICATION_METHODS = PASSWORD
SEC_NEGOTIATOR_AUTHENTICATION = REQUIRED
SEC_NEGOTIATOR_INTEGRITY = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION_METHODS = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS
ALLOW_DAEMON = condor_pool@$(UID_DOMAIN)/*.cs.wisc.edu, \
               condor@$(UID_DOMAIN)/$(IP_ADDRESS)
ALLOW_NEGOTIATOR = condor_pool@$(UID_DOMAIN)/negotiator.machine.name
```

Example Using Pool Password for *condor_startd* Advertisement

One problem with the pool password method of authentication is that it involves a single, shared secret. This does not scale well with the addition of remote users who flock to the local pool. However, the pool password may still be used for authenticating portions of the local pool, while others (such as the remote *condor_schedd* daemons involved in flocking) are authenticated by other means.

In this example, only the *condor_startd* daemons in the local pool are required to have the pool password when they advertise themselves to the *condor_collector* daemon.

```
SEC_PASSWORD_FILE = $(LOCK)/pool_password
SEC_ADVERTISE_STARTD_AUTHENTICATION = REQUIRED
SEC_ADVERTISE_STARTD_INTEGRITY = REQUIRED
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS
ALLOW_ADVERTISE_STARTD = condor_pool@$(UID_DOMAIN)/*.cs.wisc.edu
```

Token Authentication

Password authentication requires both parties (client and server) in an authenticated session to have access to the same password file. Further, both client and server authenticate the remote side as the user `condor_pool` which, by default, has a high level of privilege to the entire pool. Hence, it is only reasonable for daemon-to-daemon authentication. Further, as only *one* password is allowed, it is impossible to use `PASSWORD` authentication to flock to a remote pool.

Token-based authentication is a newer extension to `PASSWORD` authentication that allows the pool administrator to generate new, low-privilege tokens using one of several pool signing keys. It also allows a daemon or tool to authenticate to a remote pool without having that pool's password. As tokens are derived from a specific signing key, if an administrator removes a signing key from the directory specified in , then all derived tokens are immediately invalid. Most simple installs will utilize a single signing key, named `POOL`.

While most token signing keys are placed in the directory specified by `SEC_PASSWORD_DIRECTORY`, with the filename within the directory determining the key's name, the `POOL` token signing key can be located elsewhere by setting to the full pathname of the desired file. On Linux the same file can be both the pool signing key and the pool password if and `SEC_TOKEN_POOL_SIGNING_KEY_FILE` refer to the same file. However this is not preferred because in order to properly interoperate with older versions of HTCondor the pool password will be read as a text file and truncated at the first NUL character. This differs from the pool signing key which is read as binary in HTCondor 9.0. Some 8.9 releases used the pool password as the pool signing key for tokens, those versions will not interoperate with 9.0 if the pool signing key file contains NUL characters.

The *condor_collector* process will automatically generate the pool signing key named `POOL` on startup if that file does not exist.

To generate a token, the administrator may utilize the `condor_token_create` command-line utility:

```
$ condor_token_create -identity frida@pool.example.com
```

The resulting token may be given to Frida and appended to a file in the directory specified by (defaults to `~/.condor/tokens.d`). Subsequent authentications to the pool will utilize this token and cause Frida to be authenticated as the identity `frida@pool.example.com`. For daemons, tokens are stored in ; on Unix platforms, this defaults to `/etc/condor/tokens.d` which should be a directory with permissions that only allow read and write access by user root.

Note that each pool signing key is named (the pool signing key defaults to the special name `POOL`) by its corresponding filename in ; HTCondor will assume that, for all daemons in the same *trust domain* (defaulting to the HTCondor pool) will have the same signing key for the same name. That is, the signing key contained in `key1` in host `pool.example.com` is identical to the signing key contained in `key1` in host `submit.example.com`.

Unlike pool passwords, tokens can have a limited lifetime and can limit the authorizations allowed to the client. For example,

```
$ condor_token_create -identity condor@pool.example.com \
      -lifetime 3600 \
      -authz ADVERTISE_STARTD
```

will create a new token that maps to user `condor@pool.example.com`. However, this token is *only* valid for the `ADVERTISE_STARTD` authorization, regardless of what the server has configured for the `condor` user (the intersection of the identity's configured authorization and the token's authorizations, if specified, are used). Further, the token will only be valid for 3600 seconds (one hour).

In many cases, it is difficult or awkward for the administrator to securely provide the new token to the user; an email or text message from administrator to user is typically insufficiently secure to send the token (especially as old emails are often archived for many years). In such a case, the user may instead anonymously *request* a token from the administrator. The user will receive a request ID, which the administrator will need in order to approve the request. The ID (typically, a 7 digit number) is easier to communicate over the phone (compared to the token, which is hundreds of characters long). Importantly, neither user nor administrator is responsible for securely moving the token - e.g., there is no chance it will be leaked into an email archive.

If a *condor_master*, *condor_startd*, or *condor_schedd* daemon cannot authenticate with the collector, it will automatically perform a token request from the collector.

To use the token request workflow, the user needs a confidential channel to the server or an appropriate auto-approval rule needs to be in place. The simplest way to establish a confidential channel is using *SSL Authentication* without a client certificate; configure the collector using a host certificate.

Using the SSL authentication, the client can request a new authentication token:

```
$ condor_token_request
Token request enqueued.  Ask an administrator to please approve request 9235785.
```

This will enqueue a request for a token corresponding to the superuser `condor`; the HTCondor pool administrator will subsequently need to approve request `9235785` using the `condor_token_request_approve` tool.

If the host trusts requests coming from a specific network (i.e., the same administrator manages the network and no unprivileged users are currently on the network), then the auto-approval mechanism may be used. When in place, auto-approval allows any token authentication request on an approved network to be automatically approved by HTCondor on behalf of the pool administrator - even when requests do not come over confidential connections.

When a daemon issues a token for a client (e.g. for `condor_token_fetch` or `condor_token_request`), the signing key it uses must appear in the list . If the client doesn't request a specific signing key to use, then the key given by is used. The default for both of these configuration parameters is `POOL`.

If there are multiple tokens in files in the , then the daemon will search for tokens in that directory based on lexicographical order; the exception is that the file `$(SUBSYS)_auto_generated_token` will be searched first for daemons of type `$(SUBSYS)`. For example, if `SEC_TOKEN_SYSTEM_DIRECTORY` is set to `/etc/condor/tokens.d`, then the *condor_schedd* will search at `/etc/condor/tokens.d/SCHEDD_auto_generated_token` by default.

Users may create their own tokens with `condor_token_fetch`. This command-line utility will contact the default `condor_schedd` and request a new token given the user's authenticated identity. Unlike `condor_token_create`, the `condor_token_fetch` has no control over the mapped identity (but does not need to read the files in).

If no security authentication methods specified by the administrator - and the daemon or user has access to at least one token - then `IDTOKENS` authentication is automatically added to the list of valid authentication methods. Otherwise, to setup `IDTOKENS` authentication, enable it in the list of authentication methods:

```
SEC_DEFAULT_AUTHENTICATION_METHODS=$(SEC_DEFAULT_AUTHENTICATION_METHODS), IDTOKENS
SEC_CLIENT_AUTHENTICATION_METHODS=$(SEC_CLIENT_AUTHENTICATION_METHODS), IDTOKENS
```

Revoking Token: If a token is lost, stolen, or accidentally exposed, then the system administrator may use the token revocation mechanism in order to prevent unauthorized use. Revocation can be accomplished by setting the configuration parameter; when set, the value of this parameter will be evaluated as a ClassAd expression against the token's contents.

For example, consider the following token:

```
eyJhbGciOiJIUzI1NiIsImtpZCI6IlBPT0wifQ.
↪eyJpYXQiOjE1ODg0NzQ3MTksImlzcyI6ImhjYy1icmlhbnRlc3Q3LnVubC5lZHUiLCJqdGkiOiJjNzYwYzJhZjE5M2ExZmQ0ZTQwYm
↪fiqfgwjyTkxMSdxwm84xxMTVcGfearddEDj_rhiIbi4ummU
```

When printed using `condor_token_list`, the human-readable form is as follows (line breaks added for readability):

```
$ condor_token_list
Header: {"alg":"HS256","kid":"POOL"}
Payload: {
    "iat": 1588474719,
    "iss": "pool.example.com",
```

(continues on next page)

(continued from previous page)

```
    "jti": "c760c2af193a1fd4e40bc9c53c96ee7c",
    "sub": "alice@pool.example.com"
}
```

If we would like to revoke this token, we could utilize any of the following values for SEC_TOKEN_REVOCATION_EXPR, depending on the desired breadth of the revocation:

```
# Revokes all tokens from the user Alice:
SEC_TOKEN_REVOCATION_EXPR = sub =?= "alice@pool.example.com"

# Revokes all tokens from Alice issued before or after this one:
SEC_TOKEN_REVOCATION_EXPR = sub =?= "alice@pool.example.com" && \
    iat <= 1588474719

# Revokes *only* this token:
SEC_TOKEN_REVOCATION_EXPR = jti =?= "c760c2af193a1fd4e40bc9c53c96ee7c"
```

The revocation only works on the daemon where is set; to revoke a token across the entire pool, set on every host.

In order to invalidate all tokens issued by a given master password in , simply remove the file from the directory.

File System Authentication

This form of authentication utilizes the ownership of a file in the identity verification of a client. A daemon authenticating a client requires the client to write a file in a specific location (/tmp). The daemon then checks the ownership of the file. The file's ownership verifies the identity of the client. In this way, the file system becomes the trusted authority. This authentication method is only appropriate for clients and daemons that are on the same computer.

File System Remote Authentication

Like file system authentication, this form of authentication utilizes the ownership of a file in the identity verification of a client. In this case, a daemon authenticating a client requires the client to write a file in a specific location, but the location is not restricted to /tmp. The location of the file is specified by the configuration variable .

Windows Authentication

This authentication is done only among Windows machines using a proprietary method. The Windows security interface SSPI is used to enforce NTLM (NT LAN Manager). The authentication is based on challenge and response, using the user's password as a key. This is similar to Kerberos. The main difference is that Kerberos provides an access token that typically grants access to an entire network, whereas NTLM authentication only verifies an identity to one machine at a time. NTSSPI is best-used in a way similar to file system authentication in Unix, and probably should not be used for authentication between two computers.

SciTokens Authentication

A SciToken is a form of JSON Web Token (JWT) that the client can present that the server can verify. Authentication of the server by the client is done via an SSL host certificate (the same as with SSL authentication). More information about SciTokens can be found at https://scitokens.org.

Some other JWT token types can be used with the SciTokens authentication method. WLCG tokens are accepted automatically. Other token types, such as EGI CheckIn tokens, require some relaxation of the SciTokens validation checks. Configuration parameter determines whether any tokens will be accepted under these relaxed checks. It's a boolean value that defaults to `True`. Configuration parameter determines which issuers' tokens will be accepted under these relaxed checks. It's a list of issuer URLs that defaults to the EGI CheckIn issuer. These parameters should be used with caution, as they disable some security checks.

Ask MUNGE for Authentication

Ask the MUNGE service to validate both sides of the authentication. See: https://dun.github.io/munge/ for instructions on installing.

Claim To Be Authentication

Claim To Be authentication accepts any identity claimed by the client. As such, it does not authenticate. It is included in HTCondor and in the list of authentication methods for testing purposes only.

Anonymous Authentication

Anonymous authentication causes authentication to be skipped entirely. As such, it does not authenticate. It is included in HTCondor and in the list of authentication methods for testing purposes only.

5.9.7 The Unified Map File for Authentication

HTCondor's unified map file allows the mappings from authenticated names to an HTCondor canonical user name to be specified as a single list within a single file. The location of the unified map file is defined by the configuration variable ; it specifies the path and file name of the unified map file. Each mapping is on its own line of the unified map file. Each line contains either an `@include` directive, or 3 fields, separated by white space (space or tab characters):

1. The name of the authentication method to which the mapping applies.
2. A name or a regular expression representing the authenticated name to be mapped.
3. The canonical HTCondor user name.

Allowable authentication method names are the same as used to define any of the configuration variables , as repeated here:

```
SSL
KERBEROS
PASSWORD
FS
FS_REMOTE
IDTOKENS
SCITOKENS
```

(continues on next page)

(continued from previous page)

```
NTSSPI
MUNGE
CLAIMTOBE
ANONYMOUS
```

The fields that represent an authenticated name and the canonical HTCondor user name may utilize regular expressions as defined by PCRE2 (Perl-Compatible Regular Expressions). Due to this, more than one line (mapping) within the unified map file may match. Look ups are therefore defined to use the first mapping that matches.

For HTCondor version 8.5.8 and later, the authenticated name field will be interpreted as a regular expression or as a simple string based on the value of the configuration variable. If this configuration variable is true, then the authenticated name field is a regular expression only when it begins and ends with the / character. If this configuration variable is false, or on HTCondor versions older than 8.5.8, the authenticated name field is always a regular expression.

A regular expression may need to contain spaces, and in this case the entire expression can be surrounded by double quote marks. If a double quote character also needs to appear in such an expression, it is preceded by a backslash.

If the first field is the special value `@include`, it should be followed by a file or directory path in the second field. If a file is specified, it will be read and parsed as map file. If a directory is specified, then each file in the directory is read as a map file unless the name of the file matches the pattern specified in the configuration variable. Files in the directory are read in lexical order. When a map file is read as a result of an `@include` statement, any `@include` statements that it contains will be ignored. If the file or directory path specified with an `@include` statement is a relative path, it will be treated as relative to the file currently being read.

The default behavior of HTCondor when no map file is specified is to do the following mappings, with some additional logic noted below:

```
FS (.*) \1
FS_REMOTE (.*) \1
SSL (.*) ssl@unmapped
KERBEROS ([^/]*)/?[^@]*@(.*) \1@\2
NTSSPI (.*) \1
MUNGE (.*) \1
CLAIMTOBE (.*) \1
PASSWORD (.*) \1
SCITOKENS .* PLUGIN:*
```

For SciTokens, the authenticated name is the `iss` and `sub` claims of the token, separated by a comma.

For Kerberos, if is specified, the domain portion of the name is obtained by mapping the Kerberos realm to the value specified in the map file, rather than just using the realm verbatim as the domain portion of the condor user name. See the *Authentication* section for details.

If authentication did not happen or failed and was not required, then the user is given the name unauthenticated@unmapped.

SciTokens Mapping Plugins

For SciTokens, the `iss` and `sub` claims of the token may not be sufficient to map the token to the appropriate canonical HTCondor user name. For these situations, a series of plugins can be employed to perform the mapping based on the full token payload. Each plugin can accept the token and provide a mapped identity or decline the token. If the plugin declines, then additional plugins are consulted. If all plugins decline the token, then the mapped identity `scitokens@unmapped` is used.

Each plugin is given a name consisting of alphanumeric characters. To use a set of plugins to perform a mapping, the third field of the matching line in the map file (the canonical name) should be the text `PLUGIN:` followed by a comma-separated list of plugin names. Note that no spaces should be used within the list.

For each plugin, the configuration parameter gives the executable and optional command line arguments needed to invoke the plugin. The optional configuration parameter specifies the mapped identity if the plugin accepts the token. If this parameter isn't set, then the plugin must write the mapped identity to its stdout. If the special value `PLUGIN:*` is given in the map file, then the configuration parameter is consulted to determine the names of the plugins to run.

When a plugin is invoked, the given binary is run. The payload of the token is provided via stdin and a series of environment variables (compatible with those set by ARC CE for its token plugins). If the plugin exits with status 0, then it accepts the token. If the plugin exits with status 1, then it declines the token and other plugins may be consulted. If the plugin exits with any other status, the entire mapping procedure fails and the client is rejected.

Here's an example where one plugin is used for tokens from a specific issuer, and two other plugins are used for tokens from all other issuers. The first plugin has a fixed mapping given via configuration, while the other plugins will write the mapping to their stdout. The last plugin uses a command-line argument.

First, this would appear in the map file:

```
# Mapfile snippet:
# Plugin for specific token issuer
SCITOKENS ^https://phys.uz.edu, PLUGIN:A

# Plugins for all other token issuers
SCITOKENS .* PLUGIN:B,C
```

Then, this would appear in the configuration files:

```
# Configuration file snippet:
# Plugin A for specific issuer with fixed mapping result
SEC_SCITOKENS_PLUGIN_A_COMMAND = $(LIBEXEC)/A.plugin
SEC_SCITOKENS_PLUGIN_A_MAPPING = physgrp

# Plugins B,C for all other tokens
SEC_SCITOKENS_PLUGIN_B_COMMAND = $(LIBEXEC)/B.plugin
SEC_SCITOKENS_PLUGIN_C_COMMAND = $(LIBEXEC)/C.plugin -A
```

5.9.8 Encryption

Encryption provides privacy support between two communicating parties. Through configuration macros, both the client and the daemon can specify whether encryption is required for further communication.

The client uses one of two macros to enable or disable encryption:

For the daemon, there are many macros to enable or disable encryption:

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_ENCRYPTION = REQUIRED
```

signifies that any communication that changes a daemon's configuration must be encrypted. If a daemon's configuration contains

```
SEC_DEFAULT_ENCRYPTION = REQUIRED
```

and does not contain any other security configuration for ENCRYPTION, then this default defines the daemon's needs for encryption over all access levels. Where a specific macro is present, its value takes precedence over any default given.

If encryption is to be done, then the communicating parties must find (negotiate) a mutually acceptable method of encryption to be used. A list of acceptable methods may be provided by the client, using the macros and

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_CLIENT_CRYPTO_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

The methods are given as a comma-separated list of acceptable values. These variables list the encryption methods that are available to be used. The ordering of the list gives preference; the first item in the list indicates the highest preference. Possible values are

```
AES
BLOWFISH
3DES
```

As of version 9.0.2 HTCondor can be configured to be FIPS compliant. This disallows BLOWFISH as an encryption method. Please see the *FIPS* section below.

5.9.9 Integrity

An integrity check assures that the messages between communicating parties have not been tampered with. Any change, such as addition, modification, or deletion can be detected. Through configuration macros, both the client and the daemon can specify whether an integrity check is required of further communication.

The client uses one of two macros to enable or disable an integrity check:

For the daemon, there are macros to enable or disable an integrity check:

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_INTEGRITY = REQUIRED
```

signifies that any communication that changes a daemon's configuration must have its integrity assured. If a daemon's configuration contains

```
SEC_DEFAULT_INTEGRITY = REQUIRED
```

and does not contain any other security configuration for INTEGRITY, then this default defines the daemon's needs for integrity checks over all access levels. Where a specific macro is present, its value takes precedence over any default given.

If `AES` encryption is used for a connection, then a secure checksum is included within the AES data regardless of any INTEGRITY settings.

If another type of encryption was used (i.e. `BLOWFISH` or `3DES`), then a signed MD5 check sum is the only available method for integrity checking. Its use is implied whenever integrity checks occur.

As of version 9.0.2 HTCondor can be configured to be FIPS compliant. This disallows MD5 as an integrity method. We suggest you use AES encryption as the AES-GCM mode we have implemented also provides integrity checks. Please see the *FIPS* section below.

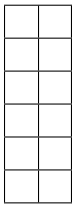
5.9.10 Authorization

Authorization protects resource usage by granting or denying access requests made to the resources. It defines who is allowed to do what.

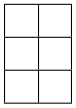
Authorization is defined in terms of users. An initial implementation provided authorization based on hosts (machines), while the current implementation relies on user-based authorization. The *Host-Based Security in HTCondor* section describes the previous implementation. This IP/Host-Based security still exists, and it can be used, but significantly stronger and more flexible security can be achieved with the newer authorization based on fully qualified user names. This section discusses user-based authorization.

The authorization portion of the security of an HTCondor pool is based on a set of configuration macros. The macros list which user will be authorized to issue what request given a specific access level. When a daemon is to be authorized, its user name is the login under which the daemon is executed.

These configuration macros define a set of users that will be allowed to (or denied from) carrying out various HTCondor commands. Each access level may have its own list of authorized users. A complete list of the authorization macros:



In addition, the following are used to control authorization of specific types of HTCondor daemons when advertising themselves to the pool. If unspecified, these default to the broader `ALLOW_DAEMON` and `DENY_DAEMON` settings.



Each client side of a connection may also specify its own list of trusted servers. This is done using the following settings. Note that the FS and CLAIMTOBE authentication methods are not symmetric. The client is authenticated by the server, but the server is not authenticated by the client. When the server is not authenticated to the client, only the network address of the host may be authorized and not the specific identity of the server.

```
ALLOW_CLIENT
DENY_CLIENT
```

The names and should be thought of as "when I am acting as a client, these are the servers I allow or deny." It should not be confused with the incorrect thought "when I am the server, these are the clients I allow or deny."

All authorization settings are defined by a comma-separated list of fully qualified users. Each fully qualified user is described using the following format:

```
username@domain/hostname
```

The information to the left of the slash character describes a user within a domain. The information to the right of the slash character describes one or more machines from which the user would be issuing a command. This host name may take the form of either a fully qualified host name of the form

```
bird.cs.wisc.edu
```

or an IP address of the form

```
128.105.128.0
```

An example is

```
zmiller@cs.wisc.edu/bird.cs.wisc.edu
```

Within the format, wild card characters (the asterisk, *) are allowed. The use of wild cards is limited to one wild card on either side of the slash character. A wild card character used in the host name is further limited to come at the beginning of a fully qualified host name or at the end of an IP address. For example,

```
*@cs.wisc.edu/bird.cs.wisc.edu
```

refers to any user that comes from cs.wisc.edu, where the command is originating from the machine bird.cs.wisc.edu. Another valid example,

```
zmiller@cs.wisc.edu/*.cs.wisc.edu
```

refers to commands coming from any machine within the cs.wisc.edu domain, and issued by zmiller. A third valid example,

```
*@cs.wisc.edu/*
```

refers to commands coming from any user within the cs.wisc.edu domain where the command is issued from any machine. A fourth valid example,

```
*@cs.wisc.edu/128.105.*
```

refers to commands coming from any user within the cs.wisc.edu domain where the command is issued from machines within the network that match the first two octets of the IP address.

If the set of machines is specified by an IP address, then further specification using a net mask identifies a physical set (subnet) of machines. This physical set of machines is specified using the form

```
network/netmask
```

The network is an IP address. The net mask takes one of two forms. It may be a decimal number which refers to the number of leading bits of the IP address that are used in describing a subnet. Or, the net mask may take the form of

```
a.b.c.d
```

where a, b, c, and d are decimal numbers that each specify an 8-bit mask. An example net mask is

```
255.255.192.0
```

which specifies the bit mask

```
11111111.11111111.11000000.00000000
```

A single complete example of a configuration variable that uses a net mask is

```
ALLOW_WRITE = joesmith@cs.wisc.edu/128.105.128.0/17
```

User joesmith within the cs.wisc.edu domain is given write authorization when originating from machines that match their leftmost 17 bits of the IP address.

For Unix platforms where netgroups are implemented, a netgroup may specify a set of fully qualified users by using an extension to the syntax for all configuration variables of the form ALLOW_* and DENY_*. The syntax is the plus sign character (+) followed by the netgroup name. Permissions are applied to all members of the netgroup.

This flexible set of configuration macros could be used to define conflicting authorization. Therefore, the following protocol defines the precedence of the configuration macros.

1. DENY_* macros take precedence over where there is a conflict. This implies that if a specific user is both denied and granted authorization, the conflict is resolved by denying access.
2. If macros are omitted, the default behavior is to deny authorization for all users.

In addition, there are some hard-coded authorization rules that cannot be modified by configuration.

1. Connections with a name matching *@unmapped are not allowed to do any job management commands (e.g. submitting, removing, or modifying jobs). This prevents these operations from being done by unauthenticated users and users who are authenticated but lacking a name in the map file.
2. To simplify flocking, the *condor_schedd* automatically grants the *condor_startd* READ access for the duration of a claim so that claim-related communications are possible. The *condor_shadow* grants the *condor_starter* DAEMON access so that file transfers can be done. The identity that is granted access in both these cases is the authenticated name (if available) and IP address of the *condor_startd* when the *condor_schedd* initially connects to it to request the claim. It is important that only trusted *condor_startd* s are allowed to publish themselves to the collector or that the *condor_schedd* 's ALLOW_CLIENT setting prevent it from allowing connections to *condor_startd* s that it does not trust to run jobs.
3. When is true, execute-side@matchsession is automatically granted READ access to the *condor_schedd* and DAEMON access to the *condor_shadow*.
4. When is true, then negotiator-side@matchsession is automatically granted NEGOTIATOR access to the *condor_schedd*.

Example of Authorization Security Configuration

An example of the configuration variables for the user-side authorization is derived from the necessary access levels as described in *HTCondor's Security Model*.

```
ALLOW_READ            = *@cs.wisc.edu/*
ALLOW_WRITE           = *@cs.wisc.edu/*.cs.wisc.edu
ALLOW_ADMINISTRATOR   = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_CONFIG          = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_NEGOTIATOR      = condor@cs.wisc.edu/condor.cs.wisc.edu, \
                        condor@cs.wisc.edu/condor2.cs.wisc.edu
ALLOW_DAEMON          = condor@cs.wisc.edu/*.cs.wisc.edu
```

This example configuration authorizes any authenticated user in the cs.wisc.edu domain to carry out a request that requires the READ access level from any machine. Any user in the cs.wisc.edu domain may carry out a request that requires the WRITE access level from any machine in the cs.wisc.edu domain. Only the user called condor-admin may carry out a request that requires the ADMINISTRATOR access level from any machine in the cs.wisc.edu domain. The administrator, logged into any machine within the cs.wisc.edu domain is authorized at the CONFIG access level. Only the negotiator daemon, running as condor on the two central managers are authorized with the NEGOTIATOR access level. And, the last line of the example presumes that there is a user called condor, and that the daemons have all been

started up as this user. It authorizes only programs (which will be the daemons) running as condor to carry out requests that require the DAEMON access level, where the commands originate from any machine in the cs.wisc.edu domain.

Debugging Security Configuration

If the authorization policy denies a network request, an explanation of why the request was denied is printed in the log file of the daemon that denied the request. The line in the log file contains the words PERMISSION DENIED.

To get HTCondor to generate a similar explanation of why requests are accepted, add D_SECURITY to the daemon's debug options (and restart or reconfig the daemon). The line in the log file for these cases will contain the words PERMISSION GRANTED. If you do not want to see a full explanation but just want to see when requests are made, add D_COMMAND to the daemon's debug options.

If the authorization policy makes use of host or domain names, then be aware that HTCondor depends on DNS to map IP addresses to names. The security and accuracy of your DNS service is therefore a requirement. Typos in DNS mappings are an occasional source of unexpected behavior. If the authorization policy is not behaving as expected, carefully compare the names in the policy with the host names HTCondor mentions in the explanations of why requests are granted or denied.

5.9.11 FIPS

As of version 9.0.2, HTCondor is now FIPS compliant when configured to be so. In practice this means that MD5 digests and Blowfish encryption are not used anywhere. To make this easy to configure, we have added a configuration macro, and all you need to add to your config is the following:

```
use security:FIPS
```

This will configure HTCondor to use AES encryption with AES-GCM message digests for all TCP network connections. If you are using UDP for any reason, HTCondor will then fall back to using 3DES for UDP packet encryption because HTCondor does not currently support AES for UDP. The main reasons anyone would be using UDP would be if you had configured a large pool to be supported by Collector trees using UDP, or if you are using Windows (because HTCondor sends signals to daemons on Windows using UDP).

Currently, the use of the High-Availability Daemon (HAD) is not supported when running on a machine that is FIPS compliant.

5.9.12 Security Sessions

To set up and configure secure communications in HTCondor, authentication, encryption, and integrity checks can be used. However, these come at a cost: performing strong authentication can take a significant amount of time, and generating the cryptographic keys for encryption and integrity checks can take a significant amount of processing power.

The HTCondor system makes many network connections between different daemons. If each one of these was to be authenticated, and new keys were generated for each connection, HTCondor would not be able to scale well. Therefore, HTCondor uses the concept of sessions to cache relevant security information for future use and greatly speed up the establishment of secure communications between the various HTCondor daemons.

A new session is established the first time a connection is made from one daemon to another. Each session has a fixed lifetime after which it will expire and a new session will need to be created again. But while a valid session exists, it can be re-used as many times as needed, thereby preventing the need to continuously re-establish secure connections. Each entity of a connection will have access to a session key that proves the identity of the other entity on the opposing side of the connection. This session key is exchanged securely using a strong authentication method, such as Kerberos. Other authentication methods, such as NTSSPI, FS_REMOTE, CLAIMTOBE, and ANONYMOUS, do not support secure key

exchange. An entity listening on the wire may be able to impersonate the client or server in a session that does not use a strong authentication method.

Establishing a secure session requires that either the encryption or the integrity options be enabled. If the encryption capability is enabled, then the session will be restarted using the session key as the encryption key. If integrity capability is enabled, then the check sum includes the session key even though it is not transmitted. Without either of these two methods enabled, it is possible for an attacker to use an open session to make a connection to a daemon and use that connection for nefarious purposes. It is strongly recommended that if you have authentication turned on, you should also turn on integrity and/or encryption.

The configuration parameter will allow a user to set the default level of secure sessions in HTCondor. Like other security settings, the possible values for this parameter can be REQUIRED, PREFERRED, OPTIONAL, or NEVER. If you disable sessions and you have authentication turned on, then most authentication (other than commands like *condor_submit*) will fail because HTCondor requires sessions when you have security turned on. On the other hand, if you are not using strong security in HTCondor, but you are relying on the default host-based security, turning off sessions may be useful in certain situations. These might include debugging problems with the security session management or slightly decreasing the memory consumption of the daemons, which keep track of the sessions in use.

Session lifetimes for specific daemons are already properly configured in the default installation of HTCondor. HTCondor tools such as *condor_q* and *condor_status* create a session that expires after one minute. Theoretically they should not create a session at all, because the session cannot be reused between program invocations, but this is difficult to do in the general case. This allows a very small window of time for any possible attack, and it helps keep the memory footprint of running daemons down, because they are not keeping track of all of the sessions. The session durations may be manually tuned by using macros in the configuration file, but this is not recommended.

5.9.13 Host-Based Security in HTCondor

This section describes the mechanisms for setting up HTCondor's host-based security. This is now an outdated form of implementing security levels for machine access. It remains available and documented for purposes of backward compatibility. If used at the same time as the user-based authorization, the two specifications are merged together.

The host-based security paradigm allows control over which machines can join an HTCondor pool, which machines can find out information about your pool, and which machines within a pool can perform administrative commands. By default, HTCondor is configured to allow anyone to view or join a pool. It is recommended that this parameter is changed to only allow access from machines that you trust.

This section discusses how the host-based security works inside HTCondor. It lists the different levels of access and what parts of HTCondor use which levels. There is a description of how to configure a pool to grant or deny certain levels of access to various machines. Configuration examples and the settings of configuration variables using the *condor_config_val* command complete this section.

Inside the HTCondor daemons or tools that use DaemonCore (see the *DaemonCore* section), most tasks are accomplished by sending commands to another HTCondor daemon. These commands are represented by an integer value to specify which command is being requested, followed by any optional information that the protocol requires at that point (such as a ClassAd, capability string, etc). When the daemons start up, they will register which commands they are willing to accept, what to do with arriving commands, and the access level required for each command. When a command request is received by a daemon, HTCondor identifies the access level required and checks the IP address of the sender to verify that it satisfies the allow/deny settings from the configuration file. If permission is granted, the command request is honored; otherwise, the request will be aborted.

Settings for the access levels in the global configuration file will affect all the machines in the pool. Settings in a local configuration file will only affect the specific machine. The settings for a given machine determine what other hosts can send commands to that machine. If a machine foo is to be given administrator access on machine bar, place foo in bar's configuration file access list (not the other way around).

The following are the various access levels that commands within HTCondor can be registered with:

READ
Machines with READ access can read information from the HTCondor daemons. For example, they can view the status of the pool, see the job queue(s), and view user permissions. READ access does not allow a machine to alter any information, and does not allow job submission. A machine listed with READ permission will be unable join an HTCondor pool; the machine can only view information about the pool.

WRITE
Machines with WRITE access can write information to the HTCondor daemons. Most important for granting a machine with this access is that the machine will be able to join a pool since they are allowed to send ClassAd updates to the central manager. The machine can talk to the other machines in a pool in order to submit or run jobs.

Note: For a machine to join an HTCondor pool, the machine must have both WRITE permission **AND** READ permission. WRITE permission is not enough.

ADMINISTRATOR
Machines with ADMINISTRATOR access are granted additional HTCondor administrator rights to the pool. This includes the ability to change user priorities with the command *condor_userprio*, and the ability to turn HTCondor on and off using *condor_on* and *condor_off*. It is recommended that few machines be granted administrator access in a pool; typically these are the machines that are used by HTCondor and system administrators as their primary workstations, or the machines running as the pool's central manager.

Note: Giving ADMINISTRATOR privileges to a machine grants administrator access for the pool to **ANY USER** on that machine. This includes any users who can run HTCondor jobs on that machine. It is recommended that ADMINISTRATOR access is granted with due diligence.

NEGOTIATOR
This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run.

CONFIG
This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, machines with this level of access are able to change any configuration parameter, except those specified in the condor_config.root configuration file. Therefore, one should exercise extreme caution before granting this level of host-wide access. Because of the implications caused by CONFIG privileges, it is disabled by default for all hosts.

DAEMON
This access level is used for commands that are internal to the operation of HTCondor. An example of this internal operation is when the *condor_startd* daemon sends its ClassAd updates to the *condor_collector* daemon (which may be more specifically controlled by the ADVERTISE_STARTD access level). Authorization at this access level should only be given to hosts that actually run HTCondor in your pool. The DAEMON level of access implies both READ and WRITE access. Any setting for this access level that is not defined will default to the corresponding setting in the WRITE access level.

ADVERTISE_MASTER
This access level is used specifically for commands used to advertise a *condor_master* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level.

ADVERTISE_STARTD
This access level is used specifically for commands used to advertise a *condor_startd* daemon to the collector.

Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level.

ADVERTISE_SCHEDD
This access level is used specifically for commands used to advertise a *condor_schedd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level.

CLIENT
This access level is different from all the others. Whereas all of the other access levels refer to the security policy for accepting connections from others, the CLIENT access level applies when an HTCondor daemon or tool is connecting to some other HTCondor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

ADMINISTRATOR and NEGOTIATOR access default to the central manager machine. CONFIG access is not granted to any machine as its default. These defaults are sufficient for most pools, and should not be changed without a compelling reason.

5.9.14 Examples of Security Configuration

Here is a sample security configuration:

```
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
ALLOW_READ = *
ALLOW_WRITE = *
ALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
ALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
ALLOW_WRITE_COLLECTOR = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_WRITE_STARTD    = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_READ_COLLECTOR  = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_READ_STARTD     = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_CLIENT = *
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine.

For each access level, an ALLOW or a DENY may be added.

- If there is an ALLOW, it means "only allow these machines". No ALLOW means allow anyone.
- If there is a DENY, it means "deny these machines". No DENY means deny nobody.
- If there is both an ALLOW and a DENY, it means allow the machines listed in ALLOW except for the machines listed in DENY.
- Exclusively for the CONFIG access, no ALLOW means allow no one. Note that this is different than the other ALLOW configurations. It is different to enable more stringent security where older configurations are used, since older configuration files would not have a CONFIG configuration entry.

Multiple machine entries in the configuration files may be separated by either a space or a comma. The machines may be listed by

- Individual host names, for example: `condor.cs.wisc.edu`
- Individual IP address, for example: `128.105.67.29`
- IP subnets (use a trailing *), for example: `144.105.*, 128.105.67.*`

- Host names with a wild card * character (only one * is allowed per name), for example: `*.cs.wisc.edu, sol*.cs.wisc.edu`

To resolve an entry that falls into both allow and deny: individual machines have a higher order of precedence than wild card entries, and host names with a wild card have a higher order of precedence than IP subnets. Otherwise, DENY has a higher order of precedence than ALLOW. This is how most people would intuitively expect it to work.

In addition, the above access levels may be specified on a per-daemon basis, instead of machine-wide for all daemons. Do this with the subsystem string (described in *Pre-Defined Macros* on Subsystem Names), which is one of: `STARTD`, `SCHEDD`, `MASTER`, `NEGOTIATOR`, or `COLLECTOR`. For example, to grant different read access for the *condor_schedd*:

```
ALLOW_READ_SCHEDD = <list of machines>
```

Here are more examples of configuration settings. Notice that `ADMINISTRATOR` access is only granted through an `ALLOW` setting to explicitly grant access to a small number of machines. We recommend this.

- Let any machine join the pool. Only the central manager has administrative access.

```
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
```

- Only allow machines at NCSA to join or view the pool. The central manager is the only machine with `ADMINISTRATOR` access.

```
ALLOW_READ = *.ncsa.uiuc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
```

- Only allow machines at NCSA and the U of I Math department join the pool, except do not allow lab machines to do so. Also, do not allow the 177.55 subnet (perhaps this is the dial-in subnet). Allow anyone to view pool statistics. The machine named bigcheese administers the pool (not the central manager).

```
ALLOW_WRITE = *.ncsa.uiuc.edu, *.math.uiuc.edu
DENY_WRITE = lab-*.edu, *.lab.uiuc.edu, 177.55.*
ALLOW_ADMINISTRATOR = bigcheese.ncsa.uiuc.edu
```

- Only allow machines at NCSA and UW-Madison's CS department to view the pool. Only NCSA machines and the machine raven.cs.wisc.edu can join the pool. Note: the machine raven.cs.wisc.edu has the read access it needs through the wild card setting in . This example also shows how to use the continuation character, \, to continue a long list of machines onto multiple lines, making it more readable. This works for all configuration file entries, not just host access entries.

```
ALLOW_READ = *.ncsa.uiuc.edu, *.cs.wisc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu, raven.cs.wisc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                      biggercheese.uiuc.edu
```

- Allow anyone except the military to view the status of the pool, but only let machines at NCSA view the job queues. Only NCSA machines can join the pool. The central manager, bigcheese, and biggercheese can perform most administrative functions. However, only biggercheese can update user priorities.

```
DENY_READ = *.mil
ALLOW_READ_SCHEDD = *.ncsa.uiuc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                      biggercheese.uiuc.edu
ALLOW_ADMINISTRATOR_NEGOTIATOR = biggercheese.uiuc.edu
```

5.9.15 Changing the Security Configuration

A new security feature introduced in HTCondor version 6.3.2 enables more fine-grained control over the configuration settings that can be modified remotely with the *condor_config_val* command. The manual page for *condor_config_val* details how to use *condor_config_val* to modify configuration settings remotely. Since certain configuration attributes can have a large impact on the functioning of the HTCondor system and the security of the machines in an HTCondor pool, it is important to restrict the ability to change attributes remotely.

For each security access level described, the HTCondor administrator can define which configuration settings a host at that access level is allowed to change. Optionally, the administrator can define separate lists of settable attributes for each HTCondor daemon, or the administrator can define one list that is used by all daemons.

For each command that requests a change in configuration setting, HTCondor searches all the different possible security access levels to see which, if any, the request satisfies. (Some hosts can qualify for multiple access levels. For example, any host with `ADMINISTRATOR` permission probably has `WRITE` permission also). Within the qualified access level, HTCondor searches for the list of attributes that may be modified. If the request is covered by the list, the request will be granted. If not covered, the request will be refused.

The default configuration shipped with HTCondor is exceedingly restrictive. HTCondor users or administrators cannot set configuration values from remote hosts with *condor_config_val*. Enabling this feature requires a change to the settings in the configuration file. Use this security feature carefully. Grant access only for attributes which you need to be able to modify in this manner, and grant access only at the most restrictive security level possible.

The most secure use of this feature allows HTCondor users to set attributes in the configuration file which are not used by HTCondor directly. These are custom attributes published by various HTCondor daemons with the setting described in *DaemonCore Configuration File Entries*. It is secure to grant access only to modify attributes that are used by HTCondor to publish information. Granting access to modify settings used to control the behavior of HTCondor is not secure. The goal is to ensure no one can use the power to change configuration attributes to compromise the security of your HTCondor pool.

The control lists are defined by configuration settings that contain `SETTABLE_ATTRS` in their name. The name of the control lists have the following form:

```
<SUBSYS>.SETTABLE_ATTRS_<PERMISSION-LEVEL>
```

The two parts of this name that can vary are the <PERMISSION-LEVEL> and the <SUBSYS>. The <PERMISSION-LEVEL> can be any of the security access levels described earlier in this section. Examples include `WRITE` and `CONFIG`.

The <SUBSYS> is an optional portion of the name. It can be used to define separate rules for which configuration attributes can be set for each kind of HTCondor daemon (for example, `STARTD`, `SCHEDD`, and `MASTER`). There are many configuration settings that can be defined differently for each daemon that use this <SUBSYS> naming convention. See *Pre-Defined Macros* for a list. If there is no daemon-specific value for a given daemon, HTCondor will look for .

Each control list is defined by a comma-separated list of attribute names which should be allowed to be modified. The lists can contain wild cards characters (*).

Some examples of valid definitions of control lists with explanations:

- ```
  SETTABLE_ATTRS_CONFIG = *
  ```

  Grant unlimited access to modify configuration attributes to any request that came from a machine in the `CONFIG` access level. This was the default behavior before HTCondor version 6.3.2.

- ```
  SETTABLE_ATTRS_ADMINISTRATOR = *_DEBUG, MAX_*_LOG
  ```

  Grant access to change any configuration setting that ended with _DEBUG (for example,) and any attribute that matched MAX_*_LOG (for example,) to any host with `ADMINISTRATOR` access.

5.9.16 User Accounts in HTCondor on Unix Platforms

On a Unix system, UIDs (User IDentification numbers) form part of an operating system's tools for maintaining access control. Each executing program has a UID, a unique identifier of a user executing the program. This is also called the real UID. A common situation has one user executing the program owned by another user. Many system commands work this way, with a user (corresponding to a person) executing a program belonging to (owned by) root. Since the program may require privileges that root has which the user does not have, a special bit in the program's protection specification (a setuid bit) allows the program to run with the UID of the program's owner, instead of the user that executes the program. This UID of the program's owner is called an effective UID.

HTCondor works most smoothly when its daemons run as root. The daemons then have the ability to switch their effective UIDs at will. When the daemons run as root, they normally leave their effective UID and GID (Group IDentification) to be those of user and group condor. This allows access to the log files without changing the ownership of the log files. It also allows access to these files when the user condor's home directory resides on an NFS server. root can not normally access NFS files.

If there is no condor user and group on the system, an administrator can specify which UID and GID the HTCondor daemons should use when they do not need root privileges in two ways: either with the `CONDOR_IDS` environment variable or the configuration variable. In either case, the value should be the UID integer, followed by a period, followed by the GID integer. For example, if an HTCondor administrator does not want to create a condor user, and instead wants their HTCondor daemons to run as the daemon user (a common non-root user for system daemons to execute as), the daemon user's UID was 2, and group daemon had a GID of 2, the corresponding setting in the HTCondor configuration file would be `CONDOR_IDS = 2.2`.

On a machine where a job is submitted, the *condor_schedd* daemon changes its effective UID to root such that it has the capability to start up a *condor_shadow* daemon for the job. Before a *condor_shadow* daemon is created, the *condor_schedd* daemon switches back to root, so that it can start up the *condor_shadow* daemon with the (real) UID of the user who submitted the job. Since the *condor_shadow* runs as the owner of the job, all remote system calls are performed under the owner's UID and GID. This ensures that as the job executes, it can access only files that its owner could access if the job were running locally, without HTCondor.

On the machine where the job executes, the job runs either as the submitting user or as user nobody, to help ensure that the job cannot access local resources or do harm. If the matches, and the user exists as the same UID in password files on both the submitting machine and on the execute machine, the job will run as the submitting user. If the user does not exist in the execute machine's password file and is True, then the job will run under the submitting user's UID anyway (as defined in the submitting machine's password file). If is False, and matches, and the user is not in the execute machine's password file, then the job execution attempt will be aborted.

Jobs that run as nobody are low privilege, but can still interfere with each other. To avoid this, you can configure to the value `$(STARTER_SLOT_NAME)` or configure for each slot to define a different username to use for each slot instead of the user nobody. If `NOBODY_SLOT_USER` is configured to be `$(STARTER_SLOT_NAME)` usernames such as `slot1`, `slot2` and `slot1_2` will be used instead of nobody and each slot will use a different name than every other slot.

Running HTCondor as Non-Root

While we strongly recommend starting up the HTCondor daemons as root, we understand that it is not always possible to do so. The main problems of not running HTCondor daemons as root appear when one HTCondor installation is shared by many users on a single machine, or if machines are set up to only execute HTCondor jobs. With a submit-only installation for a single user, there is no need for or benefit from running as root.

The effects of HTCondor of running both with and without root access are classified for each daemon:

condor_startd
: An HTCondor machine set up to execute jobs where the *condor_startd* is not started as root relies on the good will of the HTCondor users to agree to the policy configured for the *condor_startd* to enforce for starting, suspending,

vacating, and killing HTCondor jobs. When the *condor_startd* is started as root, however, these policies may be enforced regardless of malicious users. By running as root, the HTCondor daemons run with a different UID than the HTCondor job. The user's job is started as either the UID of the user who submitted it, or as user nobody, depending on the settings. Therefore, the HTCondor job cannot do anything to the HTCondor daemons. Without starting the daemons as root, all processes started by HTCondor, including the user's job, run with the same UID. Only root can switch UIDs. Therefore, a user's job could kill the *condor_startd* and *condor_starter*. By doing so, the user's job avoids getting suspended or vacated. This is nice for the job, as it obtains unlimited access to the machine, but it is awful for the machine owner or administrator. If there is trust of the users submitting jobs to HTCondor, this might not be a concern. However, to ensure that the policy chosen is enforced by HTCondor, the *condor_startd* should be started as root.

In addition, some system information cannot be obtained without root access on some platforms. As a result, when running without root access, the *condor_startd* must call other programs such as *uptime*, to get this information. This is much less efficient than getting the information directly from the kernel, as is done when running as root. On Linux, this information is available without root access, so it is not a concern on those platforms.

If all of HTCondor cannot be run as root, at least consider installing the *condor_startd* as setuid root. That would solve both problems. Barring that, install it as a setgid sys or kmem program, depending on whatever group has read access to `/dev/kmem` on the system. That would solve the system information problem.

condor_schedd
: The biggest problem with running the *condor_schedd* without root access is that the *condor_shadow* processes which it spawns are stuck with the same UID that the *condor_schedd* has. This requires users to go out of their way to grant write access to user or group that the *condor_schedd* is run as for any files or directories their jobs write or create. Similarly, read access must be granted to their input files.

  Consider installing *condor_submit* as a setgid condor program so that at least the `stdout`, `stderr` and job event log files get created with the right permissions. If *condor_submit* is a setgid program, it will automatically set its umask to 002 and create group-writable files. This way, the simple case of a job that only writes to `stdout` and `stderr` will work. If users have programs that open their own files, they will need to know and set the proper permissions on the directories they submit from.

condor_master
: The *condor_master* spawns both the *condor_startd* and the *condor_schedd*. To have both running as root, have the *condor_master* run as root. This happens automatically if the *condor_master* is started from boot scripts.

condor_negotiator* and *condor_collector
: There is no need to have either of these daemons running as root.

condor_kbdd
: On platforms that need the *condor_kbdd*, the *condor_kbdd* must run as root. If it is started as any other user, it will not work. Consider installing this program as a setuid root binary if the *condor_master* will not be run as root. Without the *condor_kbdd*, the *condor_startd* has no way to monitor USB mouse or keyboard activity, although it will notice keyboard activity on ttys such as xterms and remote logins.

If HTCondor is not run as root, then choose almost any user name. A common choice is to set up and use the condor user; this simplifies the setup, because HTCondor will look for its configuration files in the condor user's directory. If condor is not selected, then the configuration must be placed properly such that HTCondor can find its configuration files.

If users will be submitting jobs as a user different than the user HTCondor is running as (perhaps you are running as the condor user and users are submitting as themselves), then users have to be careful to only have file permissions properly set up to be accessible by the user HTCondor is using. In practice, this means creating world-writable directories for output from HTCondor jobs. This creates a potential security risk, in that any user on the machine where the job is submitted can alter the data, remove it, or do other undesirable things. It is only acceptable in an environment where users can trust other users.

Normally, users without root access who wish to use HTCondor on their machines create a `condor` home directory somewhere within their own accounts and start up the daemons (to run with the UID of the user). As in the case where

the daemons run as user condor, there is no ability to switch UIDs or GIDs. The daemons run as the UID and GID of the user who started them. On a machine where jobs are submitted, the *condor_shadow* daemons all run as this same user. But, if other users are using HTCondor on the machine in this environment, the *condor_shadow* daemons for these other users' jobs execute with the UID of the user who started the daemons. This is a security risk, since the HTCondor job of the other user has access to all the files and directories of the user who started the daemons. Some installations have this level of trust, but others do not. Where this level of trust does not exist, it is best to set up a condor account and group, or to have each user start up their own Personal HTCondor submit installation.

When a machine is an execution site for an HTCondor job, the HTCondor job executes with the UID of the user who started the *condor_startd* daemon. This is also potentially a security risk, which is why we do not recommend starting up the execution site daemons as a regular user. Use either root or a user such as condor that exists only to run HTCondor jobs.

Who Jobs Run As

Under Unix, HTCondor runs jobs as one of

- the user called nobody

  Running jobs as the nobody user is the least preferable. HTCondor uses user nobody if the value of the configuration variable of the submitting and executing machines are different, or if configuration variable is `False`, or if the job ClassAd contains `RunAsOwner=False`.

  When HTCondor cleans up after executing a vanilla universe job, it does the best that it can by deleting all of the processes started by the job. During the life of the job, it also does its best to track the CPU usage of all processes created by the job. There are a variety of mechanisms used by HTCondor to detect all such processes, but, in general, the only foolproof mechanism is for the job to run under a dedicated execution account (as it does under Windows by default). With all other mechanisms, it is possible to fool HTCondor, and leave processes behind after HTCondor has cleaned up. In the case of a shared account, such as the Unix user nobody, it is possible for the job to leave a lurker process lying in wait for the next job run as nobody. The lurker process may prey maliciously on the next nobody user job, wreaking havoc.

  HTCondor could prevent this problem by simply killing all processes run by the nobody user, but this would annoy many system administrators. The nobody user is often used for non-HTCondor system processes. It may also be used by other HTCondor jobs running on the same machine, if it is a multi-processor machine.

- dedicated accounts called slot users set up for the purpose of running HTCondor jobs

  Better than the nobody user will be to create user accounts for HTCondor to use. These can be low-privilege accounts, just as the nobody user is. Create one of these accounts for each job execution slot per computer, so that distinct user names can be used for concurrently running jobs. This prevents malicious or naive behavior from one slot to affect another slot. For a sample machine with two compute slots, create two users that are intended only to be used by HTCondor. As an example, call them cndrusr1 and cndrusr2. Configuration identifies these users with the configuration variable, where <N> is replaced with the slot number. Here is configuration for this example:

  ```
  SLOT1_USER = cndrusr1
  SLOT2_USER = cndrusr2
  ```

  Also tell HTCondor that these accounts are intended only to be used by HTCondor, so HTCondor can kill all the processes belonging to these users upon job completion. The configuration variable is introduced and set to a regular expression that matches the account names just created:

  ```
  DEDICATED_EXECUTE_ACCOUNT_REGEXP = cndrusr[0-9]+
  ```

  Finally, tell HTCondor not to run jobs as the job owner:

```
STARTER_ALLOW_RUNAS_OWNER = False
```

- the user that submitted the jobs

  Four conditions must be set correctly to run jobs as the user that submitted the job.

  1. In the configuration, the value of variable must be `True` on the machine that will run the job. Its default value is `True` on Unix platforms and `False` on Windows platforms.
  2. If the job's ClassAd has the attribute `RunAsOwner`, it must be set to `True`; if unset, the job must be run on a Unix system. This attribute can be set up for all users by adding an attribute to configuration variable . If this were the only attribute to be added to all job ClassAds, it would be set up with

     ```
     SUBMIT_ATTRS = RunAsOwner
     RunAsOwner = True
     ```

  3. The value of configuration variable must be the same for both the *condor_startd* and *condor_schedd* daemons.
  4. The UID_DOMAIN must be trusted. For example, if the *condor_starter* daemon does a reverse DNS lookup on the *condor_schedd* daemon, and finds that the result is not the same as defined for configuration variable , then it is not trusted. To correct this, set in the configuration for the *condor_starter*

     ```
     TRUST_UID_DOMAIN = True
     ```

Notes:

1. Under Windows, HTCondor by default runs jobs under a dynamically created local account that exists for the duration of the job, but it can optionally run the job as the user account that owns the job if is `True` and the job contains `RunAsOwner`=True.

   will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. for details of this command. However, the default behavior in Windows is to run jobs under a dynamically created dedicated execution account, so just using the default behavior is sufficient to avoid problems with lurker processes. See *Executing Jobs as the Submitting User*, and the *condor_store_cred* manual page for details.

2. The *condor_starter* logs a line similar to

   ```
   Tracking process family by login "cndrusr1"
   ```

   when it treats the account as a dedicated account.

Working Directories for Jobs

Every executing process has a notion of its current working directory. This is the directory that acts as the base for all file system access. There are two current working directories for any HTCondor job: one where the job is submitted and a second where the job executes. When a user submits a job, the submit-side current working directory is the same as for the user when the *condor_submit* command is issued. The **initialdir** submit command may change this, thereby allowing different jobs to have different working directories. This is useful when submitting large numbers of jobs. This submit-side current working directory remains unchanged for the entire life of a job. The submit-side current working directory is also the working directory of the *condor_shadow* daemon.

There is also an execute-side current working directory.

5.10 Networking (includes sections on Port Usage and CCB)

This section on network communication in HTCondor discusses which network ports are used, how HTCondor behaves on machines with multiple network interfaces and IP addresses, and how to facilitate functionality in a pool that spans firewalls and private networks.

The security section of the manual contains some information that is relevant to the discussion of network communication which will not be duplicated here, so please see the *Security* section as well.

Firewalls, private networks, and network address translation (NAT) pose special problems for HTCondor. There are currently two main mechanisms for dealing with firewalls within HTCondor:

1. Restrict HTCondor to use a specific range of port numbers, and allow connections through the firewall that use any port within the range.
2. Use HTCondor Connection Brokering (CCB).

Each method has its own advantages and disadvantages, as described below.

5.10.1 Port Usage in HTCondor

IPv4 Port Specification

The general form for IPv4 port specification is

```
<IP:port?param1name=value1&param2name=value2&param3name=value3&...>
```

These parameters and values are URL-encoded. This means any special character is encoded with %, followed by two hexadecimal digits specifying the ASCII value. Special characters are any non-alphanumeric character.

HTCondor currently recognizes the following parameters with an IPv4 port specification:

CCBID
: Provides contact information for forming a CCB connection to a daemon, or a space separated list, if the daemon is registered with more than one CCB server. Each contact information is specified in the form of IP:port#ID. Note that spaces between list items will be URL encoded by `%20`.

PrivNet
: Provides the name of the daemon's private network. This value is specified in the configuration with `PRIVATE_NETWORK_NAME`.

sock
: Provides the name of *condor_shared_port* daemon named socket.

PrivAddr
: Provides the daemon's private address in form of `IP:port`.

Default Port Usage

Every HTCondor daemon listens on a network port for incoming commands. (Using *condor_shared_port*, this port may be shared between multiple daemons.) Most daemons listen on a dynamically assigned port. In order to send a message, HTCondor daemons and tools locate the correct port to use by querying the *condor_collector*, extracting the port number from the ClassAd. One of the attributes included in every daemon's ClassAd is the full IP address and port number upon which the daemon is listening.

To access the *condor_collector* itself, all HTCondor daemons and tools must know the port number where the *condor_collector* is listening. The *condor_collector* is the only daemon with a well-known, fixed port. By default, HTCondor uses port 9618 for the *condor_collector* daemon. However, this port number can be changed (see below).

As an optimization for daemons and tools communicating with another daemon that is running on the same host, each HTCondor daemon can be configured to write its IP address and port number into a well-known file. The file names are controlled using the `<SUBSYS>_ADDRESS_FILE` configuration variables, as described in the *DaemonCore Configuration File Entries* section.

NOTE: In the 6.6 stable series, and HTCondor versions earlier than 6.7.5, the *condor_negotiator* also listened on a fixed, well-known port (the default was 9614). However, beginning with version 6.7.5, the *condor_negotiator* behaves like all other HTCondor daemons, and publishes its own ClassAd to the *condor_collector* which includes the dynamically assigned port the *condor_negotiator* is listening on. All HTCondor tools and daemons that need to communicate with the *condor_negotiator* will either use the or will query the *condor_collector* for the *condor_negotiator* 's ClassAd.

Using a Non Standard, Fixed Port for the *condor_collector*

By default, HTCondor uses port 9618 for the *condor_collector* daemon. To use a different port number for this daemon, the configuration variables that tell HTCondor these communication details are modified. Instead of

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST)
```

the configuration might be

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST):9650
```

If a non standard port is defined, the same value of (including the port) must be used for all machines in the HTCondor pool. Therefore, this setting should be modified in the global configuration file (`condor_config` file), or the value must be duplicated across all configuration files in the pool if a single configuration file is not being shared.

When querying the *condor_collector* for a remote pool that is running on a non standard port, any HTCondor tool that accepts the **-pool** argument can optionally be given a port number. For example:

```
$ condor_status -pool foo.bar.org:1234
```

Using a Dynamically Assigned Port for the *condor_collector*

On single machine pools, it is permitted to configure the *condor_collector* daemon to use a dynamically assigned port, as given out by the operating system. This prevents port conflicts with other services on the same machine. However, a dynamically assigned port is only to be used on single machine HTCondor pools, and only if the configuration variable has also been defined. This mechanism allows all of the HTCondor daemons and tools running on the same machine to find the port upon which the *condor_collector* daemon is listening, even when this port is not defined in the configuration file and is not known in advance.

To enable the *condor_collector* daemon to use a dynamically assigned port, the port number is set to 0 in the variable. The COLLECTOR_ADDRESS_FILE configuration variable must also be defined, as it provides a known file where the IP address and port information will be stored. All HTCondor clients know to look at the information stored in this file. For example:

```
COLLECTOR_HOST = $(CONDOR_HOST):0
COLLECTOR_ADDRESS_FILE = $(LOG)/.collector_address
```

Configuration definition of COLLECTOR_ADDRESS_FILE is in the *DaemonCore Configuration File Entries* section and COLLECTOR_HOST is in the *HTCondor-wide Configuration File Entries* section.

Restricting Port Usage to Operate with Firewalls

If an HTCondor pool is completely behind a firewall, then no special consideration or port usage is needed. However, if there is a firewall between the machines within an HTCondor pool, then configuration variables may be set to force the usage of specific ports, and to utilize a specific range of ports.

By default, HTCondor uses port 9618 for the *condor_collector* daemon, and dynamic (apparently random) ports for everything else. See *Port Usage in HTCondor*, if a dynamically assigned port is desired for the *condor_collector* daemon.

All of the HTCondor daemons on a machine may be configured to share a single port. See the *condor_shared_port Configuration File Macros* section for more information.

The configuration variables and facilitate setting a restricted range of ports that HTCondor will use. This may be useful when some machines are behind a firewall. The configuration macros HIGHPORT and LOWPORT will restrict dynamic ports to the range specified. The configuration variables are fully defined in the *Network-Related Configuration File Entries* section. All of these ports must be greater than 0 and less than 65,536. Note that both HIGHPORT and LOWPORT must be at least 1024 for HTCondor version 6.6.8. In general, use ports greater than 1024, in order to avoid port conflicts with standard services on the machine. Another reason for using ports greater than 1024 is that daemons and tools are often not run as root, and only root may listen to a port lower than 1024. Also, the range must include enough ports that are not in use, or HTCondor cannot work.

The range of ports assigned may be restricted based on incoming (listening) and outgoing (connect) ports with the configuration variables , , , and See the *Network-Related Configuration File Entries* section for complete definitions of these configuration variables. A range of ports lower than 1024 for daemons running as root is appropriate for incoming ports, but not for outgoing ports. The use of ports below 1024 (versus above 1024) has security implications; therefore, it is inappropriate to assign a range that crosses the 1024 boundary.

NOTE: Setting HIGHPORT and LOWPORT will not automatically force the *condor_collector* to bind to a port within the range. The only way to control what port the *condor_collector* uses is by setting the COLLECTOR_HOST (as described above).

The total number of ports needed depends on the size of the pool, the usage of the machines within the pool (which machines run which daemons), and the number of jobs that may execute at one time. Here we discuss how many ports

are used by each participant in the system. This assumes that *condor_shared_port* is not being used. If it is being used, then all daemons can share a single incoming port.

The central manager of the pool needs `5 + number of condor_schedd daemons` ports for outgoing connections and 2 ports for incoming connections for daemon communication.

Each execute machine (those machines running a *condor_startd* daemon) requires `` 5 + (5 * number of slots advertised by that machine)`` ports. By default, the number of slots advertised will equal the number of physical CPUs in that machine.

Submit machines (those machines running a *condor_schedd* daemon) require `` 5 + (5 * MAX_JOBS_RUNNING``) ports. The configuration variable limits (on a per-machine basis, if desired) the maximum number of jobs. Without this configuration macro, the maximum number of jobs that could be simultaneously executing at one time is a function of the number of reachable execute machines.

Also be aware that `HIGHPORT` and `LOWPORT` only impact dynamic port selection used by the HTCondor system, and they do not impact port selection used by jobs submitted to HTCondor. Thus, jobs submitted to HTCondor that may create network connections may not work in a port restricted environment. For this reason, specifying `HIGHPORT` and `LOWPORT` is not going to produce the expected results if a user submits MPI applications to be executed under the parallel universe.

Where desired, a local configuration for machines not behind a firewall can override the usage of `HIGHPORT` and `LOWPORT`, such that the ports used for these machines are not restricted. This can be accomplished by adding the following to the local configuration file of those machines not behind a firewall:

```
HIGHPORT = UNDEFINED
LOWPORT  = UNDEFINED
```

If the maximum number of ports allocated using `HIGHPORT` and `LOWPORT` is too few, socket binding errors of the form

```
failed to bind any port within <$LOWPORT> - <$HIGHPORT>
```

are likely to appear repeatedly in log files.

Multiple Collectors

This section has not yet been written

Port Conflicts

This section has not yet been written

5.10.2 Reducing Port Usage with the *condor_shared_port* Daemon

The *condor_shared_port* is an optional daemon responsible for creating a TCP listener port shared by all of the HTCondor daemons.

The main purpose of the *condor_shared_port* daemon is to reduce the number of ports that must be opened. This is desirable when HTCondor daemons need to be accessible through a firewall. This has a greater security benefit than simply reducing the number of open ports. Without the *condor_shared_port* daemon, HTCondor can use a range of ports, but since some HTCondor daemons are created dynamically, this full range of ports will not be in use by

HTCondor at all times. This implies that other non-HTCondor processes not intended to be exposed to the outside network could unintentionally bind to ports in the range intended for HTCondor, unless additional steps are taken to control access to those ports. While the *condor_shared_port* daemon is running, it is exclusively bound to its port, which means that other non-HTCondor processes cannot accidentally bind to that port.

A second benefit of the *condor_shared_port* daemon is that it helps address the scalability issues of a access point. Without the *condor_shared_port* daemon, more than 2 ephemeral ports per running job are often required, depending on the rate of job completion. There are only 64K ports in total, and most standard Unix installations only allocate a subset of these as ephemeral ports. Therefore, with long running jobs, and with between 11K and 14K simultaneously running jobs, port exhaustion has been observed in typical Linux installations. After increasing the ephemeral port range to its maximum, port exhaustion occurred between 20K and 25K running jobs. Using the *condor_shared_port* daemon dramatically reduces the required number of ephemeral ports on the submit node where the submit node connects directly to the execute node. If the submit node connects via CCB to the execute node, no ports are required per running job; only the one port allocated to the *condor_shared_port* daemon is used.

When CCB is enabled, the *condor_shared_port* daemon registers with the CCB server on behalf of all daemons sharing the port. This means that it is not possible to individually enable or disable CCB connectivity to daemons that are using the shared port; they all effectively share the same setting, and the *condor_shared_port* daemon handles all CCB connection requests on their behalf.

HTCondor's authentication and authorization steps are unchanged by the use of a shared port. Each HTCondor daemon continues to operate according to its configured policy. Requests for connections to the shared port are not authenticated or restricted by the *condor_shared_port* daemon. They are simply passed to the requested daemon, which is then responsible for enforcing the security policy.

When the *condor_master* is configured to use the shared port by setting the configuration variable

```
USE_SHARED_PORT = True
```

the *condor_shared_port* daemon is treated specially. is automatically added to . A command such as *condor_off*, which shuts down all daemons except for the *condor_master*, will also leave the *condor_shared_port* running. This prevents the *condor_master* from getting into a state where it can no longer receive commands.

Also when `USE_SHARED_PORT = True`, the *condor_collector* needs to be configured to use a shared port, so that connections to the shared port that are destined for the *condor_collector* can be forwarded. As an example, the shared port socket name of the *condor_collector* with shared port number 11000 is

```
COLLECTOR_HOST = cm.host.name:11000?sock=collector
```

This example assumes that the socket name used by the *condor_collector* is `collector`, and it runs on `cm.host.name`. This configuration causes the *condor_collector* to automatically choose this socket name. If multiple *condor_collector* daemons are started on the same machine, the socket name can be explicitly set in the daemon's invocation arguments, as in the example:

```
COLLECTOR_ARGS = -sock collector
```

When the *condor_collector* address is a shared port, TCP updates will be automatically used instead of UDP, because the *condor_shared_port* daemon does not work with UDP messages. Under Unix, this means that the *condor_collector* daemon should be configured to have enough file descriptors. See *Using TCP to Send Updates to the condor_collector* for more information on using TCP within HTCondor.

SOAP commands cannot be sent through the *condor_shared_port* daemon. However, a daemon may be configured to open a fixed, non-shared port, in addition to using a shared port. This is done both by setting `USE_SHARED_PORT = True` and by specifying a fixed port for the daemon using `<SUBSYS>_ARGS = -p <portnum>`.

5.10.3 Configuring HTCondor for Machines With Multiple Network Interfaces

HTCondor can run on machines with multiple network interfaces. Starting with HTCondor version 6.7.13 (and therefore all HTCondor 6.8 and more recent versions), new functionality is available that allows even better support for multi-homed machines, using the configuration variable . A multi-homed machine is one that has more than one NIC (Network Interface Card). Further improvements to this new functionality will remove the need for any special configuration in the common case. For now, care must still be given to machines with multiple NICs, even when using this new configuration variable.

Using BIND_ALL_INTERFACES

Machines can be configured such that whenever HTCondor daemons or tools call `bind()`, the daemons or tools use all network interfaces on the machine. This means that outbound connections will always use the appropriate network interface to connect to a remote host, instead of being forced to use an interface that might not have a route to the given destination. Furthermore, sockets upon which a daemon listens for incoming connections will be bound to all network interfaces on the machine. This means that so long as remote clients know the right port, they can use any IP address on the machine and still contact a given HTCondor daemon.

This functionality is on by default. To disable this functionality, the boolean configuration variable is defined and set to `False`:

```
BIND_ALL_INTERFACES = FALSE
```

This functionality has limitations. Here are descriptions of the limitations.

Using all network interfaces does not work with Kerberos.
Every Kerberos ticket contains a specific IP address within it. Authentication over a socket (using Kerberos) requires the socket to also specify that same specific IP address. Use of `BIND_ALL_INTERFACES` causes outbound connections from a multi-homed machine to originate over any of the interfaces. Therefore, the IP address of the outbound connection and the IP address in the Kerberos ticket will not necessarily match, causing the authentication to fail. Sites using Kerberos authentication on multi-homed machines are strongly encouraged not to enable `BIND_ALL_INTERFACES`, at least until HTCondor's Kerberos functionality supports using multiple Kerberos tickets together with finding the right one to match the IP address a given socket is bound to.

There is a potential security risk.
Consider the following example of a security risk. A multi-homed machine is at a network boundary. One interface is on the public Internet, while the other connects to a private network. Both the multi-homed machine and the private network machines comprise an HTCondor pool. If the multi-homed machine enables `BIND_ALL_INTERFACES`, then it is at risk from hackers trying to compromise the security of the pool. Should this multi-homed machine be compromised, the entire pool is vulnerable. Most sites in this situation would run an *sshd* on the multi-homed machine so that remote users who wanted to access the pool could log in securely and use the HTCondor tools directly. In this case, remote clients do not need to use HTCondor tools running on machines in the public network to access the HTCondor daemons on the multi-homed machine. Therefore, there is no reason to have HTCondor daemons listening on ports on the public Internet, causing a potential security threat.

Up to two IP addresses will be advertised.
At present, even though a given HTCondor daemon will be listening to ports on multiple interfaces, each with their own IP address, there is currently no mechanism for that daemon to advertise all of the possible IP addresses where it can be contacted. Therefore, HTCondor clients (other HTCondor daemons or tools) will not necessarily able to locate and communicate with a given daemon running on a multi-homed machine where `BIND_ALL_INTERFACES` has been enabled.

Currently, HTCondor daemons can only advertise two IP addresses in the ClassAd they send to their *condor_collector*. One is the public IP address and the other is the private IP address. HTCondor tools and other daemons that wish to connect to the daemon will use the private IP address if they are configured with the same private network name, and they will use the public IP address otherwise. So, even if the daemon is listening on 3 or more different interfaces, each with a separate IP, the daemon must choose which two IP addresses to advertise so that other daemons and tools can connect to it.

By default, HTCondor advertises the most public IP address available on the machine. The configuration variable can be used to specify the public IP address HTCondor should advertise, and , along with can be used to specify the private IP address to advertise.

Sites that make heavy use of private networks and multi-homed machines should consider if using the HTCondor Connection Broker, CCB, is right for them. More information about CCB and HTCondor can be found in the *HTCondor Connection Brokering (CCB)* section.

Central Manager with Two or More NICs

Often users of HTCondor wish to set up compute farms where there is one machine with two network interface cards (one for the public Internet, and one for the private net). It is convenient to set up the head node as a central manager in most cases and so here are the instructions required to do so.

Setting up the central manager on a machine with more than one NIC can be a little confusing because there are a few external variables that could make the process difficult. One of the biggest mistakes in getting this to work is that either one of the separate interfaces is not active, or the host/domain names associated with the interfaces are incorrectly configured.

Given that the interfaces are up and functioning, and they have good host/domain names associated with them here is how to configure HTCondor:

In this example, `farm-server.farm.org` maps to the private interface. In the central manager's global (to the cluster) configuration file:

```
CONDOR_HOST = farm-server.farm.org
```

In the central manager's local configuration file:

```
NETWORK_INTERFACE = <IP address of farm-server.farm.org>
NEGOTIATOR = $(SBIN)/condor_negotiator
COLLECTOR = $(SBIN)/condor_collector
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, SCHEDD, STARTD
```

Now, if the cluster is set up so that it is possible for a machine name to never have a domain name (for example, there is machine name but no fully qualified domain name in `/etc/hosts`), configure `DEFAULT_DOMAIN_NAME` to be the domain that is to be added on to the end of the host name.

A Client Machine with Multiple Interfaces

If client machine has two or more NICs, then there might be a specific network interface on which the client machine desires to communicate with the rest of the HTCondor pool. In this case, the local configuration file for the client should have

```
NETWORK_INTERFACE = <IP address of desired interface>
```

5.10.4 HTCondor Connection Brokering (CCB)

HTCondor Connection Brokering, or CCB, is a way of allowing HTCondor components to communicate with each other when one side is in a private network or behind a firewall. Specifically, CCB allows communication across a private network boundary in the following scenario: an HTCondor tool or daemon (process A) needs to connect to an HTCondor daemon (process B), but the network does not allow a TCP connection to be created from A to B; it only allows connections from B to A. In this case, B may be configured to register itself with a CCB server that both A and B can connect to. Then when A needs to connect to B, it can send a request to the CCB server, which will instruct B to connect to A so that the two can communicate.

As an example, consider an HTCondor execute node that is within a private network. This execute node's *condor_startd* is process B. This execute node cannot normally run jobs submitted from a machine that is outside of that private network, because bi-directional connectivity between the submit node and the execute node is normally required. However, if both execute and access point can connect to the CCB server, if both are authorized by the CCB server, and if it is possible for the execute node within the private network to connect to the submit node, then it is possible for the submit node to run jobs on the execute node.

To effect this CCB solution, the execute node's *condor_startd* within the private network registers itself with the CCB server by setting the configuration variable . The submit node's *condor_schedd* communicates with the CCB server, requesting that the execute node's *condor_startd* open the TCP connection. The CCB server forwards this request to the execute node's *condor_startd*, which opens the TCP connection. Once the connection is open, bi-directional communication is enabled.

If the location of the execute and submit nodes is reversed with respect to the private network, the same idea applies: the submit node within the private network registers itself with a CCB server, such that when a job is running and the execute node needs to connect back to the submit node (for example, to transfer output files), the execute node can connect by going through CCB to request a connection.

If both A and B are in separate private networks, then CCB alone cannot provide connectivity. However, if an incoming port or port range can be opened in one of the private networks, then the situation becomes equivalent to one of the scenarios described above and CCB can provide bi-directional communication given only one-directional connectivity. See *Port Usage in HTCondor* for information on opening port ranges. Also note that CCB works nicely with *condor_shared_port*.

Any *condor_collector* may be used as a CCB server. There is no requirement that the *condor_collector* acting as the CCB server be the same *condor_collector* that a daemon advertises itself to (as with `COLLECTOR_HOST`). However, this is often a convenient choice.

Example Configuration

This example assumes that there is a pool of machines in a private network that need to be made accessible from the outside, and that the *condor_collector* (and therefore CCB server) used by these machines is accessible from the outside. Accessibility might be achieved by a special firewall rule for the *condor_collector* port, or by being on a dual-homed machine in both networks.

The configuration of variable on machines in the private network causes registration with the CCB server as in the example:

```
CCB_ADDRESS = $(COLLECTOR_HOST)
PRIVATE_NETWORK_NAME = cs.wisc.edu
```

The definition of `PRIVATE_NETWORK_NAME` ensures that all communication between nodes within the private network continues to happen as normal, and without going through the CCB server. The name chosen for `PRIVATE_NETWORK_NAME` should be different from the private network name chosen for any HTCondor installations that will be communicating with this pool.

Under Unix, and with large HTCondor pools, it is also necessary to give the *condor_collector* acting as the CCB server a large enough limit of file descriptors. This may be accomplished with the configuration variable or an equivalent. Each HTCondor process configured to use CCB with `CCB_ADDRESS` requires one persistent TCP connection to the CCB server. A typical execute node requires one connection for the *condor_master*, one for the *condor_startd*, and one for each running job, as represented by a *condor_starter*. A typical access point requires one connection for the *condor_master*, one for the *condor_schedd*, and one for each running job, as represented by a *condor_shadow*. If there will be no administrative commands required to be sent to the *condor_master* from outside of the private network, then CCB may be disabled in the *condor_master* by assigning `MASTER.CCB_ADDRESS` to nothing:

```
MASTER.CCB_ADDRESS =
```

Completing the count of TCP connections in this example: suppose the pool consists of 500 8-slot execute nodes and CCB is not disabled in the configuration of the *condor_master* processes. In this case, the count of needed file descriptors plus some extra for other transient connections to the collector is 500*(1+1+8)=5000. Be generous, and give it twice as many descriptors as needed by CCB alone:

```
COLLECTOR.MAX_FILE_DESCRIPTORS = 10000
```

Security and CCB

The CCB server authorizes all daemons that register themselves with it (using) at the DAEMON authorization level (these are playing the role of process A in the above description). It authorizes all connection requests (from process B) at the READ authorization level. As usual, whether process B authorizes process A to do whatever it is trying to do is up to the security policy for process B; from the HTCondor security model's point of view, it is as if process A connected to process B, even though at the network layer, the reverse is true.

Troubleshooting CCB

Errors registering with CCB or requesting connections via CCB are logged at level `D_ALWAYS` in the debugging log. These errors may be identified by searching for "CCB" in the log message. Command-line tools require the argument **-debug** for this information to be visible. To see details of the CCB protocol add `D_FULLDEBUG` to the debugging options for the particular HTCondor subsystem of interest. Or, add `D_FULLDEBUG` to `ALL_DEBUG` to get extra debugging from all HTCondor components.

A daemon that has successfully registered itself with CCB will advertise this fact in its address in its ClassAd. The ClassAd attribute `MyAddress` will contain information about its `"CCBID"`.

Scalability and CCB

Any number of CCB servers may be used to serve a pool of HTCondor daemons. For example, half of the pool could use one CCB server and half could use another. Or for redundancy, all daemons could use both CCB servers and then CCB connection requests will load-balance across them. Typically, the limit of how many daemons may be registered with a single CCB server depends on the authentication method used by the *condor_collector* for DAEMON-level and READ-level access, and on the amount of memory available to the CCB server. We are not able to provide specific recommendations at this time, but to give a very rough idea, a server class machine should be able to handle CCB service plus normal *condor_collector* service for a pool containing a few thousand slots without much trouble.

5.10.5 Using TCP to Send Updates to the *condor_collector*

TCP sockets are reliable, connection-based sockets that guarantee the delivery of any data sent. However, TCP sockets are fairly expensive to establish, and there is more network overhead involved in sending and receiving messages.

UDP sockets are datagrams, and are not reliable. There is very little overhead in establishing or using a UDP socket, but there is also no guarantee that the data will be delivered. The lack of guaranteed delivery of UDP will negatively affect some pools, particularly ones comprised of machines across a wide area network (WAN) or highly-congested network links, where UDP packets are frequently dropped.

By default, HTCondor daemons will use TCP to send updates to the *condor_collector*, with the exception of the *condor_collector* forwarding updates to any *condor_collector* daemons specified in , where UDP is used. These configuration variables control the protocol used:

UPDATE_COLLECTOR_WITH_TCP
: When set to `False`, the HTCondor daemons will use UDP to update the *condor_collector*, instead of the default TCP. Defaults to `True`.

UPDATE_VIEW_COLLECTOR_WITH_TCP
: When set to `True`, the HTCondor collector will use TCP to forward updates to *condor_collector* daemons specified by `CONDOR_VIEW_HOST`, instead of the default UDP. Defaults to `False`.

TCP_UPDATE_COLLECTORS
: A list of *condor_collector* daemons which will be updated with TCP instead of UDP, when `UPDATE_COLLECTOR_WITH_TCP` or `UPDATE_VIEW_COLLECTOR_WITH_TCP` is set to `False`.

When there are sufficient file descriptors, the *condor_collector* leaves established TCP sockets open, facilitating better performance. Subsequent updates can reuse an already open socket.

Each HTCondor daemon that sends updates to the *condor_collector* will have 1 socket open to it. So, in a pool with N machines, each of them running a *condor_master*, *condor_schedd*, and *condor_startd*, the *condor_collector* would need at least 3*N file descriptors. If the *condor_collector* is also acting as a CCB server, it will require an additional file descriptor for each registered daemon. In the default configuration, the number of file descriptors available to the *condor_collector* is 10240. For very large pools, the number of descriptor can be modified with the configuration:

```
COLLECTOR_MAX_FILE_DESCRIPTORS = 40960
```

If there are insufficient file descriptors for all of the daemons sending updates to the *condor_collector*, a warning will be printed in the *condor_collector* log file. The string `"file descriptor safety level exceeded"` identifies this warning.

5.10.6 Running HTCondor on an IPv6 Network Stack

HTCondor supports using IPv4, IPv6, or both.

To require IPv4, you may set to true; if the machine does not have an interface with an IPv4 address, HTCondor will not start. Likewise, to require IPv6, you may set to true.

If you set to false, HTCondor will not use IPv4, even if it is available; likewise for `ENABLE_IPV6` and IPv6.

The default setting for and is `auto`. If HTCondor does not find an interface with an address of the corresponding protocol, that protocol will not be used. Additionally, if only one of the protocols has a private or public address, the other protocol will be disabled. For instance, a machine with a private IPv4 address and a loopback IPv6 address will only use IPv4; there's no point trying to contact some other machine via IPv6 over a loopback interface.

If both IPv4 and IPv6 networking are enabled, HTCondor runs in mixed mode. In mixed mode, HTCondor daemons have at least one IPv4 address and at least one IPv6 address. Other daemons and the command-line tools choose between these addresses based on which protocols are enabled for them; if both are, they will prefer the first address listed by that daemon.

A daemon may be listening on one, some, or all of its machine's addresses. Daemons may presently list at most two addresses, one IPv6 and one IPv4. Each address is the "most public" address of its protocol; by default, the IPv6 address is listed first. HTCondor selects the "most public" address heuristically.

Nonetheless, there are two cases in which HTCondor may not use an IPv6 address when one is available:

- When given a literal IP address, HTCondor will use that IP address.
- When looking up a host name using DNS, HTCondor will use the first address whose protocol is enabled for the tool or daemon doing the look up.

You may force HTCondor to prefer IPv4 in all three of these situations by setting the macro to true; this is the default. With set, HTCondor daemons will list their "most public" IPv4 address first; prefer the IPv4 address when choosing from another's daemon list; and prefer the IPv4 address when looking up a host name in DNS.

In practice, both an HTCondor pool's central manager and any submit machines within a mixed mode pool must have both IPv4 and IPv6 addresses for both IPv4-only and IPv6-only *condor_startd* daemons to function properly.

IPv6 and Host-Based Security

You may freely intermix IPv6 and IPv4 address literals. You may also specify IPv6 netmasks as a legal IPv6 address followed by a slash followed by the number of bits in the mask; or as the prefix of a legal IPv6 address followed by two colons followed by an asterisk. The latter is entirely equivalent to the former, except that it only allows you to (implicitly) specify mask bits in groups of sixteen. For example, `fe8f:1234::/60` and `fe8f:1234::*` specify the same network mask.

The HTCondor security subsystem resolves names in the ALLOW and DENY lists and uses all of the resulting IP addresses. Thus, to allow or deny IPv6 addresses, the names must have IPv6 DNS entries (AAAA records), or `NO_DNS` must be enabled.

IPv6 Address Literals

When you specify an IPv6 address and a port number simultaneously, you must separate the IPv6 address from the port number by placing square brackets around the address. For instance:

```
COLLECTOR_HOST = [2607:f388:1086:0:21e:68ff:fe0f:6462]:5332
```

If you do not (or may not) specify a port, do not use the square brackets. For instance:

```
NETWORK_INTERFACE = 1234:5678::90ab
```

IPv6 without DNS

When using the configuration variable `NO_DNS` , IPv6 addresses are turned into host names by taking the IPv6 address, changing colons to dashes, and appending `$(DEFAULT_DOMAIN_NAME)`. So,

```
2607:f388:1086:0:21b:24ff:fedf:b520
```

becomes

```
2607-f388-1086-0-21b-24ff-fedf-b520.example.com
```

assuming

```
DEFAULT_DOMAIN_NAME=example.com
```

5.11 DaemonCore

This section is a brief description of DaemonCore. DaemonCore is a library that is shared among most of the HTCondor daemons which provides common functionality. Currently, the following daemons use DaemonCore:

- *condor_master*
- *condor_startd*
- *condor_schedd*
- *condor_collector*
- *condor_negotiator*
- *condor_kbdd*
- *condor_gridmanager*
- *condor_credd*
- *condor_had*
- *condor_replication*
- *condor_transferer*
- *condor_job_router*
- *condor_lease_manager*
- *condor_rooster*
- *condor_shared_port*
- *condor_defrag*
- *condor_c-gahp*
- *condor_c-gahp_worker_thread*
- *condor_dagman*
- *condor_ft-gahp*

- *condor_rooster*
- *condor_shadow*
- *condor_shared_port*
- *condor_transferd*
- *condor_vm-gahp*

Most of DaemonCore's details are not interesting for administrators. However, DaemonCore does provide a uniform interface for the daemons to various Unix signals, and provides a common set of command-line options that can be used to start up each daemon.

5.11.1 DaemonCore and Unix signals

One of the most visible features that DaemonCore provides for administrators is that all daemons which use it behave the same way on certain Unix signals. The signals and the behavior DaemonCore provides are listed below:

SIGHUP
Causes the daemon to reconfigure itself.

SIGTERM
Causes the daemon to gracefully shutdown.

SIGQUIT
Causes the daemon to quickly shutdown.

Exactly what gracefully and quickly means varies from daemon to daemon. For daemons with little or no state (the *condor_kbdd*, *condor_collector* and *condor_negotiator*) there is no difference, and both `SIGTERM` and `SIGQUIT` signals result in the daemon shutting itself down quickly. For the *condor_master*, a graceful shutdown causes the *condor_master* to ask all of its children to perform their own graceful shutdown methods. The quick shutdown causes the *condor_master* to ask all of its children to perform their own quick shutdown methods. In both cases, the *condor_master* exits after all its children have exited. In the *condor_startd*, if the machine is not claimed and running a job, both the `SIGTERM` and `SIGQUIT` signals result in an immediate exit. In the *condor_schedd*, if there are no jobs currently running, there will be no *condor_shadow* processes, and both signals result in an immediate exit. However, with jobs running, a graceful shutdown causes the *condor_schedd* to ask each *condor_shadow* to gracefully vacate the job it is serving, while a quick shutdown results in a hard kill of every *condor_shadow*.

For all daemons, a reconfigure results in the daemon re-reading its configuration file(s), causing any settings that have changed to take effect. See the *Introduction to Configuration* section for full details on what settings are in the configuration files and what they do.

5.11.2 DaemonCore and Command-line Arguments

The second visible feature that DaemonCore provides to administrators is a common set of command-line arguments that all daemons understand. These arguments and what they do are described below:

-a string
Append a period character ('.') concatenated with **string** to the file name of the log for this daemon, as specified in the configuration file.

-b
Causes the daemon to start up in the background. When a DaemonCore process starts up with this option,

it disassociates itself from the terminal and forks itself, so that it runs in the background. This is the default behavior for the *condor_master*. Prior to 8.9.7 it was the default for all HTCondor daemons.

-c filename
Causes the daemon to use the specified **filename** as a full path and file name as its global configuration file. This overrides the `CONDOR_CONFIG` environment variable and the regular locations that HTCondor checks for its configuration file.

-d
Use dynamic directories. The `$(LOG)`, `$(SPOOL)`, and `$(EXECUTE)` directories are all created by the daemon at run time, and they are named by appending the parent's IP address and PID to the value in the configuration file. These values are then inherited by all children of the daemon invoked with this **-d** argument. For the *condor_master*, all HTCondor processes will use the new directories. If a *condor_schedd* is invoked with the *-d* argument, then only the *condor_schedd* daemon and any *condor_shadow* daemons it spawns will use the dynamic directories (named with the *condor_schedd* daemon's PID).

Note that by using a dynamically-created spool directory named by the IP address and PID, upon restarting daemons, jobs submitted to the original *condor_schedd* daemon that were stored in the old spool directory will not be noticed by the new *condor_schedd* daemon, unless you manually specify the old, dynamically-generated `SPOOL` directory path in the configuration of the new *condor_schedd* daemon.

-f
Causes the daemon to start up in the foreground. Instead of forking, the daemon runs in the foreground. Since 8.9.7, this has been the default for all daemons other than the *condor_master*.

NOTE: Before 8.9.7, When the *condor_master* started up daemons, it would do so with the **-f** option, as it has already forked a process for the new daemon. There will be a **-f** in the argument list for all HTCondor daemons that the *condor_master* spawns.

-k filename
For non-Windows operating systems, causes the daemon to read out a PID from the specified **filename**, and send a SIGTERM to that process. The daemon started with this optional argument waits until the daemon it is attempting to kill has exited.

-l directory
Overrides the value of `LOG` as specified in the configuration files. Primarily, this option is used with the *condor_kbdd* when it needs to run as the individual user logged into the machine, instead of running as root. Regular users would not normally have permission to write files into HTCondor's log directory. Using this option, they can override the value of `LOG` and have the *condor_kbdd* write its log file into a directory that the user has permission to write to.

-local-name name
Specify a local name for this instance of the daemon. This local name will be used to look up configuration parameters. The *Configuration File Macros* section contains details on how this local name will be used in the configuration.

-p port
Causes the daemon to bind to the specified port as its command socket. The *condor_master* daemon uses this option to ensure that the *condor_collector* and *condor_negotiator* start up using well-known ports that the rest of HTCondor depends upon them using.

-pidfile filename
Causes the daemon to write out its PID (process id number) to the specified **filename**. This file can be used to help shutdown the daemon without first searching through the output of the Unix *ps* command.

Since daemons run with their current working directory set to the value of `LOG`, if a full path (one that begins with a slash character, `/`) is not specified, the file will be placed in the `LOG` directory.

-q
Quiet output; write less verbose error messages to `stderr` when something goes wrong, and before regular

logging can be initialized.

-r minutes
Causes the daemon to set a timer, upon expiration of which, it sends itself a SIGTERM for graceful shutdown.

-t
Causes the daemon to print out its error message to `stderr` instead of its specified log file. This option forces the **-f** option.

-v
Causes the daemon to print out version information and exit.

5.12 Logging in HTCondor

HTCondor records many types of information in a variety of logs. Administration may require locating and using the contents of a log to debug issues. Listed here are details of the logs, to aid in identification.

5.12.1 Job and Daemon Logs

job event log
The job event log is an optional, chronological list of events that occur as a job runs. The job event log is written on the submit machine. The submit description file for the job requests a job event log with the submit command **log** . The log is created on and remains on the access point. Contents of the log are detailed in the *In the Job Event Log File* section. Examples of events are that the job is running, that the job is placed on hold, or that the job completed.

daemon logs
Each daemon configured to have a log writes events relevant to that daemon. Each event written consists of a timestamp and message. The name of the log file is set by the value of configuration variable <SUBSYS>_LOG , where <SUBSYS> is replaced by the name of the daemon. The log is not permitted to grow without bound; log rotation takes place after a configurable maximum size or length of time is encountered. This maximum is specified by configuration variable MAX_<SUBSYS>_LOG .

Which events are logged for a particular daemon are determined by the value of configuration variable <SUBSYS>_DEBUG . The possible values for <SUBSYS>_DEBUG categorize events, such that it is possible to control the level and quantity of events written to the daemon's log.

Configuration variables that affect daemon logs are

```
MAX_NUM_<SUBSYS>_LOG    TRUNC_<SUBSYS>_LOG_ON_OPEN    <SUBSYS>_LOG_KEEP_OPEN
<SUBSYS>_LOCK   FILE_LOCK_VIA_MUTEX   TOUCH_LOG_INTERVAL   LOGS_USE_TIMESTAMP
LOG_TO_SYSLOG
```

Daemon logs are often investigated to accomplish administrative debugging. *condor_config_val* can be used to determine the location and file name of the daemon log. For example, to display the location of the log for the *condor_collector* daemon, use

```
$ condor_config_val COLLECTOR_LOG
```

job queue log
The job queue log is a transactional representation of the current job queue. If the *condor_schedd* crashes, the job queue can be rebuilt using this log. The file name is set by configuration variable , and defaults to `$(SPOOL)/job_queue.log`.

Within the log, each transaction is identified with an integer value and followed where appropriate with other values relevant to the transaction. To reduce the size of the log and remove any transactions that are no longer relevant, a copy of the log is kept by renaming the log at each time interval defined by configuration variable , and then a new log is written with only current and relevant transactions.

Configuration variables that affect the job queue log are

SCHEDD_BACKUP_SPOOL QUEUE_CLEAN_INTERVAL MAX_JOB_QUEUE_LOG_ROTATIONS

***condor_schedd* audit log**
The optional *condor_schedd* audit log records user-initiated events that modify the job queue, such as invocations of *condor_submit*, *condor_rm*, *condor_hold* and *condor_release*. Each event has a time stamp and a message that describes details of the event.

This log exists to help administrators track the activities of pool users.

The file name is set by configuration variable SCHEDD_AUDIT_LOG .

Configuration variables that affect the audit log are

MAX_SCHEDD_AUDIT_LOG MAX_NUM_SCHEDD_AUDIT_LOG

***condor_shared_port* audit log**
The optional *condor_shared_port* audit log records connections made through the DAEMON_SOCKET_DIR . Each record includes the source address, the socket file name, and the target process's PID, UID, GID, executable path, and command line.

This log exists to help administrators track the activities of pool users.

The file name is set by configuration variable SHARED_PORT_AUDIT_LOG .

Configuration variables that affect the audit log are

MAX_SHARED_PORT_AUDIT_LOG MAX_NUM_SHARED_PORT_AUDIT_LOG

event log
The event log is an optional, chronological list of events that occur for all jobs and all users. The events logged are the same as those that would go into a job event log. The file name is set by configuration variable . The log is created only if this configuration variable is set.

Configuration variables that affect the event log, setting details such as the maximum size to which this log may grow and details of file rotation and locking are

EVENT_LOG_MAX_SIZE EVENT_LOG_MAX_ROTATIONS EVENT_LOG_LOCKING EVENT_LOG_FSYNC
EVENT_LOG_ROTATION_LOCK EVENT_LOG_JOB_AD_INFORMATION_ATTRS EVENT_LOG_USE_XML

accountant log
The accountant log is a transactional representation of the *condor_negotiator* daemon's database of accounting information, which are user priorities. The file name of the accountant log is $(SPOOL)/Accountantnew.log. Within the log, users are identified by username@uid_domain.

To reduce the size and remove information that is no longer relevant, a copy of the log is made when its size hits the number of bytes defined by configuration variable , and then a new log is written in a more compact form.

Administrators can change user priorities kept in this log by using the command line tool *condor_userprio*.

negotiator match log
The negotiator match log is a second daemon log from the *condor_negotiator* daemon. Events written to this log are those with debug level of D_MATCH. The file name is set by configuration variable NEGOTIATOR_MATCH_LOG , and defaults to $(LOG)/MatchLog.

history log
This optional log contains information about all jobs that have been completed. It is written by the *condor_schedd* daemon. The file name is $(SPOOL)/history.

Administrators can change view this historical information by using the command line tool *condor_history*.

Configuration variables that affect the history log, setting details such as the maximum size to which this log may grow are

ENABLE_HISTORY_ROTATION MAX_HISTORY_LOG MAX_HISTORY_ROTATIONS
ROTATE_HISTORY_DAILY ROTATE_HISTORY_MONTHLY

5.12.2 DAGMan Logs

default node log
A job event log of all node jobs within a single DAG. It is used to enforce the dependencies of the DAG.

The file name is set by configuration variable , and the full path name of this file must be unique while any and all submitted DAGs and other jobs from the submit host run. The syntax used in the definition of this configuration variable is different to enable the setting of a unique file name. See the *Configuration File Entries for DAGMan* section for the complete definition.

Configuration variables that affect this log are

DAGMAN_ALWAYS_USE_NODE_LOG

the `.dagman.out` file
A log created or appended to for each DAG submitted with timestamped events and extra information about the configuration applied to the DAG. The name of this log is formed by appending `.dagman.out` to the name of the DAG input file. The file remains after the DAG completes.

This log may be helpful in debugging what has happened in the execution of a DAG, as well as help to determine the final state of the DAG.

Configuration variables that affect this log are

DAGMAN_VERBOSITY DAGMAN_PENDING_REPORT_INTERVAL

the `jobstate.log` file
This optional, machine-readable log enables automated monitoring of DAG. The page *Machine-Readable Event History* details this log.

5.13 Monitoring

Information that the *condor_collector* collects can be used to monitor a pool. The *condor_status* command can be used to display snapshot of the current state of the pool. Monitoring systems can be set up to track the state over time, and they might go further, to alert the system administrator about exceptional conditions.

5.13.1 Ganglia

Support for the Ganglia monitoring system (http://ganglia.info/) is integral to HTCondor. Nagios (http://www.nagios.org/) is often used to provide alerts based on data from the Ganglia monitoring system. The *condor_gangliad* daemon provides an efficient way to take information from an HTCondor pool and supply it to the Ganglia monitoring system.

The *condor_gangliad* gathers up data as specified by its configuration, and it streamlines getting that data to the Ganglia monitoring system. Updates sent to Ganglia are done using the Ganglia shared libraries for efficiency.

If Ganglia is already deployed in the pool, the monitoring of HTCondor is enabled by running the *condor_gangliad* daemon on a single machine within the pool. If the machine chosen is the one running Ganglia's *gmetad*, then the HTCondor configuration consists of adding to the definition of configuration variable on that machine. It may be advantageous to run the *condor_gangliad* daemon on the same machine as is running the *condor_collector* daemon, because on a large pool with many ClassAds, there is likely to be less network traffic. If the *condor_gangliad* daemon is to run on a different machine than the one running Ganglia's *gmetad*, modify configuration variable to get the list of monitored hosts from the master *gmond* program.

If the pool does not use Ganglia, the pool can still be monitored by a separate server running Ganglia.

By default, the *condor_gangliad* will only propagate metrics to hosts that are already monitored by Ganglia. Set configuration variable to `True` to set up a Ganglia host to monitor a pool not monitored by Ganglia or have a heterogeneous pool where some hosts are not monitored. In this case, default graphs that Ganglia provides will not be present. However, the HTCondor metrics will appear.

On large pools, setting configuration variable to `False` will reduce the amount of data sent to Ganglia. The execute node data is the least important to monitor. One can also limit the amount of data by setting configuration variable Be aware that aggregate sums over the entire pool will not be accurate if this variable limits the ClassAds queried.

Metrics to be sent to Ganglia are specified in all files within the directory specified by configuration variable . Each file in the directory is read, and the format within each file is that of New ClassAds. Here is an example of a single metric definition given as a New ClassAd:

```
[
  Name   = "JobsSubmitted";
  Desc   = "Number of jobs submitted";
  Units  = "jobs";
  TargetType = "Scheduler";
]
```

A nice set of default metrics is in file: `$(GANGLIAD_METRICS_CONFIG_DIR)/00_default_metrics`.

Recognized metric attribute names and their use:

Name
: The name of this metric, which corresponds to the ClassAd attribute name. Metrics published for the same machine must have unique names.

Value
: A ClassAd expression that produces the value when evaluated. The default value is the value in the daemon ClassAd of the attribute with the same name as this metric.

Desc
: A brief description of the metric. This string is displayed when the user holds the mouse over the Ganglia graph for the metric.

Verbosity
: The integer verbosity level of this metric. Metrics with a higher verbosity level than that specified by configuration variable will not be published.

TargetType
A string containing a comma-separated list of daemon ClassAd types that this metric monitors. The specified values should match the value of `MyType` of the daemon ClassAd. In addition, there are special values that may be included. "Machine_slot1" may be specified to monitor the machine ClassAd for slot 1 only. This is useful when monitoring machine-wide attributes. The special value "ANY" matches any type of ClassAd.

Requirements
A boolean expression that may restrict how this metric is incorporated. It defaults to `True`, which places no restrictions on the collection of this ClassAd metric.

Title
The graph title used for this metric. The default is the metric name.

Group
A string specifying the name of this metric's group. Metrics are arranged by group within a Ganglia web page. The default is determined by the daemon type. Metrics in different groups must have unique names.

Cluster
A string specifying the cluster name for this metric. The default cluster name is taken from the configuration variable .

Units
A string describing the units of this metric.

Scale
A scaling factor that is multiplied by the value of the `Value` attribute. The scale factor is used when the value is not in the basic unit or a human-interpretable unit. For example, duty cycle is commonly expressed as a percent, but the HTCondor value ranges from 0 to 1. So, duty cycle is scaled by 100. Some metrics are reported in KiB. Scaling by 1024 allows Ganglia to pick the appropriate units, such as number of bytes rather than number of KiB. When scaling by large values, converting to the "float" type is recommended.

Derivative
A boolean value that specifies if Ganglia should graph the derivative of this metric. Ganglia versions prior to 3.4 do not support this.

Type
A string specifying the type of the metric. Possible values are "double", "float", "int32", "uint32", "int16", "uint16", "int8", "uint8", and "string". The default is "string" for string values, the default is "int32" for integer values, the default is "float" for real values, and the default is "int8" for boolean values. Integer values can be coerced to "float" or "double". This is especially important for values stored internally as 64-bit values.

Regex
This string value specifies a regular expression that matches attributes to be monitored by this metric. This is useful for dynamic attributes that cannot be enumerated in advance, because their names depend on dynamic information such as the users who are currently running jobs. When this is specified, one metric per matching attribute is created. The default metric name is the name of the matched attribute, and the default value is the value of that attribute. As usual, the `Value` expression may be used when the raw attribute value needs to be manipulated before publication. However, since the name of the attribute is not known in advance, a special ClassAd attribute in the daemon ClassAd is provided to allow the `Value` expression to refer to it. This special attribute is named `Regex`. Another special feature is the ability to refer to text matched by regular expression groups defined by parentheses within the regular expression. These may be substituted into the values of other string attributes such as `Name` and `Desc`. This is done by putting macros in the string values. "\\1" is replaced by the first group, "\\2" by the second group, and so on.

Aggregate
This string value specifies an aggregation function to apply, instead of publishing individual metrics for each daemon ClassAd. Possible values are "sum", "avg", "max", and "min".

AggregateGroup
When an aggregate function has been specified, this string value specifies which aggregation group the current daemon ClassAd belongs to. The default is the metric `Name`. This feature works like GROUP BY in SQL. The aggregation function produces one result per value of `AggregateGroup`. A single aggregate group would therefore be appropriate for a pool-wide metric. As an example, to publish the sum of an attribute across different types of slot ClassAds, make the metric name an expression that is unique to each type. The default `AggregateGroup` would be set accordingly. Note that the assumption is still that the result is a pool-wide metric, so by default it is associated with the *condor_collector* daemon's host. To group by machine and publish the result into the Ganglia page associated with each machine, make the `AggregateGroup` contain the machine name and override the default `Machine` attribute to be the daemon's machine name, rather than the *condor_collector* daemon's machine name.

Machine
The name of the host associated with this metric. If configuration variable is not specified, the default is taken from the `Machine` attribute of the daemon ClassAd. If the daemon name is of the form name@hostname, this may indicate that there are multiple instances of HTCondor running on the same machine. To avoid the metrics from these instances overwriting each other, the default machine name is set to the daemon name in this case. For aggregate metrics, the default value of `Machine` will be the name of the *condor_collector* host.

IP
A string containing the IP address of the host associated with this metric. If is not specified, the default is extracted from the `MyAddress` attribute of the daemon ClassAd. This value must be unique for each machine published to Ganglia. It need not be a valid IP address. If the value of `Machine` contains an "@" sign, the default IP value will be set to the same value as `Machine` in order to make the IP value unique to each instance of HTCondor running on the same host.

Lifetime
A positive integer value representing the max number of seconds without updating a metric will be kept before deletion. This is represented in ganglia as DMAX. If no Lifetime is defined for a metric then the default value will be set to a calculated value based on the ganglia publish interval with a minimum value set by .

5.13.2 Absent ClassAds

By default, HTCondor assumes that resources are transient: the *condor_collector* will discard ClassAds older than seconds. Its default configuration value is 15 minutes, and as such, the default value for will pass three times before HTCondor forgets about a resource. In some pools, especially those with dedicated resources, this approach may make it unnecessarily difficult to determine what the composition of the pool ought to be, in the sense of knowing which machines would be in the pool, if HTCondor were properly functioning on all of them.

This assumption of transient machines can be modified by the use of absent ClassAds. When a machine ClassAd would otherwise expire, the *condor_collector* evaluates the configuration variable against the machine ClassAd. If `True`, the machine ClassAd will be saved in a persistent manner and be marked as absent; this causes the machine to appear in the output of `condor_status -absent`. When the machine returns to the pool, its first update to the *condor_collector* will invalidate the absent machine ClassAd.

Absent ClassAds, like offline ClassAds, are stored to disk to ensure that they are remembered, even across *condor_collector* crashes. The configuration variable defines the file in which the ClassAds are stored. Absent ClassAds are retained on disk as maintained by the *condor_collector* for a length of time in seconds defined by the configuration

variable . A value of 0 for this variable means that the ClassAds are never discarded, and the default value is thirty days.

Absent ClassAds are only returned by the *condor_collector* and displayed when the **-absent** option to *condor_status* is specified, or when the absent machine ClassAd attribute is mentioned on the *condor_status* command line. This renders absent ClassAds invisible to the rest of the HTCondor infrastructure.

A daemon may inform the *condor_collector* that the daemon's ClassAd should not expire, but should be removed right away; the daemon asks for its ClassAd to be invalidated. It may be useful to place an invalidated ClassAd in the absent state, instead of having it removed as an invalidated ClassAd. An example of a ClassAd that could benefit from being absent is a system with an uninterruptible power supply that shuts down cleanly but unexpectedly as a result of a power outage. To cause all invalidated ClassAds to become absent instead of invalidated, set to `True`. Invalidated ClassAds will instead be treated as if they expired, including when evaluating .

5.13.3 GPUs

HTCondor supports monitoring GPU utilization for NVidia GPUs. This feature is enabled by default if you set `use feature : GPUs` in your configuration file.

Doing so will cause the startd to run the `condor_gpu_utilization` tool. This tool polls the (NVidia) GPU device(s) in the system and records their utilization and memory usage values. At regular intervals, the tool reports these values to the *condor_startd*, assigning them to each device's usage to the slot(s) to which those devices have been assigned.

Please note that `condor_gpu_utilization` can not presently assign GPU utilization directly to HTCondor jobs. As a result, jobs sharing a GPU device, or a GPU device being used by from outside HTCondor, will result in GPU usage and utilization being misreported accordingly.

However, this approach does simplify monitoring for the owner/administrator of the GPUs, because usage is reported by the *condor_startd* in addition to the jobs themselves.

`DeviceGPUsAverageUsage`
: The number of seconds executed by GPUs assigned to this slot, divided by the number of seconds since the startd started up.

`DeviceGPUsMemoryPeakUsage`
: The largest amount of GPU memory used GPUs assigned to this slot, since the startd started up.

5.13.4 Elasticsearch

HTCondor supports pushing *condor_schedd* and *condor_startd* job history ClassAds to Elasticsearch (and other targets) via the *condor_adstash* tool/daemon. *condor_adstash* collects job history ClassAds as specified by its configuration, either querying specified daemons' histories or reading job history ClassAds from a specified file, converts each ClassAd to a JSON document, and pushes each doc to the configured Elasticsearch index. The index is automatically created if it does not exist, and fields are added and configured based on well known job ClassAd attributes. (Custom attributes are also pushed, though always as keyword fields.)

condor_adstash is a Python 3.6+ script that uses the HTCondor *Python Bindings* and the Python Elasticsearch Client, both of which must be available to the system Python 3 installation if using the daemonized version of *condor_adstash*. *condor_adstash* can also be run as a standalone tool (e.g. in a Python 3 virtual environment containing the necessary libraries).

Running *condor_adstash* as a daemon (i.e. under the watch of the *condor_master*) can be enabled by adding `use feature : adstash` to your HTCondor configuration. By default, this configuration will poll all *condor_schedds* that report to the `$(CONDOR_HOST)` *condor_collector* every 20 minutes and push the contents of the job history ClassAds to an Elasticsearch instance running on `localhost` to an index named `htcondor-000001`. Your situation and monitoring needs are likely different! See the `condor_config.local.adstash` example configuration file in the `examples/` directory for detailed information on how to modify your configuration.

If you prefer to run *condor_adstash* in standalone mode, or are curious about other ClassAd sources or targets, see the *condor_adstash* man page for more details.

Configuring a Pool to Report to the HTCondorView Server

For the HTCondorView server to function, configure the existing collector to forward ClassAd updates to it. This configuration is only necessary if the HTCondorView collector is a different collector from the existing *condor_collector* for the pool. All the HTCondor daemons in the pool send their ClassAd updates to the regular *condor_collector*, which in turn will forward them on to the HTCondorView server.

Define the following configuration variable:

```
CONDOR_VIEW_HOST = full.hostname[:portnumber]
```

where full.hostname is the full host name of the machine running the HTCondorView collector. The full host name is optionally followed by a colon and port number. This is only necessary if the HTCondorView collector is configured to use a port number other than the default.

Place this setting in the configuration file used by the existing *condor_collector*. It is acceptable to place it in the global configuration file. The HTCondorView collector will ignore this setting (as it should) as it notices that it is being asked to forward ClassAds to itself.

Once the HTCondorView server is running with this change, send a *condor_reconfig* command to the main *condor_collector* for the change to take effect, so it will begin forwarding updates. A query to the HTCondorView collector will verify that it is working. A query example:

```
$ condor_status -pool condor.view.host[:portnumber]
```

A *condor_collector* may also be configured to report to multiple HTCondorView servers. The configuration variable can be given as a list of HTCondorView servers separated by commas and/or spaces.

The following demonstrates an example configuration for two HTCondorView servers, where both HTCondorView servers (and the *condor_collector*) are running on the same machine, localhost.localdomain:

```
VIEWSERV01 = $(COLLECTOR)
VIEWSERV01_ARGS = -f -p 12345 -local-name VIEWSERV01
VIEWSERV01_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/ViewServerLog01"
VIEWSERV01.POOL_HISTORY_DIR = $(LOCAL_DIR)/poolhist01
VIEWSERV01.KEEP_POOL_HISTORY = TRUE
VIEWSERV01.CONDOR_VIEW_HOST =

VIEWSERV02 = $(COLLECTOR)
VIEWSERV02_ARGS = -f -p 24680 -local-name VIEWSERV02
VIEWSERV02_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/ViewServerLog02"
VIEWSERV02.POOL_HISTORY_DIR = $(LOCAL_DIR)/poolhist02
VIEWSERV02.KEEP_POOL_HISTORY = TRUE
VIEWSERV02.CONDOR_VIEW_HOST =
```

(continues on next page)

(continued from previous page)

```
CONDOR_VIEW_HOST = localhost.localdomain:12345 localhost.localdomain:24680
DAEMON_LIST = $(DAEMON_LIST) VIEWSERV01 VIEWSERV02
```

Note that the value of for VIEWSERV01 and VIEWSERV02 is unset, to prevent them from inheriting the global value of CONDOR_VIEW_HOST and attempting to report to themselves or each other. If the HTCondorView servers are running on different machines where there is no global value for CONDOR_VIEW_HOST, this precaution is not required.

5.14 The High Availability of Daemons

In the case that a key machine no longer functions, HTCondor can be configured such that another machine takes on the key functions. This is called High Availability. While high availability is generally applicable, there are currently two specialized cases for its use: when the central manager (running the *condor_negotiator* and *condor_collector* daemons) becomes unavailable, and when the machine running the *condor_schedd* daemon (maintaining the job queue) becomes unavailable.

5.14.1 High Availability of the Job Queue

For a pool where all jobs are submitted through a single machine in the pool, and there are lots of jobs, this machine becoming nonfunctional means that jobs stop running. The *condor_schedd* daemon maintains the job queue. No job queue due to having a nonfunctional machine implies that no jobs can be run. This situation is worsened by using one machine as the single submission point. For each HTCondor job (taken from the queue) that is executed, a *condor_shadow* process runs on the machine where submitted to handle input/output functionality. If this machine becomes nonfunctional, none of the jobs can continue. The entire pool stops running jobs.

The goal of High Availability in this special case is to transfer the *condor_schedd* daemon to run on another designated machine. Jobs caused to stop without finishing can be restarted from the beginning, or can continue execution using the most recent checkpoint. New jobs can enter the job queue. Without High Availability, the job queue would remain intact, but further progress on jobs would wait until the machine running the *condor_schedd* daemon became available (after fixing whatever caused it to become unavailable).

HTCondor uses its flexible configuration mechanisms to allow the transfer of the *condor_schedd* daemon from one machine to another. The configuration specifies which machines are chosen to run the *condor_schedd* daemon. To prevent multiple *condor_schedd* daemons from running at the same time, a lock (semaphore-like) is held over the job queue. This synchronizes the situation in which control is transferred to a secondary machine, and the primary machine returns to functionality. Configuration variables also determine time intervals at which the lock expires, and periods of time that pass between polling to check for expired locks.

To specify a single machine that would take over, if the machine running the *condor_schedd* daemon stops working, the following additions are made to the local configuration of any and all machines that are able to run the *condor_schedd* daemon (becoming the single pool submission point):

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
VALID_SPOOL_FILES = $(VALID_SPOOL_FILES) SCHEDD.lock
```

Configuration macro MASTER_HA_LIST identifies the *condor_schedd* daemon as the daemon that is to be watched to make sure that it is running. Each machine with this configuration must have access to the lock (the job queue) which synchronizes which single machine does run the *condor_schedd* daemon. This lock and the job queue must both be

located in a shared file space, and is currently specified only with a file URL. The configuration specifies the shared space (SPOOL), and the URL of the lock. *condor_preen* is not currently aware of the lock file and will delete it if it is placed in the SPOOL directory, so be sure to add file SCHEDD.lock to VALID_SPOOL_FILES .

As HTCondor starts on machines that are configured to run the single *condor_schedd* daemon, the *condor_master* daemon of the first machine that looks at (polls) the lock and notices that no lock is held. This implies that no *condor_schedd* daemon is running. This *condor_master* daemon acquires the lock and runs the *condor_schedd* daemon. Other machines with this same capability to run the *condor_schedd* daemon look at (poll) the lock, but do not run the daemon, as the lock is held. The machine running the *condor_schedd* daemon renews the lock periodically.

If the machine running the *condor_schedd* daemon fails to renew the lock (because the machine is not functioning), the lock times out (becomes stale). The lock is released by the *condor_master* daemon if *condor_off* or *condor_off -schedd* is executed, or when the *condor_master* daemon knows that the *condor_schedd* daemon is no longer running. As other machines capable of running the *condor_schedd* daemon look at the lock (poll), one machine will be the first to notice that the lock has timed out or been released. This machine (correctly) interprets this situation as the *condor_schedd* daemon is no longer running. This machine's *condor_master* daemon then acquires the lock and runs the *condor_schedd* daemon.

See the *condor_master Configuration File Macros* section for details relating to the configuration variables used to set timing and polling intervals.

Working with Remote Job Submission

Remote job submission requires identification of the job queue, submitting with a command similar to:

```
$ condor_submit -remote condor@example.com myjob.submit
```

This implies the identification of a single *condor_schedd* daemon, running on a single machine. With the high availability of the job queue, there are multiple *condor_schedd* daemons, of which only one at a time is acting as the single submission point. To make remote submission of jobs work properly, set the configuration variable SCHEDD_NAME in the local configuration to have the same value for each potentially running *condor_schedd* daemon. In addition, the value chosen for the variable SCHEDD_NAME will need to include the at symbol (@), such that HTCondor will not modify the value set for this variable. See the description of MASTER_NAME in the *condor_master Configuration File Macros* section for defaults and composition of valid values for SCHEDD_NAME. As an example, include in each local configuration a value similar to:

```
SCHEDD_NAME = had-schedd@
```

Then, with this sample configuration, the submit command appears as:

```
$ condor_submit -remote had-schedd@  myjob.submit
```

5.14.2 High Availability of the Central Manager

Interaction with Flocking

The HTCondor high availability mechanisms discussed in this section currently do not work well in configurations involving flocking. The individual problems listed listed below interact to make the situation worse. Because of these problems, we advise against the use of flocking to pools with high availability mechanisms enabled.

- The *condor_schedd* has a hard configured list of *condor_collector* and *condor_negotiator* daemons, and does not query redundant collectors to get the current *condor_negotiator*, as it does when communicating with its local pool. As a result, if the default *condor_negotiator* fails, the *condor_schedd* does not learn of the failure, and thus, talk to the new *condor_negotiator*.
- When the *condor_negotiator* is unable to communicate with a *condor_collector*, it utilizes the next *condor_collector* within the list. Unfortunately, it does not start over at the top of the list. When combined with the previous problem, a backup *condor_negotiator* will never get jobs from a flocked *condor_schedd*.

Introduction

The *condor_negotiator* and *condor_collector* daemons are the heart of the HTCondor matchmaking system. The availability of these daemons is critical to an HTCondor pool's functionality. Both daemons usually run on the same machine, most often known as the central manager. The failure of a central manager machine prevents HTCondor from matching new jobs and allocating new resources. High availability of the *condor_negotiator* and *condor_collector* daemons eliminates this problem.

Configuration allows one of multiple machines within the pool to function as the central manager. While there are may be many active *condor_collector* daemons, only a single, active *condor_negotiator* daemon will be running. The machine with the *condor_negotiator* daemon running is the active central manager. The other potential central managers each have a *condor_collector* daemon running; these are the idle central managers.

All submit and execute machines are configured to report to all potential central manager machines.

Each potential central manager machine runs the high availability daemon, *condor_had*. These daemons communicate with each other, constantly monitoring the pool to ensure that one active central manager is available. If the active central manager machine crashes or is shut down, these daemons detect the failure, and they agree on which of the idle central managers is to become the active one. A protocol determines this.

In the case of a network partition, idle *condor_had* daemons within each partition detect (by the lack of communication) a partitioning, and then use the protocol to chose an active central manager. As long as the partition remains, and there exists an idle central manager within the partition, there will be one active central manager within each partition. When the network is repaired, the protocol returns to having one central manager.

Through configuration, a specific central manager machine may act as the primary central manager. While this machine is up and running, it functions as the central manager. After a failure of this primary central manager, another idle central manager becomes the active one. When the primary recovers, it again becomes the central manager. This is a recommended configuration, if one of the central managers is a reliable machine, which is expected to have very short periods of instability. An alternative configuration allows the promoted active central manager (in the case that the central manager fails) to stay active after the failed central manager machine returns.

This high availability mechanism operates by monitoring communication between machines. Note that there is a significant difference in communications between machines when

1. a machine is down
2. a specific daemon (the *condor_had* daemon in this case) is not running, yet the machine is functioning

The high availability mechanism distinguishes between these two, and it operates based only on first (when a central manager machine is down). A lack of executing daemons does not cause the protocol to choose or use a new active central manager.

The central manager machine contains state information, and this includes information about user priorities. The information is kept in a single file, and is used by the central manager machine. Should the primary central manager fail, a pool with high availability enabled would lose this information (and continue operation, but with re-initialized priorities). Therefore, the *condor_replication* daemon exists to replicate this file on all potential central manager machines. This daemon promulgates the file in a way that is safe from error, and more secure than dependence on a shared file system copy.

The *condor_replication* daemon runs on each potential central manager machine as well as on the active central manager machine. There is a unidirectional communication between the *condor_had* daemon and the *condor_replication* daemon on each machine. To properly do its job, the *condor_replication* daemon must transfer state files. When it needs to transfer a file, the *condor_replication* daemons at both the sending and receiving ends of the transfer invoke the *condor_transferer* daemon. These short lived daemons do the task of file transfer and then exit. Do not place `TRANSFERER` into `DAEMON_LIST`, as it is not a daemon that the *condor_master* should invoke or watch over.

Configuration

The high availability of central manager machines is enabled through configuration. It is disabled by default. All machines in a pool must be configured appropriately in order to make the high availability mechanism work. See the *Configuration File Entries Relating to High Availability* section, for definitions of these configuration variables.

The *condor_had* and *condor_replication* daemons use the *condor_shared_port* daemon by default. If you want to use more than one *condor_had* or *condor_replication* daemon with the *condor_shared_port* daemon under the same master, you must configure those additional daemons to use nondefault socket names. (Set the `-sock` option in `<NAME>_ARGS`.) Because the *condor_had* daemon must know the *condor_replication* daemon's address a priori, you will also need to set `<NAME>.REPLICATION_SOCKET_NAME` appropriately.

The stabilization period is the time it takes for the *condor_had* daemons to detect a change in the pool state such as an active central manager failure or network partition, and recover from this change. It may be computed using the following formula:

```
stabilization period = 12 * (number of central managers) *
                           $(HAD_CONNECTION_TIMEOUT)
```

To disable the high availability of central managers mechanism, it is sufficient to remove `HAD`, `REPLICATION`, and `NEGOTIATOR` from the `DAEMON_LIST` configuration variable on all machines, leaving only one *condor_negotiator* in the pool.

To shut down a currently operating high availability mechanism, follow the given steps. All commands must be invoked from a host which has administrative permissions on all central managers. The first three commands kill all *condor_had*, *condor_replication*, and all running *condor_negotiator* daemons. The last command is invoked on the host where the single *condor_negotiator* daemon is to run.

1. condor_off -all -neg
2. condor_off -all -subsystem -replication
3. condor_off -all -subsystem -had
4. condor_on -neg

When configuring *condor_had* to control the *condor_negotiator*, if the default backoff constant value is too small, it can result in a churning of the *condor_negotiator*, especially in cases in which the primary negotiator is unable to run due to misconfiguration. In these cases, the *condor_master* will kill the *condor_had* after the *condor_negotiator* exists, wait a short period, then restart *condor_had*. The *condor_had* will then win the election, so the secondary *condor_negotiator* will be killed, and the primary will be restarted, only to exit again. If this happens too quickly, neither *condor_negotiator* will run long enough to complete a negotiation cycle, resulting in no jobs getting started. Increasing this value via `MASTER_HAD_BACKOFF_CONSTANT` to be larger than a typical negotiation cycle can help solve this problem.

To run a high availability pool without the replication feature, do the following operations:

1. Set the HAD_USE_REPLICATION configuration variable to False, and thus disable the replication on configuration level.
2. Remove REPLICATION from both DAEMON_LIST and DC_DAEMON_LIST in the configuration file.

Sample Configuration

This section provides sample configurations for high availability.

We begin with a sample configuration using shared port, and then include a sample configuration for not using shared port. Both samples relate to the high availability of central managers.

Each sample is split into two parts: the configuration for the central manager machines, and the configuration for the machines that will not be central managers.

The following shared-port configuration is for the central manager machines.

```
## THE FOLLOWING MUST BE IDENTICAL ON ALL CENTRAL MANAGERS

CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
CONDOR_HOST = $(CENTRAL_MANAGER1), $(CENTRAL_MANAGER2)

# Since we're using shared port, we set the port number to the shared
# port daemon's port number.  NOTE: this assumes that each machine in
# the list is using the same port number for shared port.  While this
# will be true by default, if you've changed it in configuration any-
# where, you need to reflect that change here.

HAD_USE_SHARED_PORT = TRUE
HAD_LIST = \
$(CENTRAL_MANAGER1):$(SHARED_PORT_PORT), \
$(CENTRAL_MANAGER2):$(SHARED_PORT_PORT)

REPLICATION_USE_SHARED_PORT = TRUE
REPLICATION_LIST = \
$(CENTRAL_MANAGER1):$(SHARED_PORT_PORT), \
$(CENTRAL_MANAGER2):$(SHARED_PORT_PORT)

# The recommended setting.
HAD_USE_PRIMARY = TRUE

# If you change which daemon(s) you're making highly-available, you must
# change both of these values.
HAD_CONTROLLEE = NEGOTIATOR
MASTER_NEGOTIATOR_CONTROLLER = HAD

## THE FOLLOWING MAY DIFFER BETWEEN CENTRAL MANAGERS

# The daemon list may contain additional entries.
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
```

(continues on next page)

(continued from previous page)

```
# Using replication is optional.
HAD_USE_REPLICATION = TRUE

# This is the default location for the state file.
STATE_FILE = $(SPOOL)/Accountantnew.log

# See note above the length of the negotiation cycle.
MASTER_HAD_BACKOFF_CONSTANT = 360
```

The following shared-port configuration is for the machines which that will not be central managers.

```
CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
CONDOR_HOST = $(CENTRAL_MANAGER1), $(CENTRAL_MANAGER2)
```

The following configuration sets fixed port numbers for the central manager machines.

```
##############################################################################
# A sample configuration file for central managers, to enable the        #
# the high availability  mechanism.                                      #
##############################################################################

##########################################################################
## THE FOLLOWING MUST BE IDENTICAL ON ALL POTENTIAL CENTRAL MANAGERS.   #
##########################################################################
## For simplicity in writing other expressions, define a variable
## for each potential central manager in the pool.
## These are samples.
CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
## A list of all potential central managers in the pool.
CONDOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)

## Define the port number on which the condor_had daemon will
## listen.  The port must match the port number used
## for when defining HAD_LIST.  This port number is
## arbitrary; make sure that there is no port number collision
## with other applications.
HAD_PORT = 51450
HAD_ARGS = -f -p $(HAD_PORT)

## The following macro defines the port number condor_replication will listen
## on on this machine. This port should match the port number specified
## for that replication daemon in the REPLICATION_LIST
## Port number is arbitrary (make sure no collision with other applications)
## This is a sample port number
REPLICATION_PORT = 41450
REPLICATION_ARGS = -p $(REPLICATION_PORT)

## The following list must contain the same addresses in the same order
## as CONDOR_HOST. In addition, for each hostname, it should specify
## the port number of condor_had daemon running on that host.
```

(continues on next page)

(continued from previous page)

```
## The first machine in the list will be the PRIMARY central manager
## machine, in case HAD_USE_PRIMARY is set to true.
HAD_LIST = \
$(CENTRAL_MANAGER1):$(HAD_PORT), \
$(CENTRAL_MANAGER2):$(HAD_PORT)

## The following list must contain the same addresses
## as HAD_LIST. In addition, for each hostname, it should specify
## the port number of condor_replication daemon running on that host.
## This parameter is mandatory and has no default value
REPLICATION_LIST = \
$(CENTRAL_MANAGER1):$(REPLICATION_PORT), \
$(CENTRAL_MANAGER2):$(REPLICATION_PORT)

## The following is the name of the daemon that the HAD controls.
## This must match the name of a daemon in the master's DAEMON_LIST.
## The default is NEGOTIATOR, but can be any daemon that the master
## controls.
HAD_CONTROLLEE = NEGOTIATOR

## HAD connection time.
## Recommended value is 2 if the central managers are on the same subnet.
## Recommended value is 5 if Condor security is enabled.
## Recommended value is 10 if the network is very slow, or
## to reduce the sensitivity of HA daemons to network failures.
HAD_CONNECTION_TIMEOUT = 2

##If true, the first central manager in HAD_LIST is a primary.
HAD_USE_PRIMARY = true


##########################################################################
## THE PARAMETERS BELOW ARE ALLOWED TO BE DIFFERENT ON EACH       #
## CENTRAL MANAGER                                               #
## THESE ARE MASTER SPECIFIC PARAMETERS
##########################################################################


## the master should start at least these four daemons
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION


## Enables/disables the replication feature of HAD daemon
## Default: false
HAD_USE_REPLICATION = true

## Name of the file from the SPOOL directory that will be replicated
## Default: $(SPOOL)/Accountantnew.log
STATE_FILE = $(SPOOL)/Accountantnew.log

## Period of time between two successive awakenings of the replication daemon
## Default: 300
```

(continues on next page)

(continued from previous page)

```
REPLICATION_INTERVAL = 300

## Period of time, in which transferer daemons have to accomplish the
## downloading/uploading process
## Default: 300
MAX_TRANSFER_LIFETIME = 300


## Period of time between two successive sends of classads to the collector by HAD
## Default: 300
HAD_UPDATE_INTERVAL = 300


## The HAD controls the negotiator, and should have a larger
## backoff constant
MASTER_NEGOTIATOR_CONTROLLER = HAD
MASTER_HAD_BACKOFF_CONSTANT = 360
```

The configuration for machines that will not be central managers is identical for the fixed- and shared- port cases.

```
##############################################################################
# Sample configuration relating to high availability for machines        #
# that DO NOT run the condor_had daemon.                                 #
##############################################################################

## For simplicity define a variable for each potential central manager
## in the pool.
CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
## List of all potential central managers in the pool
CONDOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)
```

5.15 Third Party/Delegated file and credential transfer

5.15.1 Enabling the Transfer of Files Specified by a URL

HTCondor permits input files to be directly transferred from a location specified by a URL to the EP; likewise, output files may be transferred to a location specified by a URL. All transfers (both input and output) are accomplished by invoking a **file transfer plugin**: an executable or shell script that handles the task of file transfer.

This URL specification works for most HTCondor job universes, but not grid, local or scheduler. The execute machine directly retrieves the files from their source. Each URL-transferred file, is separately listed in the job submit description file with the command `transfer_input_files`; see *Submitting Jobs Without a Shared File System: HTCondor's File Transfer Mechanism* for details.

For transferring output files, either the entire output sandbox, or a subset of these files, as specified by the submit description file command `transfer_output_files` are transferred to the directory specified by the URL. The URL itself is specified in the separate submit description file command `output_destination`; see *Submitting Jobs Without a Shared File System: HTCondor's File Transfer Mechanism* for details. The plug-in is invoked once for each output file to be transferred.

Configuration identifies the availability of the one or more plug-in(s). The plug-ins must be installed and available on every execute machine that may run a job which might specify a URL, for either direction.

URL transfers are enabled by default in the configuration of execute machines. To Disable URL transfers, set

```
ENABLE_URL_TRANSFERS = FALSE
```

A comma separated list giving the absolute path and name of all available plug-ins is specified as in the example:

```
FILETRANSFER_PLUGINS = /opt/condor/plugins/wget-plugin, \
                       /opt/condor/plugins/hdfs-plugin, \
                       /opt/condor/plugins/custom-plugin
```

The *condor_starter* invokes all listed plug-ins to determine their capabilities. Each may handle one or more protocols (scheme names). The plug-in's response to invocation identifies which protocols it can handle. When a URL transfer is specified by a job, the *condor_starter* invokes the proper one to do the transfer. If more than one plugin is capable of handling a particular protocol, then the last one within the list given by is used.

HTCondor assumes that all plug-ins will respond in specific ways. To determine the capabilities of the plug-ins as to which protocols they handle, the *condor_starter* daemon invokes each plug-in giving it the command line argument `-classad`. In response to invocation with this command line argument, the plug-in must respond with an output of four ClassAd attributes. The first three are fixed:

```
MultipleFileSupport = true
PluginVersion = "0.1"
PluginType = "FileTransfer"
```

The fourth ClassAd attribute is `SupportedMethods`. This attribute is a string containing a comma separated list of the protocols that the plug-in handles. So, for example

```
SupportedMethods = "http,ftp,file"
```

would identify that the three protocols described by http, ftp, and file are supported. These strings will match the protocol specification as given within a URL in a `transfer_input_files` command or within a URL in an `output_destination` command in a submit description file for a job.

When a job specifies a URL transfer, the plug-in is invoked, without the command line argument `-classad`. It will instead be given two other command line arguments. For the transfer of input file(s), the first will be the URL of the file to retrieve and the second will be the absolute path identifying where to place the transferred file. For the transfer of output file(s), the first will be the absolute path on the local machine of the file to transfer, and the second will be the URL of the directory and file name at the destination.

The plug-in is expected to do the transfer, exiting with status 0 if the transfer was successful, and a non-zero status if the transfer was not successful. When not successful, the job is placed on hold, and the job ClassAd attribute `HoldReason` will be set as appropriate for the job. The job ClassAd attribute `HoldReasonSubCode` will be set to the exit status of the plug-in.

As an example of the transfer of a subset of output files, assume that the submit description file contains

```
output_destination = url://server/some/directory/
transfer_output_files = foo, bar, qux
```

HTCondor invokes the plug-in that handles the `url` protocol with input classads describing all the files to be transferred and their destinations. The directory delimiter (/ on Unix, and \ on Windows) is appended to the destination URL, such that the input will look like the following:

```
[ LocalFileName = "/path/to/local/copy/of/foo"; Url = "url://server/some/directory//foo"␣
↪]
[ LocalFileName = "/path/to/local/copy/of/bar"; Url = "url://server/some/directory//bar"␣
↪]
[ LocalFileName = "/path/to/local/copy/of/qux"; Url = "url://server/some/directory//qux"␣
↪]
```

HTCondor also expects the plugin to exit with one of the following standardized exit codes:

- **0**: Transfer successful
- **Any other value**: Transfer failed

Custom File Transfer Plugins

This functionality is not limited to a predefined set of protocols or plugins. New ones can be invented. As an invented example, the `zkm` transfer type writes random bytes to a file. The plug-in that handles `zkm` transfers would respond to invocation with the `-classad` command line argument with:

```
MultipleFileSupport = true
PluginVersion = "0.1"
PluginType = "FileTransfer"
SupportedMethods = "zkm"
```

And, then when a job requested that this plug-in be invoked, for the invented example:

```
transfer_input_files = zkm://128/r-data
```

the plug-in will be invoked with a first command line argument of `zkm://128/r-data` and a second command line argument giving the full path along with the file name `r-data` as the location for the plug-in to write 128 bytes of random data.

By default, HTCondor includes plugins for standard file protocols `http://...`, `https://...` and `ftp://...`. Additionally, URL plugins exist for transferring files to/from Box.com accounts (`box://...`), Google Drive accounts (`gdrive://...`), OSDF accounts (`osdf://...`), Stash accounts (`stash://...`), and Microsoft OneDrive accounts (`onedrive://...`). These plugins require users to have obtained OAuth2 credentials for the relevant service(s) before they can be used. See *Enabling the Fetching and Use of OAuth2 Credentials* for how to enable users to fetch OAuth2 credentials.

An example template for a file transfer plugin is available in our source repository under /src/condor_examples/filetransfer_example_plugin.py. This provides most of the functionality required in the plugin, except for the transfer logic itself, which is clearly indicated in the comments.

Sending File Transfer Plugins With Your Job

You can also use custom protocols on machines that do not have the necessary plugin installed. This is achieved by sending the file transfer plugin along with your job, using the `transfer_plugins` submit attribute described on the *condor_submit* man page.

Assume you want to transfer some URLs that use the `custommethod://` protocol, and you also have a plugin script called `custommethod_plugin.py` that knows how to handle these URLs. Since this plugin is not available on any of the execution points in your pool, you can send it along with your job by including the following in the submit file:

```
transfer_plugins = custommethod=custommethod_plugin.py
transfer_output_files = custommethod://path/to/file1, custommethod://path/to/file2
```

When the job arrives at an exeuction point, it will know to use the plugin script provided to transfer these URLs. If your `custommethod://` protocol is already supported at your execution point, the plugin provided in your submit file will take precedence.

5.15.2 Enabling the Transfer of Public Input Files over HTTP

Another option for transferring files over HTTP is for users to specify a list of public input files. These are specified in the submit file as follows:

```
public_input_files = file1,file2,file3
```

HTCondor will automatically convert these files into URLs and transfer them over HTTP using plug-ins. The advantage to this approach is that system administrators can leverage Squid caches or load-balancing infrastructure, resulting in improved performance. This also allows us to gather statistics about file transfers that were not previously available.

When a user submits a job with public input files, HTCondor generates a hash link for each file in the root directory for the web server. Each of these links points back to the original file on local disk. Next, HTCondor replaces the names of the files in the submit job with web links to their hashes. These get sent to the execute node, which downloads the files using our curl_plugin tool, and are then remapped back to their original names.

In the event of any errors or configuration problems, HTCondor will fall back to a regular (non-HTTP) file transfer.

To enable HTTP public file transfers, a system administrator must perform several steps as described below.

Install a web service for public input files

An HTTP service must be installed and configured on the submit node. Any regular web server software such as Apache (https://httpd.apache.org/) or nginx (https://nginx.org) will do. The submit node must be running a Linux system.

Configuration knobs for public input files

Several knobs must be set and configured correctly for this functionality to work:

- : Must be set to true (default: false) : The full web address (hostname + port) where your web server is serving files (default: 127.0.0.1:8080) : Absolute path to the local directory where the web service is serving files from.
- : User security level used to write links to the directory specified by HTTP_PUBLIC_FILES_ROOT_DIR. There are three valid options for this knob:

  1. **<user>**: Links will be written as user who submitted the job.
  2. **<condor>**: Links will be written as user running condor daemons. By default this is the user condor unless you have changed this by setting the configuration parameter CONDOR_IDS.
  3. **<%username%>**: Links will be written as the user %username% (ie. httpd, nobody) If using this option, make sure the directory is writable by this particular user.

  The default setting is <condor>.

Additional HTTP infrastructure for public input files

The main advantage of using HTTP for file transfers is that system administrators can use additional infrastructure (such as Squid caching) to improve file transfer performance. This is outside the scope of the HTCondor configuration but is still worth mentioning here. When curl_plugin is invoked, it checks the environment variable http_proxy for a proxy server address; by setting this appropriately on execute nodes, a system can dramatically improve transfer speeds for commonly used files.

5.15.3 Enabling the Fetching and Use of OAuth2 Credentials

HTCondor supports two distinct methods for using OAuth2 credentials. One uses its own native OAuth client and credential monitor, and one uses a separate Hashicorp Vault server as the OAuth client and secure refresh token storage. Each method uses a separate credmon implementation and rpm and have their own advantages and disadvantages.

If the native OAuth client is used with a remote token issuer, then each time a new refresh token is needed the user has to re-authorize it through a web browser. An hour after all jobs of a user are stopped (by default), the refresh tokens are deleted. The resulting access tokens are only available inside HTCondor jobs.

If on the other hand a Vault server is used as the OAuth client, it stores the refresh token long term (typically about a month since last use) for multiple use cases. It can be used both by multiple HTCondor access points and by other client commands that need access tokens. Submit machines keep a medium term vault token (typically about a week) so at most users have to authorize in their web browser once a week. If Kerberos is also available, new vault tokens can be obtained automatically without any user intervention. The HTCondor vault credmon also stores a longer lived vault token for use as long as jobs might run.

Using the native OAuth client

HTCondor can be configured to allow users to request and securely store credentials from most OAuth2 service providers. Users' jobs can then request these credentials to be securely transferred to job sandboxes, where they can be used by file transfer plugins or be accessed by the users' executable(s).

There are three steps to fully setting up HTCondor to enable users to be able to request credentials from OAuth2 services:

1. Enabling the *condor_credd* and *condor_credmon_oauth* daemons,
2. Optionally enabling the companion OAuth2 credmon WSGI application, and
3. Setting up API clients and related configuration.

First, to enable the *condor_credd* and *condor_credmon_oauth* daemons, the easiest way is to install the `condor-credmon-oauth` rpm. This installs the *condor_credmon_oauth* daemon and enables both it and *condor_credd* with reasonable defaults via the `use feature: oauth` configuration template.

Second, a token issuer, an HTTPS-enabled web server running on the submit machine needs to be configured to execute its wsgi script as the user `condor`. An example configuration is available at the path found with `rpm -ql condor-credmon-oauth|grep "condor_credmon_oauth\.conf"` which you can copy to an apache webserver's configuration directory.

Third, for each OAuth2 service that one wishes to configure, an OAuth2 client application should be registered for each access point on each service's API console. For example, for Box.com, a client can be registered by logging in to https://app.box.com/developers/console, creating a new "Custom App", and selecting "Standard OAuth 2.0 (User Authentication)."

For each client, store the client ID in the HTCondor configuration under . Store the client secret in a file only readable by root, then point to it using . For our Box.com example, this might look like:

```
BOX_CLIENT_ID = ex4mpl3cl13nt1d
BOX_CLIENT_SECRET_FILE = /etc/condor/.secrets/box_client_secret
```

```
# ls -l /etc/condor/.secrets/box_client_secret
-r-------- 1 root root 33 Jan  1 10:10 /etc/condor/.secrets/box_client_secret
# cat /etc/condor/.secrets/box_client_secret
EXAmpL3ClI3NtS3cREt
```

Each service will need to redirect users back to a known URL on the access point after each user has approved access to their credentials. For example, Box.com asks for the "OAuth 2.0 Redirect URI." This should be set to match such that the user is returned to `https://<submit_hostname>/<return_url_suffix>`. The return URL suffix should be composed using the directory where the WSGI application is running, the subdirectory `return/`, and then the name of the OAuth2 service. For our Box.com example, if running the WSGI application at the root of the webserver (`/`), this should be `BOX_RETURN_URL_SUFFIX = /return/box`.

The *condor_credmon_oauth* and its companion WSGI application need to know where to send users to fetch their initial credentials and where to send API requests to refresh these credentials. Some well known service providers (`condor_config_val -dump TOKEN_URL`) already have their authorization and token URLs predefined in the default HTCondor config. Other service providers will require searching through API documentation to find these URLs, which then must be added to the HTCondor configuration. For example, if you search the Box.com API documentation, you should find the following authorization and token URLs, and these URLs could be added them to the HTCondor config as below:

```
BOX_AUTHORIZATION_URL = https://account.box.com/api/oauth2/authorize
BOX_TOKEN_URL = https://api.box.com/oauth2/token
```

After configuring OAuth2 clients, make sure users know which names (`<OAuth2ServiceName>s`) have been configured so that they know what they should put under `use_oauth_services` in their job submit files.

Using Vault as the OAuth client

To instead configure HTCondor to use Vault as the OAuth client, install the `condor-credmon-vault` rpm. Also install the htgettoken (https://github.com/fermitools/htgettoken) rpm on the access point. Additionally, on the access point set the configuration option to `-a <vault.name>` where <vault.name> is the fully qualified domain name of the Vault machine. *condor_submit* users will then be able to select the oauth services that are defined on the Vault server. See the htvault-config (https://github.com/fermitools/htvault-config) documentation to see how to set up and configure the Vault server.

5.16 Setting Up the Docker Universe

5.16.1 The Docker Universe

The execution of a docker universe job causes the instantiation of a Docker container on an execute host.

The docker universe job is mapped to a vanilla universe job, and the submit description file must specify the submit command **docker_image** to identify the Docker image. The job's `requirement` ClassAd attribute is automatically appended, such that the job will only match with an execute machine that has Docker installed.

The Docker service must be pre-installed on each execute machine that can execute a docker universe job. Upon start up of the *condor_startd* daemon, the capability of the execute machine to run docker universe jobs is probed, and the machine ClassAd attribute `HasDocker` is advertised for a machine that is capable of running Docker universe jobs.

When a docker universe job is matched with a Docker-capable execute machine, HTCondor invokes the Docker CLI to instantiate the image-specific container. The job's scratch directory tree is mounted as a Docker volume. When the job completes, is put on hold, or is evicted, the container is removed.

An administrator of a machine can optionally make additional directories on the host machine readable and writable by a running container. To do this, the admin must first give an HTCondor name to each directory with the DOCKER_VOLUMES parameter. Then, each volume must be configured with the path on the host OS with the DOCKER_VOLUME_DIR_XXX parameter. Finally, the parameter DOCKER_MOUNT_VOLUMES tells HTCondor which of these directories to always mount onto containers running on this machine.

For example,

```
DOCKER_VOLUMES = SOME_DIR, ANOTHER_DIR
DOCKER_VOLUME_DIR_SOME_DIR = /path1
DOCKER_VOLUME_DIR_ANOTHER_DIR = /path/to/no2
DOCKER_MOUNT_VOLUMES = SOME_DIR, ANOTHER_DIR
```

The *condor_startd* will advertise which docker volumes it has available for mounting with the machine attributes HasDockerVolumeSOME_NAME = true so that jobs can match to machines with volumes they need.

Optionally, if the directory name is two directories, separated by a colon, the first directory is the name on the host machine, and the second is the value inside the container. If a ":ro" is specified after the second directory name, the volume will be mounted read-only inside the container.

These directories will be bind-mounted unconditionally inside the container. If an administrator wants to bind mount a directory only for some jobs, perhaps only those submitted by some trusted user, the setting may be used. This is a class ad expression, evaluated in the context of the job ad and the machine ad. Only when it evaluted to TRUE, is the volume mounted. Extending the above example,

```
DOCKER_VOLUMES = SOME_DIR, ANOTHER_DIR
DOCKER_VOLUME_DIR_SOME_DIR = /path1
DOCKER_VOLUME_DIR_SOME_DIR_MOUNT_IF = WantSomeDirMounted && Owner == "smith"
DOCKER_VOLUME_DIR_ANOTHER_DIR = /path/to/no2
DOCKER_MOUNT_VOLUMES = SOME_DIR, ANOTHER_DIR
```

In this case, the directory /path1 will get mounted inside the container only for jobs owned by user "smith", and who set +WantSomeDirMounted = true in their submit file.

In addition to installing the Docker service, the single configuration variable must be set. It defines the location of the Docker CLI and can also specify that the *condor_starter* daemon has been given a password-less sudo permission to start the container as root. Details of the configuration variable are in the *condor_startd Configuration File Macros* section.

Docker must be installed as root by following these steps on an Enterprise Linux machine.

1. Acquire and install the docker-engine community edition by following the installations instructions from docker.com
2. Set up the groups:

   ```
   $ usermod -aG docker condor
   ```

3. Invoke the docker software:

   ```
   $ systemctl start docker
   $ systemctl enable docker
   ```

4. Reconfigure the execute machine, such that it can set the machine ClassAd attribute `HasDocker`:

```
$ condor_reconfig
```

5. Check that the execute machine properly advertises that it is docker-capable with:

```
$ condor_status -l | grep -i docker
```

The output of this command line for a correctly-installed and docker-capable execute host will be similar to

```
HasDocker = true
DockerVersion = "Docker Version 1.6.0, build xxxxx/1.6.0"
```

By default, HTCondor will keep the 8 most recently used Docker images on the local machine. This number may be controlled with the configuration variable , to increase or decrease the number of images, and the corresponding disk space, used by Docker.

By default, Docker containers will be run with all rootly capabilities dropped, and with setuid and setgid binaries disabled, for security reasons. If you need to run containers with root privilege, you may set the configuration parameter to an expression that evaluates to false. This expression is evaluted in the context of the machine ad (my) and the job ad (target).

Docker support an enormous number of command line options when creating containers. While HTCondor tries to map as many useful options from submit files and machine descriptions to command line options, an administrator may want additional options passed to the docker container create command. To do so, the parameter can be set, and condor will append these to the docker container create command.

Docker universe jobs may fail to start on certain Linux machines when SELinux is enabled. The symptom is a permission denied error when reading or executing from the condor scratch directory. To fix this problem, an administrator will need to run the following command as root on the execute directories for all the startd machines:

```
$ chcon -Rt svirt_sandbox_file_t /var/lib/condor/execute
```

All docker universe jobs can request either host-based networking or no networking at all. The latter might be for security reasons. If the worker node administrator has defined additional custom docker networks, perhaps a VPN or other custom type, those networks can be defined for HTCondor jobs to opt into with the docker_network_type submit command. Simple set

```
DOCKER_NETWORKS = some_virtual_network, another_network
```

And these two networks will be advertised by the startd, and jobs that request these network type will only match to machines that support it. Note that HTCondor cannot test the validity of these networks, and merely trusts that the administrator has correctly configured them.

To deal with a potentially user influencing option, there is an optional knob that can be configured to adapt the `--shm-size` Docker container create argument taking the machine's and job's classAds into account. Exemplary, setting the `/dev/shm` size to half the requested memory is achieved by:

```
DOCKER_SHM_SIZE = Memory * 1024 * 1024 / 2
```

or, using a user provided value `DevShmSize` if available and within the requested memory limit:

```
DOCKER_SHM_SIZE = ifThenElse(DevShmSize isnt Undefined && isInteger(DevShmSize) &&␣
↪int(DevShmSize) <= (Memory * 1024 * 1024), int(DevShmSize), 2 * 1024 * 1024 * 1024)
```

Note: `Memory` is in MB, thus it needs to be scaled to bytes.

5.17 Apptainer/Singularity Support

Singularity (https://sylabs.io/singularity/) is a container runtime system popular in scientific and HPC communities. Apptainer (https://apptainer.org) is an open source fork of Singularity that is API and CLI compatible with singularity. Everything in this document that pertains to Singularity also is true for the Apptainer container runtime. HTCondor can run jobs inside Singularity containers either in a transparent way, where the job does not know that it is being contained, or, the HTCondor administrator can configure the HTCondor startd so that a job can opt into running inside a container. This allows the operating system that the job sees to be different than the one on the host system, and provides more isolation between processes running in one job and another.

The decision to run a job inside Singularity ultimately resides on the worker node, although it can delegate that to the job.

By default, jobs will not be run in Singularity.

For Singularity to work, the administrator must install Singularity on the worker node. The HTCondor startd will detect this installation at startup. When it detects a usable installation, it will advertise two attributes in the slot ad:

```
HasSingularity = true
SingularityVersion = "singularity version 3.7.0-1.el7"
```

If the detected Singularity installation fails to run test containers at startd startup, `HasSingularity` will be set to `false`, and the slot ad attribute `SingularityOfflineReason` will contain an error string.

HTCondor will run a job under Singularity when the startd configuration knob evaluates to true. This is evaluated in the context of the slot ad and the job ad. If it evaluates to false or undefined, the job will run as normal, without singularity.

When evaluates to true, a second HTCondor knob is required to name the singularity image that must be run, . This also is evaluated in the context of the machine and the job ad, and must evaluate to a string. This image name is passed to the singularity exec command, and can be any valid value for a singularity image name. So, it may be a path to file on a local file system that contains an singularity image, in any format that singularity supports. It may be a string that begins with `docker://`, and refer to an image located on docker hub, or other repository. It can begin with `http://`, and refer to an image to be fetched from an HTTP server. In this case, singularity will fetch the image into the job's scratch directory, convert it to a .sif file and run it from there. Note this may require the job to request more disk space that it otherwise would need. It can be a relative path, in which case it refers to a file in the scratch directory, so that the image can be transferred by HTCondor's file transfer mechanism.

Here's the simplest possible configuration file. It will force all jobs on this machine to run under Singularity, and to use an image that it located in the file system in the path `/cvfms/cernvm-prod.cern.ch/cvm3`:

```
# Forces _all_ jobs to run inside singularity.
SINGULARITY_JOB = true

# Forces all jobs to use the CernVM-based image.
SINGULARITY_IMAGE_EXPR = "/cvmfs/cernvm-prod.cern.ch/cvm3"
```

Another common configuration is to allow the job to select whether to run under Singularity, and if so, which image to use. This looks like:

```
SINGULARITY_JOB = !isUndefined(TARGET.SingularityImage)
SINGULARITY_IMAGE_EXPR = TARGET.SingularityImage
```

Then, users would add the following to their submit file (note the quoting):

```
+SingularityImage = "/cvmfs/cernvm-prod.cern.ch/cvm3"
```

or maybe

```
+SingularityImage = "docker://ubuntu:20"
```

By default, singularity will bind mount the scratch directory that contains transferred input files, working files, and other per-job information into the container, and make this the initial working directory of the job. Thus, file transfer for singularity jobs works just like with vanilla universe jobs. Any new files the job writes to this directory will be copied back to the submit node, just like any other sandbox, subject to transfer_output_files, as in vanilla universe.

Assuming singularity is configured on the startd as described above, A complete submit file that uses singularity might look like

```
executable = /usr/bin/sleep
arguments = 30
+SingularityImage = "docker://ubuntu"

Requirements = HasSingularity

Request_Disk = 1024
Request_Memory = 1024
Request_cpus = 1

should_transfer_files = yes
transfer_input_files = some_input
when_to_transfer_output = on_exit

log = log
output = out.$(PROCESS)
error = err.$(PROCESS)

queue 1
```

HTCondor can also transfer the whole singularity image, just like any other input file, and use that as the container image. Given a singularity image file in the file named "image" in the submit directory, the submit file would look like:

```
executable = /usr/bin/sleep
arguments = 30
+SingularityImage = "image"

Requirements = HasSingularity

Request_Disk = 1024
Request_Memory = 1024
Request_cpus = 1

should_transfer_files = yes
transfer_input_files = image
when_to_transfer_output = on_exit

log = log
output = out.$(PROCESS)
error = err.$(PROCESS)
```

(continues on next page)

(continued from previous page)

```
queue 1
```

The administrator can optionally specify additional directories to be bind mounted into the container. For example, if there is some common shared input data located on a machine, or on a shared file system, this directory can be bind-mounted and be visible inside the container. This is controlled by the configuration parameter . This is an expression, which is evaluated in the context of the machine and job ads, and which should evaluated to a string which contains a space separated list of directories to mount.

So, to always bind mount a directory named /nfs into the image, and administrator could set

```
SINGULARITY_BIND_EXPR = "/nfs"
```

Or, if a trusted user is allowed to bind mount anything on the host, an expression could be

```
SINGULARITY_BIND_EXPR = (Target.Owner == "TrustedUser") ? SomeExpressionFromJob : ""
```

If the source directory for the bind mount is missing on the host machine, HTCondor will skip that mount and run the job without it. If the image is an exploded file directory, and the target directory is missing inside the image, and the configuration parameter is set to true (the default is false), then this mount attempt will also be skipped. Otherwise, the job will return an error when run.

In general, HTCondor will try to set as many Singularity command line options as possible from settings in the machine ad and job ad, as make sense. For example, if the slot the job runs in is provisioned with GPUs, perhaps in response to a `request_GPUs` line in the submit file, the Singularity flag `-nv` will be passed to Singularity, which should make the appropriate nvidia devices visible inside the container. If the submit file requests environment variables to be set for the job, HTCondor passes those through Singularity into the job.

Before the *condor_starter* runs a job with singularity, it first runs singularity test on that image. If no test is defined inside the image, it runs `/bin/sh /bin/true`. If the test returns non-zero, for example if the image is missing, or malformed, the job is put on hold. This is controlled by the condor knob , which defaults to true.

If an administrator wants to pass additional arguments to the singularity exec command instead of the defaults used by HTCondor, several parameters exist to do this - see the *condor_starter* configuration parameters that begin with the prefix SINGULARITY in defined in section *condor_starter Configuration File Entries*. There you will find parameters to customize things such as the use of PID namespaces, cache directory, and several other options. However, should an administrator need to customize Singularity behavior that HTCondor does not currently support, the parameter allows arbitrary additional parameters to be passed to the singularity exec command. Note that this can be a classad expression, evaluated in the context of the slot ad and the job ad, where the slot ad can be referenced via "MY.", and the job ad via the "TARGET." reference. In this way, the admin could set different options for different kinds of jobs. For example, to pass the `-w` argument, to make the image writable, an administrator could set

```
SINGULARITY_EXTRA_ARGUMENTS = "-w"
```

There are some rarely-used settings that some administrators may need to set. By default, HTCondor looks for the Singularity runtime in `/usr/bin/singularity`, but this can be overridden with the SINGULARITY parameter:

```
SINGULARITY = /opt/singularity/bin/singularity
```

By default, the initial working directory of the job will be the scratch directory, just like a vanilla universe job. This directory probably doesn't exist in the image's file system. Usually, Singularity will be able to create this directory in the image, but unprivileged versions of singularity with certain image types may not be able to do so. If this is the case, the current directory on the inside of the container can be set via a knob. This will still map to the scratch directory outside the container.

```
# Maps $_CONDOR_SCRATCH_DIR on the host to /srv inside the image.
SINGULARITY_TARGET_DIR = /srv
```

If is not specified by the admin, it may be specified in the job submit file via the submit command `container_target_dir`. If both are set, the config knob version takes precedence.

When the HTCondor starter runs a job under Singularity, it always prints to the log the exact command line used. This can be helpful for debugging or for the curious. An example command line printed to the StarterLog might look like the following:

```
About to exec /usr/bin/singularity -s exec -S /tmp -S /var/tmp --pwd /execute/dir_462373␣
↪-B /execute/dir_462373 --no-home -C /images/debian /execute/dir_462373/demo 3
```

In this example, no GPUs have been requested, so there is no `-nv` option. is set to the default of `/tmp,/var/tmp`, so condor translates those into `-S` (scratch directory) requests in the command line. The `--pwd` is set to the scratch directory, `-B` bind mounts the scratch directory with the same name on the inside of the container, and the `-C` option is set to contain all namespaces. Then the image is named, and the executable, which in this case has been transferred by HTCondor into the scratch directory, and the job's argument (3). Not visible in the log are any environment variables that HTCondor is setting for the job.

All of the singularity container runtime's logging, warning and error messages are written to the job's stderr. This is an unfortunate aspect of the runtime we hope to fix in the future. By default, HTCondor passes "-s" (silent) to the singularity runtime, so that the only messages it writes to the job's stderr are fatal error messages. If a worker node administrator needs more debugging information, they can change the value of the worker node config parameter and set it to -d or -v to increase the debugging level.

5.18 Power Management

HTCondor supports placing machines in low power states. A machine in the low power state is identified as being offline. Power setting decisions are based upon HTCondor configuration.

Power conservation is relevant when machines are not in heavy use, or when there are known periods of low activity within the pool.

5.18.1 Entering a Low Power State

By default, HTCondor does not do power management. When desired, the ability to place a machine into a low power state is accomplished through configuration. This occurs when all slots on a machine agree that a low power state is desired.

A slot's readiness to hibernate is determined by the evaluating the configuration variable (see the *condor_startd Configuration File Macros* section) within the context of the slot. Readiness is evaluated at fixed intervals, as determined by the configuration variable. A non-zero value of this variable enables the power management facility. It is an integer value representing seconds, and it need not be a small value. There is a trade off between the extra time not at a low power state and the unnecessary computation of readiness.

To put the machine in a low power state rapidly after it has become idle, consider checking each slot's state frequently, as in the example configuration:

```
HIBERNATE_CHECK_INTERVAL = 20
```

This checks each slot's readiness every 20 seconds. A more common value for frequency of checks is 300 (5 minutes). A value of 300 loses some degree of granularity, but it is more reasonable as machines are likely to be put in to a low power state after a few hours, rather than minutes.

A slot's readiness or willingness to enter a low power state is determined by the HIBERNATE expression. Because this expression is evaluated in the context of each slot, and not on the machine as a whole, any one slot can veto a change of power state. The HIBERNATE expression may reference a wide array of variables. Possibilities include the change in power state if none of the slots are claimed, or if the slots are not in the Owner state.

Here is a concrete example. Assume that the START expression is not set to always be True. This permits an easy determination whether or not the machine is in an Unclaimed state through the use of an auxiliary macro called ShouldHibernate.

```
TimeToWait  = (2 * $(HOUR))
ShouldHibernate = ( (KeyboardIdle > $(StartIdleTime)) \
                    && $(CPUIdle) \
                    && ($(StateTimer) > $(TimeToWait)) )
```

This macro evaluates to True if the following are all True:

- The keyboard has been idle long enough.
- The CPU is idle.
- The slot has been Unclaimed for more than 2 hours.

The sample HIBERNATE expression that enters the power state called "RAM", if ShouldHibernate evaluates to True, and remains in its current state otherwise is

```
HibernateState  = "RAM"
HIBERNATE = ifThenElse($(ShouldHibernate), $(HibernateState), "NONE" )
```

If any slot returns "NONE", that slot vetoes the decision to enter a low power state. Only when values returned by all slots are all non-zero is there a decision to enter a low power state. If all agree to enter the low power state, but differ in which state to enter, then the largest magnitude value is chosen.

5.18.2 Returning From a Low Power State

The HTCondor command line tool *condor_power* may wake a machine from a low power state by sending a UDP Wake On LAN (WOL) packet. See the *condor_power* manual page.

To automatically call *condor_power* under specific conditions, *condor_rooster* may be used. The configuration options for *condor_rooster* are described in the *condor_rooster Configuration File Macros* section.

5.18.3 Keeping a ClassAd for a Hibernating Machine

A pool's *condor_collector* daemon can be configured to keep a persistent ClassAd entry for each machine, once it has entered hibernation. This is required by *condor_rooster* so that it can evaluate the expression of the offline machines.

To do this, define a log file using the configuration variable. See the *condor_startd Configuration File Macros* section for the definition. An optional expiration time for each ClassAd can be specified with . The timing begins from the time the hibernating machine's ClassAd enters the *condor_collector* daemon. See the *condor_startd Configuration File Macros* section for the definition.

5.18.4 Linux Platform Details

Depending on the Linux distribution and version, there are three methods for controlling a machine's power state. The methods:

1. *pm-utils* is a set of command line tools which can be used to detect and switch power states. In HTCondor, this is defined by the string "pm-utils".
2. The directory in the virtual file system `/sys/power` contains virtual files that can be used to detect and set the power states. In HTCondor, this is defined by the string "/sys".
3. The directory in the virtual file system `/proc/acpi` contains virtual files that can be used to detect and set the power states. In HTCondor, this is defined by the string "/proc".

By default, the HTCondor attempts to detect the method to use in the order shown. The first method detected as usable on the system is chosen.

This ordered detection may be bypassed, to use a specified method instead by setting the configuration variable with one of the defined strings. This variable is defined in the *condor_startd Configuration File Macros* section. If no usable methods are detected or the method specified by is either not detected or invalid, hibernation is disabled.

The details of this selection process, and the final method selected can be logged via enabling `D_FULLDEBUG` in the relevant subsystem's log configuration.

5.18.5 Windows Platform Details

If after a suitable amount of time, a Windows machine has not entered the expected power state, then HTCondor is having difficulty exercising the operating system's low power capabilities. While the cause will be specific to the machine's hardware, it may also be due to improperly configured software. For hardware difficulties, the likely culprit is the configuration within the machine's BIOS, for which HTCondor can offer little guidance. For operating system difficulties, the *powercfg* tool can be used to discover the available power states on the machine. The following command demonstrates how to list all of the supported power states of the machine:

```
> powercfg -A
The following sleep states are available on this system:
Standby (S3) Hibernate Hybrid Sleep
The following sleep states are not available on this system:
Standby (S1)
        The system firmware does not support this standby state.
Standby (S2)
        The system firmware does not support this standby state.
```

Note that the `HIBERNATE` expression is written in terms of the Sn state, where n is the value evaluated from the expression.

This tool can also be used to enable and disable other sleep states. This example turns hibernation on.

```
> powercfg -h on
```

If this tool is insufficient for configuring the machine in the manner required, the *Power Options* control panel application offers the full extent of the machine's power management abilities. Windows 2000 and XP lack the *powercfg* program, so all configuration must be done via the *Power Options* control panel application.

5.19 Hooks

A hook is an external program or script invoked by an HTCondor daemon to change its behavior or implement some policy. There are three kinds of Job hooks in HTCondor: Fetch work job hooks, Prepare Job hooks, and Job Router hooks.

5.19.1 Job Hooks That Fetch Work

In the past, HTCondor has always sent work to the execute machines by pushing jobs to the *condor_startd* daemon from the *condor_schedd* daemon. Beginning with the HTCondor version 7.1.0, the *condor_startd* daemon now has the ability to pull work by fetching jobs via a system of plug-ins or hooks. Any site can configure a set of hooks to fetch work, completely outside of the usual HTCondor matchmaking system.

A projected use of the hook mechanism implements what might be termed a glide-in factory, especially where the factory is behind a firewall. Without using the hook mechanism to fetch work, a glide-in *condor_startd* daemon behind a firewall depends on CCB to help it listen and eventually receive work pushed from elsewhere. With the hook mechanism, a glide-in *condor_startd* daemon behind a firewall uses the hook to pull work. The hook needs only an outbound network connection to complete its task, thereby being able to operate from behind the firewall, without the intervention of CCB.

Periodically, each execution slot managed by a *condor_startd* will invoke a hook to see if there is any work that can be fetched. Whenever this hook returns a valid job, the *condor_startd* will evaluate the current state of the slot and decide if it should start executing the fetched work. If the slot is unclaimed and the `Start` expression evaluates to `True`, a new claim will be created for the fetched job. If the slot is claimed, the *condor_startd* will evaluate the `Rank` expression relative to the fetched job, compare it to the value of `Rank` for the currently running job, and decide if the existing job should be preempted due to the fetched job having a higher rank. If the slot is unavailable for whatever reason, the *condor_startd* will refuse the fetched job and ignore it. Either way, once the *condor_startd* decides what it should do with the fetched job, it will invoke another hook to reply to the attempt to fetch work, so that the external system knows what happened to that work unit.

If the job is accepted, a claim is created for it and the slot moves into the Claimed state. As soon as this happens, the *condor_startd* will spawn a *condor_starter* to manage the execution of the job. At this point, from the perspective of the *condor_startd*, this claim is just like any other. The usual policy expressions are evaluated, and if the job needs to be suspended or evicted, it will be. If a higher-ranked job being managed by a *condor_schedd* is matched with the slot, that job will preempt the fetched work.

The *condor_starter* itself can optionally invoke additional hooks to help manage the execution of the specific job. There are hooks to prepare or validate the execution environment for the job, periodically update information about the job as it runs, notify when the job exits, and to take special actions when the job is being evicted.

Assuming there are no interruptions, the job completes, and the *condor_starter* exits, the *condor_startd* will invoke the hook to fetch work again. If another job is available, the existing claim will be reused and a new *condor_starter* is spawned. If the hook returns that there is no more work to perform, the claim will be evicted, and the slot will return to the Owner state.

To aid with the development and debugging of hooks, output sent to stderr by the hooks will be preserved in daemon logs of either the *condor_starter* or *condor_startd* as appropriate.

Work Fetching Hooks Invoked by HTCondor

There are a handful of hooks invoked by HTCondor related to fetching work, some of which are called by the *condor_startd* and others by the *condor_starter*. Each hook is described, including when it is invoked, what task it is supposed to accomplish, what data is passed to the hook, what output is expected, and, when relevant, the exit status expected.

- The hook defined by the configuration variable `<Keyword>_HOOK_FETCH_WORK` is invoked whenever the *condor_startd* wants to see if there is any work to fetch. There is a related configuration variable called `FetchWorkDelay` which determines how long the *condor_startd* will wait between attempts to fetch work, which is described in detail in *Job Hooks That Fetch Work*. `<Keyword>_HOOK_FETCH_WORK` is the most important hook in the whole system, and is the only hook that must be defined for any of the other *condor_startd* hooks to operate.

  Command-line arguments passed to the hook
  None.

  Standard input given to the hook
  ClassAd of the slot that is looking for work.

  Expected standard output from the hook
  ClassAd of a job that can be run. If there is no work, the hook should return no output.

  User id that the hook runs as
  The `<Keyword>_HOOK_FETCH_WORK` hook runs with the same privileges as the *condor_startd*. When Condor was started as root, this is usually the condor user, or the user specified in the `CONDOR_IDS` configuration variable.

  Exit status of the hook
  Ignored.

  The job ClassAd returned by the hook needs to contain enough information for the *condor_starter* to eventually spawn the work. The required and optional attributes in this ClassAd are listed here:

  Attributes for a FetchWork application are either required or optional. The following attributes are required:

  `Cmd`
  This attribute defines the full path to the executable program to be run as a FetchWork application. Since HTCondor does not currently provide any mechanism to transfer files on behalf of FetchWork applications, this path should be a valid path on the machine where the application will be run. It is a string attribute, and must therefore be enclosed in quotation marks (“). There is no default.

  `Owner`
  If the *condor_startd* daemon is executing as root on the resource where a FetchWork application will run, the user must also define `Owner` to specify what user name the application will run as. On Windows, the *condor_startd* daemon always runs as an Administrator service, which is equivalent to running as root on Unix platforms. `Owner` must contain a valid user name on the given FetchWork resource. It is a string attribute, and must therefore be enclosed in quotation marks (“).

  `RequestCpus`
  Required when running on a *condor_startd* that uses partitionable slots. It specifies the number of CPU cores from the partitionable slot allocated for this job.

RequestDisk
Required when running on a *condor_startd* that uses partitionable slots. It specifies the disk space, in Megabytes, from the partitionable slot allocated for this job.

RequestMemory
Required when running on a *condor_startd* that uses partitionable slots. It specifies the memory, in Megabytes, from the partitionable slot allocated for this job.

The following list of attributes are optional:

JobUniverse
This attribute defines what HTCondor job universe to use for the given FetchWork application. The only tested universes are vanilla and java. This attribute must be an integer, with vanilla using the value 5, and java using the value 10.

IWD
IWD is an acronym for Initial Working Directory. It defines the full path to the directory where a given FetchWork application are to be run. Unless the application changes its current working directory, any relative path names used by the application will be relative to the IWD. If any other attributes that define file names (for example, `In`, `Out`, and so on) do not contain a full path, the `IWD` will automatically be pre-pended to those file names. It is a string attribute, and must therefore be enclosed in quotation marks (“). If the `IWD` is not specified, the temporary execution sandbox created by the *condor_starter* will be used as the initial working directory.

In
This string defines the path to the file on the FetchWork resource that should be used as standard input (`stdin`) for the FetchWork application. This file (and all parent directories) must be readable by whatever user the FetchWork application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (“).

Out
This string defines the path to the file on the FetchWork resource that should be used as standard output (`stdout`) for the FetchWork application. This file must be writable (and all parent directories readable) by whatever user the FetchWork application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (“).

Err
This string defines the path to the file on the FetchWork resource that should be used as standard error (`stderr`) for the FetchWork application. This file must be writable (and all parent directories readable) by whatever user the FetchWork application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (“).

Env
This string defines environment variables to set for a given FetchWork application. Each environment variable has the form NAME=value. Multiple variables are delimited with a semicolon. An example: Env = “PATH=/usr/local/bin:/usr/bin;TERM=vt100” It is a string attribute, and must therefore be enclosed in quotation marks (“).

Args
This string attribute defines the list of arguments to be supplied to the program on the command-line. The arguments are delimited (separated) by space characters. There is no default. If the `JobUniverse` corresponds to the Java universe, the first argument must be the name of the class containing `main`. It is a string attribute, and must therefore be enclosed in quotation marks (“).

JarFiles

This string attribute is only used if `JobUniverse` is 10 (the Java universe). If a given FetchWork application is a Java program, specify the JAR files that the program requires with this attribute. There is no default. It is a string attribute, and must therefore be enclosed in quotation marks (“). Multiple file names may be delimited with either commas or white space characters, and therefore, file names can not contain spaces.

`KillSig`
: This attribute specifies what signal should be sent whenever the HTCondor system needs to gracefully shutdown the FetchWork application. It can either be specified as a string containing the signal name (for example KillSig = “SIGQUIT”), or as an integer (KillSig = 3) The default is to use SIGTERM.

`StarterUserLog`
: This string specifies a file name for a log file that the *condor_starter* daemon can write with entries for relevant events in the life of a given FetchWork application. It is similar to the job event log file specified for regular HTCondor jobs with the **Log** command in a submit description file. However, certain attributes that are placed in a job event log do not make sense in the FetchWork environment, and are therefore omitted. The default is not to write this log file. It is a string attribute, and must therefore be enclosed in quotation marks (“).

`StarterUserLogUseXML`
: If the `StarterUserLog` attribute is defined, the default format is a human-readable format. However, HTCondor can write out this log in an XML representation, instead. To enable the XML format for this job event log, the `StarterUserLogUseXML` boolean is set to TRUE. The default if not specified is FALSE.

If any attribute that specifies a path (`Cmd`, `In`, `Out`,`Err`, `StarterUserLog`) is not a full path name, HTCondor automatically prepends the value of `IWD`.

- The hook defined by the configuration variable `<Keyword>_HOOK_REPLY_FETCH` is invoked whenever `<Keyword>_HOOK_FETCH_WORK` returns data and the *condor_startd* decides if it is going to accept the fetched job or not.

  The *condor_startd* will not wait for this hook to return before taking other actions, and it ignores all output. The hook is simply advisory, and it has no impact on the behavior of the *condor_startd*.

  Command-line arguments passed to the hook
  : Either the string accept or reject.

  Standard input given to the hook
  : A copy of the job ClassAd and the slot ClassAd (separated by the string —– and a new line).

  Expected standard output from the hook
  : None.

  User id that the hook runs as
  : The `<Keyword>_HOOK_REPLY_FETCH` hook runs with the same privileges as the *condor_startd*. When Condor was started as root, this is usually the condor user, or the user specified in the `CONDOR_IDS` configuration variable.

  Exit status of the hook
  : Ignored.

- The hook defined by the configuration variable `<Keyword>_HOOK_EVICT_CLAIM` is invoked whenever the *condor_startd* needs to evict a claim representing fetched work.

The *condor_startd* will not wait for this hook to return before taking other actions, and ignores all output. The hook is simply advisory, and has no impact on the behavior of the *condor_startd*.

Command-line arguments passed to the hook
None.

Standard input given to the hook
A copy of the job ClassAd and the slot ClassAd (separated by the string —– and a new line).

Expected standard output from the hook
None.

User id that the hook runs as
The <Keyword>_HOOK_EVICT_CLAIM hook runs with the same privileges as the *condor_startd*. When Condor was started as root, this is usually the condor user, or the user specified in the CONDOR_IDS configuration variable.

Exit status of the hook
Ignored.

Keywords to Define Job Fetch Hooks in the HTCondor Configuration files

Hooks are defined in the HTCondor configuration files by prefixing the name of the hook with a keyword. This way, a given machine can have multiple sets of hooks, each set identified by a specific keyword.

Each slot on the machine can define a separate keyword for the set of hooks that should be used with SLOT<N>_JOB_HOOK_KEYWORD . For example, on slot 1, the variable name will be called SLOT1_JOB_HOOK_KEYWORD. If the slot-specific keyword is not defined, the *condor_startd* will use a global keyword as defined by STARTD_JOB_HOOK_KEYWORD .

Once a job is fetched via <Keyword>_HOOK_FETCH_WORK , the *condor_startd* will insert the keyword used to fetch that job into the job ClassAd as HookKeyword. This way, the same keyword will be used to select the hooks invoked by the *condor_starter* during the actual execution of the job. The STARTER_DEFAULT_JOB_HOOK_KEYWORD config knob can define a default hook keyword to use in the event that keyword defined by the job is invalid or not specified. Alternatively, the STARTER_JOB_HOOK_KEYWORD can be defined to force the *condor_starter* to always use a given keyword for its own hooks, regardless of the value in the job ClassAd for the HookKeyword attribute.

For example, the following configuration defines two sets of hooks, and on a machine with 4 slots, 3 of the slots use the global keyword for running work from a database-driven system, and one of the slots uses a custom keyword to handle work fetched from a web service.

```
# Most slots fetch and run work from the database system.
STARTD_JOB_HOOK_KEYWORD = DATABASE

# Slot4 fetches and runs work from a web service.
SLOT4_JOB_HOOK_KEYWORD = WEB

# The database system needs to both provide work and know the reply
# for each attempted claim.
DATABASE_HOOK_DIR = /usr/local/condor/fetch/database
DATABASE_HOOK_FETCH_WORK = $(DATABASE_HOOK_DIR)/fetch_work.php
DATABASE_HOOK_REPLY_FETCH = $(DATABASE_HOOK_DIR)/reply_fetch.php

# The web system only needs to fetch work.
```

(continues on next page)

(continued from previous page)

```
WEB_HOOK_DIR = /usr/local/condor/fetch/web
WEB_HOOK_FETCH_WORK = $(WEB_HOOK_DIR)/fetch_work.php
```

The keywords "DATABASE" and "WEB" are completely arbitrary, so each site is encouraged to use different (more specific) names as appropriate for their own needs.

Defining the FetchWorkDelay Expression

There are two events that trigger the *condor_startd* to attempt to fetch new work:

- the *condor_startd* evaluates its own state
- the *condor_starter* exits after completing some fetched work

Even if a given compute slot is already busy running other work, it is possible that if it fetched new work, the *condor_startd* would prefer this newly fetched work (via the Rank expression) over the work it is currently running. However, the *condor_startd* frequently evaluates its own state, especially when a slot is claimed. Therefore, administrators can define a configuration variable which controls how long the *condor_startd* will wait between attempts to fetch new work. This variable is called FetchWorkDelay .

The FetchWorkDelay expression must evaluate to an integer, which defines the number of seconds since the last fetch attempt completed before the *condor_startd* will attempt to fetch more work. However, as a ClassAd expression (evaluated in the context of the ClassAd of the slot considering if it should fetch more work, and the ClassAd of the currently running job, if any), the length of the delay can be based on the current state the slot and even the currently running job.

For example, a common configuration would be to always wait 5 minutes (300 seconds) between attempts to fetch work, unless the slot is Claimed/Idle, in which case the *condor_startd* should fetch immediately:

```
FetchWorkDelay = ifThenElse(State == "Claimed" && Activity == "Idle", 0, 300)
```

If the *condor_startd* wants to fetch work, but the time since the last attempted fetch is shorter than the current value of the delay expression, the *condor_startd* will set a timer to fetch as soon as the delay expires.

If this expression is not defined, the *condor_startd* will default to a five minute (300 second) delay between all attempts to fetch work.

5.19.2 Job Hooks That Modify and Monitor Execution

The Job ClassAd can be modified before execution, and the progress of the job can be modified using hooks. These hooks are executed by the *condor_starter* and can be used with or without using Fetch Work hooks.

- The hook defined by the configuration variable <Keyword>_HOOK_PREPARE_JOB_BEFORE_TRANSFER is invoked by the *condor_starter* immediately before transferring the job's input files. This hook provides a chance to execute commands to set up or validate the job environment, and/or edit the job classad that is used by the *condor_starter*.

  The *condor_starter* waits until this hook returns before attempting to transfer the input files for the job. If the hook returns a non-zero exit status, the *condor_starter* will assume an error was reached while attempting to set up the job environment and abort the job.

  Command-line arguments passed to the hook
  None.

Standard input given to the hook
A copy of the job ClassAd.

Expected standard output from the hook
A set of attributes to insert or update into the job ad. For example, changing the `Cmd` attribute to a quoted string changes the executable to be run. Two special attributes can also be specified: `HookStatusCode` and `HookStatusMessage`. `HookStatusCode`, if specified and is not a negative number, will be used instead of the exit status of the hook unless the hook process exited due to a signal. A status code of 0 is success, and a positive integer indicates failure. A status code between 1 and 299 (inclusive) will result in the job going on hold; 300 or greater will result in the job going back to the Idle state. The `HookStatusMessage` will be echoed into the job's event log file, and also be used as the Hold Reason string if the job is placed on hold.

User id that the hook runs as
The `<Keyword>_HOOK_PREPARE_JOB` hook runs with the same privileges as the job itself. If slot users are defined, the hook runs as the slot user, just as the job does.

Exit status of the hook
0 for success preparing the job, any non-zero value on failure.

- The hook defined by the configuration variable `<Keyword>_HOOK_PREPARE_JOB` is invoked by the *condor_starter* before a job is going to be run but after the job's input files have been transferred. This hook provides a chance to execute commands to set up or validate the job environment, and/or edit the job classad that is used by the *condor_starter*.

  The *condor_starter* waits until this hook returns before attempting to execute the job. If the hook returns a non-zero exit status, the *condor_starter* will assume an error was reached while attempting to set up the job environment and abort the job.

  Command-line arguments passed to the hook
  None.

  Standard input given to the hook
  A copy of the job ClassAd.

  Expected standard output from the hook
  A set of attributes to insert or update into the job ad. For example, changing the `Cmd` attribute to a quoted string changes the executable to be run. Two special attributes can also be specified: `HookStatusCode` and `HookStatusMessage`. `HookStatusCode`, if specified and is not a negative number, will be used instead of the exit status of the hook unless the hook process exited due to a signal. A status code of 0 is success, and a positive integer indicates failure. A status code between 1 and 299 (inclusive) will result in the job going on hold; 300 or greater will result in the job going back to the Idle state. The `HookStatusMessage` will be echoed into the job's event log file, and also be used as the Hold Reason string if the job is placed on hold.

  User id that the hook runs as
  The `<Keyword>_HOOK_PREPARE_JOB` hook runs with the same privileges as the job itself. If slot users are defined, the hook runs as the slot user, just as the job does.

  Exit status of the hook
  0 for success preparing the job, any non-zero value on failure.

- The hook defined by the configuration variable `<Keyword>_HOOK_UPDATE_JOB_INFO` is invoked periodically during the life of the job to update information about the status of the job. When the job is first spawned, the *condor_starter* will invoke this hook after `STARTER_INITIAL_UPDATE_INTERVAL` seconds (defaults to 8).

Thereafter, the *condor_starter* will invoke the hook every `STARTER_UPDATE_INTERVAL` seconds (defaults to 300, which is 5 minutes).

The *condor_starter* will not wait for this hook to return before taking other actions, and ignores all output. The hook is simply advisory, and has no impact on the behavior of the *condor_starter*.

Command-line arguments passed to the hook
None.

Standard input given to the hook
A copy of the job ClassAd that has been augmented with additional attributes describing the current status and execution behavior of the job.

The additional attributes included inside the job ClassAd are:

`JobState`
The current state of the job. Can be either `"Running"` or `"Suspended"`.

`JobPid`
The process identifier for the initial job directly spawned by the *condor_starter*.

`NumPids`
The number of processes that the job has currently spawned.

`JobStartDate`
The epoch time when the job was first spawned by the *condor_starter*.

`RemoteSysCpu`
The total number of seconds of system CPU time (the time spent at system calls) the job has used.

`RemoteUserCpu`
The total number of seconds of user CPU time the job has used.

`ImageSize`
The memory image size of the job in Kbytes.

Expected standard output from the hook
None.

User id that the hook runs as
The `<Keyword>_HOOK_UPDATE_JOB_INFO` hook runs with the same privileges as the job itself.

Exit status of the hook
Ignored.

- The hook defined by the configuration variable `<Keyword>_HOOK_JOB_EXIT` is invoked by the *condor_starter* whenever a job exits, either on its own or when being evicted from an execution slot.

  The *condor_starter* will wait for this hook to return before taking any other actions. In the case of jobs that are being managed by a *condor_shadow*, this hook is invoked before the *condor_starter* does its own optional file transfer back to the submission machine, writes to the local job event log file, or notifies the *condor_shadow* that the job has exited.

  Command-line arguments passed to the hook
  A string describing how the job exited:

  - exit The job exited or died with a signal on its own.
  - remove The job was removed with *condor_rm* or as the result of user job policy expressions (for example, `PeriodicRemove`).

- hold The job was held with *condor_hold* or the user job policy expressions (for example, PeriodicHold).
- evict The job was evicted from the execution slot for any other reason (PREEMPT evaluated to TRUE in the *condor_startd*, *condor_vacate*, *condor_off*, etc).

Standard input given to the hook
A copy of the job ClassAd that has been augmented with additional attributes describing the execution behavior of the job and its final results.

The job ClassAd passed to this hook contains all of the extra attributes described above for <Keyword>_HOOK_UPDATE_JOB_INFO , and the following additional attributes that are only present once a job exits:

ExitReason
A human-readable string describing why the job exited.

ExitBySignal
A boolean indicating if the job exited due to being killed by a signal, or if it exited with an exit status.

ExitSignal
If ExitBySignal is true, the signal number that killed the job.

ExitCode
If ExitBySignal is false, the integer exit code of the job.

JobDuration
The number of seconds that the job ran during this invocation.

Expected standard output from the hook
None.

User id that the hook runs as
The <Keyword>_HOOK_JOB_EXIT hook runs with the same privileges as the job itself.

Exit status of the hook
Ignored.

Example Hook: Specifying the Executable at Execution Time

The availability of multiple versions of an application leads to the need to specify one of the versions. As an example, consider that the java universe utilizes a single, fixed JVM. There may be multiple JVMs available, and the HTCondor job may need to make the choice of JVM version. The use of a job hook solves this problem. The job does not use the java universe, and instead uses the vanilla universe in combination with a prepare job hook to overwrite the Cmd attribute of the job ClassAd. This attribute is the name of the executable the *condor_starter* daemon will invoke, thereby selecting the specific JVM installation.

In the configuration of the execute machine:

```
JAVA5_HOOK_PREPARE_JOB = $(LIBEXEC)/java5_prepare_hook
```

With this configuration, a job that sets the HookKeyword attribute with

```
+HookKeyword = "JAVA5"
```

in the submit description file causes the *condor_starter* will run the hook specified by JAVA5_HOOK_PREPARE_JOB before running this job. Note that the double quote marks are required to correctly define the attribute. Any output from this hook is an update to the job ClassAd. Therefore, the hook that changes the executable may be

```
#!/bin/sh

# Read and discard the job ClassAd
cat > /dev/null
echo 'Cmd = "/usr/java/java5/bin/java"'
```

If some machines in your pool have this hook and others do not, this fact should be advertised. Add to the configuration of every execute machine that has the hook:

```
HasJava5PrepareHook = True
STARTD_ATTRS = HasJava5PrepareHook $(STARTD_ATTRS)
```

The submit description file for this example job may be

```
universe = vanilla
executable = /usr/bin/java
arguments = Hello
# match with a machine that has the hook
requirements = HasJava5PrepareHook

should_transfer_files = always
when_to_transfer_output = on_exit
transfer_input_files = Hello.class

output = hello.out
error  = hello.err
log    = hello.log

+HookKeyword="JAVA5"

queue
```

Note that the **requirements** command ensures that this job matches with a machine that has JAVA5_HOOK_PREPARE_JOB defined.

5.19.3 Hooks for the Job Router

Job Router Hooks allow for an alternate transformation and/or monitoring than the *condor_job_router* daemon implements. Routing is still managed by the *condor_job_router* daemon, but if the Job Router Hooks are specified, then these hooks will be used to transform and monitor the job instead.

Job Router Hooks are similar in concept to Fetch Work Hooks, but they are limited in their scope. A hook is an external program or script invoked by the *condor_job_router* daemon at various points during the life cycle of a routed job.

The following sections describe how and when these hooks are used, what hooks are invoked at various stages of the job's life, and how to configure HTCondor to use these Hooks.

Hooks Invoked for Job Routing

The Job Router Hooks allow for replacement of the transformation engine used by HTCondor for routing a job. Since the external transformation engine is not controlled by HTCondor, additional hooks provide a means to update the job's status in HTCondor, and to clean up upon exit or failure cases. This allows one job to be transformed to just about any other type of job that HTCondor supports, as well as to use execution nodes not normally available to HTCondor.

It is important to note that if the Job Router Hooks are utilized, then HTCondor will not ignore or work around a failure in any hook execution. If a hook is configured, then HTCondor assumes its invocation is required and will not continue by falling back to a part of its internal engine. For example, if there is a problem transforming the job using the hooks, HTCondor will not fall back on its transformation accomplished without the hook to process the job.

There are 2 ways in which the Job Router Hooks may be enabled. A job's submit description file may cause the hooks to be invoked with

```
+HookKeyword = "HOOKNAME"
```

Adding this attribute to the job's ClassAd causes the *condor_job_router* daemon on the access point to invoke hooks prefixed with the defined keyword. `HOOKNAME` is a string chosen as an example; any string may be used.

The job's ClassAd attribute definition of `HookKeyword` takes precedence, but if not present, hooks may be enabled by defining on the access point the configuration variable

```
JOB_ROUTER_HOOK_KEYWORD = HOOKNAME
```

Like the example attribute above, `HOOKNAME` represents a chosen name for the hook, replaced as desired or appropriate.

There are 4 hooks that the Job Router can be configured to use. Each hook will be described below along with data passed to the hook and expected output. All hooks must exit successfully.

- The hook defined by the configuration variable `<Keyword>_HOOK_TRANSLATE_JOB` is invoked when the Job Router has determined that a job meets the definition for a route. This hook is responsible for doing the transformation of the job and configuring any resources that are external to HTCondor if applicable.

  Command-line arguments passed to the hook
  None.

  Standard input given to the hook
  The first line will be the information on route that the job matched including the route name. This information will be formatted as a classad. If the route has a `TargetUniverse` or `GridResource` they will be included in the classad. The route information classad will be followed by a separator line of dashes like `------` followed by a newline. The remainder of the input will be the job ClassAd.

  Expected standard output from the hook
  The transformed job.

  Exit status of the hook
  0 for success, any non-zero value on failure.

- The hook defined by the configuration variable `<Keyword>_HOOK_UPDATE_JOB_INFO` is invoked to provide status on the specified routed job when the Job Router polls the status of routed jobs at intervals set by `JOB_ROUTER_POLLING_PERIOD` .

  Command-line arguments passed to the hook
  None.

Standard input given to the hook
The routed job ClassAd that is to be updated.

Expected standard output from the hook
The job attributes to be updated in the routed job, or nothing, if there was no update. To prevent clashing with HTCondor's management of job attributes, only attributes that are not managed by HTCondor should be output from this hook.

Exit status of the hook
0 for success, any non-zero value on failure.

- The hook defined by the configuration variable `<Keyword>_HOOK_JOB_FINALIZE` is invoked when the Job Router has found that the job has completed. Any output from the hook is treated as an update to the source job.

  Command-line arguments passed to the hook
  None.

  Standard input given to the hook
  The source job ClassAd, followed by the routed copy Classad that completed, separated by the string "——" and a new line.

  Expected standard output from the hook
  An updated source job ClassAd, or nothing if there was no update.

  Exit status of the hook
  0 for success, any non-zero value on failure.

- The hook defined by the configuration variable `<Keyword>_HOOK_JOB_CLEANUP` is invoked when the Job Router finishes managing the job. This hook will be invoked regardless of whether the job completes successfully or not, and must exit successfully.

  Command-line arguments passed to the hook
  None.

  Standard input given to the hook
  The job ClassAd that the Job Router is done managing.

  Expected standard output from the hook
  None.

  Exit status of the hook
  0 for success, any non-zero value on failure.

5.20 Directories

HTCondor uses a few different directories, some of which are role-specific. Do not use these directories for any other purpose, and do not share these directories between machines. The directories are listed in here by the name of the configuration option used to tell HTCondor where they are; you will not normally need to change these.

5.20.1 Directories used by More than One Role

LOG
Each HTCondor daemon writes its own log file, and each log file is placed in the directory. You can configure the name of each daemon's log by setting , although you should never need to do so. You can also control the sizes of the log files or how often they rotate; see *Daemon Logging Configuration File Entries* for details. If you want to write your logs to a shared filesystem, we recommend including `$(HOSTNAME)` in the value of `LOG` rather than changing the names of each individual log to not collide. If you set `LOG` to a shared filesystem, you should set `LOCK` to a local filesystem; see below.

LOCK
HTCondor uses a small number of lock files to synchronize access to certain files that are shared between multiple daemons. Because of problems encountered with file locking and network file systems (particularly NFS), these lock files should be placed on a local filesystem on each machine. By default, they are placed in the `LOG` directory.

5.20.2 Directories use by the Submit Role

SPOOL
The directory holds two types of files: system data and (user) job data. The former includes the job queue and history files. The latter includes:

- the files transferred, if any, when a job which set `when_to_transfer_files` to `EXIT_OR_EVICT` is evicted.
- the input and output files of remotely-submitted jobs.
- the checkpoint files stored by self-checkpointing jobs.

Disk usage therefore varies widely based on the job mix, but since the schedd will abort if it can't append to the job queue log, you want to make sure this directory is on a partition which won't run out of space.

To help ensure this, you may set to separate the job queue log (system data) from the (user) job data. This can also be used to increase performance (or reliability) by moving the job queue log to specialized hardware (an SSD or a a high-redundancy RAID, for example).

5.20.3 Directories use by the Execute Role

EXECUTE
The directory is the parent directory of the current working directory for any HTCondor job that runs on a given execute-role machine. HTCondor copies the executable and input files for a job to its subdirectory; the job's standard output and standard error streams are also logged here. Jobs will also almost always generate their output here as well, so the `EXECUTE` directory should provide a plenty of space. `EXECUTE` should not be placed under /tmp or /var/tmp if possible, as HTCondor loses the ability to make /tmp and /var/tmp private to the job. While not a requirement, ideally `EXECUTE` should be on a distinct filesystem, so that it is impossible for a rogue job to fill up non-HTCondor related partitions.

Usually, the per-job scratch execute directory is created by the startd as a directory under `EXECUTE`. However, on Linux machines where HTCondor has root privilege, it can be configured to make an ephemeral, per-job scratch filesystem backed either by LVM, if it is configured, or a large existing file on the filesystem.

There are several advantages to this approach. The first is that disk space is more accurately measured and enforced. HTCondor can get the disk usage by a single system call, instead of traversing what

might be a very deep directory hierarchy. There may also be performance benefits, as this filesystem never needs to survive a reboot, and is thus mounted with mount options that provide the least amount of disk consistence in the face of a reboot. Also, when the job exits, all the files in the filesystem can be removed by simply unmounting and destroying the filesystem, which is much faster than having condor remove each scratch file in turn.

To enable this, first set to `true`. Then, if LVM is installed and configured, set to the name of a logical volume. `"condor_lv"` might be a good choice. Finally, set to the name of the volume group the LVM administrator has created for this purpose. `"condor_vg"` might be a good name. If there is no LVM on the system, a single large existing file can be used as the backing store, in which case the knob should be set to the name of the existing large file on disk that HTCondor will use to make filesystems from.

Warning: The per job filesystem feature is a work in progress and not currently supported.

5.21 Setting Up for Special Environments

The following sections describe how to set up HTCondor for use in special environments or configurations.

5.21.1 Configuring HTCondor for Multiple Platforms

A single, initial configuration file may be used for all platforms in an HTCondor pool, with platform-specific settings placed in separate files. This greatly simplifies administration of a heterogeneous pool by allowing specification of platform-independent, global settings in one place, instead of separately for each platform. This is made possible by treating the configuration variable as a list of files, instead of a single file. Of course, this only helps when using a shared file system for the machines in the pool, so that multiple machines can actually share a single set of configuration files.

With multiple platforms, put all platform-independent settings (the vast majority) into the single initial configuration file, which will be shared by all platforms. Then, set the configuration variable from that global configuration file to specify both a platform-specific configuration file and optionally, a local, machine-specific configuration file.

The name of platform-specific configuration files may be specified by using `$(ARCH)` and `$(OPSYS)`, as defined automatically by HTCondor. For example, for 32-bit Intel Windows 7 machines and 64-bit Intel Linux machines, the files ought to be named:

```
$ condor_config.INTEL.WINDOWS
condor_config.X86_64.LINUX
```

Then, assuming these files are in the directory defined by the `ETC` configuration variable, and machine-specific configuration files are in the same directory, named by each machine's host name, becomes:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.$(ARCH).$(OPSYS), \
                    $(ETC)/$(HOSTNAME).local
```

Alternatively, when using AFS, an `@sys` link may be used to specify the platform-specific configuration file, which lets AFS resolve this link based on platform name. For example, consider a soft link named `condor_config.platform` that points to `condor_config.@sys`. In this case, the files might be named:

```
$ condor_config.i386_linux2
condor_config.platform -> condor_config.@sys
```

and the configuration variable would be set to

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.platform, \
                    $(ETC)/$(HOSTNAME).local
```

Platform-Specific Configuration File Settings

The configuration variables that are truly platform-specific are:

Full path to the installed HTCondor binaries. While the configuration files may be shared among different platforms, the binaries certainly cannot. Therefore, maintain separate release directories for each platform in the pool.

The full path to the mail program.

Which devices in `/dev` should be treated as console devices.

Which daemons the *condor_master* should start up. The reason this setting is platform-specific is to distinguish the *condor_kbdd*. It is needed on many Linux and Windows machines, and it is not needed on other platforms.

Reasonable defaults for all of these configuration variables will be found in the default configuration files inside a given platform's binary distribution (except the , since the location of the HTCondor binaries and libraries is installation specific). With multiple platforms, use one of the `condor_config` files from either running *condor_configure* or from the `$(RELEASE_DIR)`/etc/examples/condor_config.generic file, take these settings out, save them into a platform-specific file, and install the resulting platform-independent file as the global configuration file. Then, find the same settings from the configuration files for any other platforms to be set up, and put them in their own platform-specific files. Finally, set the configuration variable to point to the appropriate platform-specific file, as described above.

Not even all of these configuration variables are necessarily going to be different. For example, if an installed mail program understands the **-s** option in `/usr/local/bin/mail` on all platforms, the macro may be set to that in the global configuration file, and not define it anywhere else. For a pool with only Linux or Windows machines, the will be the same for each, so there is no reason not to put that in the global configuration file.

Other Uses for Platform-Specific Configuration Files

An installation may want other configuration variables to be platform-specific. Perhaps a different policy is desired for one of the platforms. Perhaps different people should get the e-mail about problems with the different platforms. There is nothing hard-coded about any of this. What is shared and what should not shared is entirely configurable.

Since the macro can be an arbitrary list of files, an installation can even break up the global, platform-independent settings into separate files. In fact, the global configuration file might only contain a definition for , and all other configuration variables would be placed in separate files.

Different people may be given different permissions to change different HTCondor settings. For example, if a user is to be able to change certain settings, but nothing else, those settings may be placed in a file which was early in the list, to give that user write permission on that file. Then, include all the other files after that one. In this way, if the user was attempting to change settings that the user should not be permitted to change, the settings would be overridden.

This mechanism is quite flexible and powerful. For very specific configuration needs, they can probably be met by using file permissions, the configuration variable, and imagination.

5.21.2 The *condor_kbdd*

The HTCondor keyboard daemon, *condor_kbdd*, monitors X events on machines where the operating system does not provide a way of monitoring the idle time of the keyboard or mouse. On Linux platforms, it is needed to detect USB keyboard activity. Otherwise, it is not needed. On Windows platforms, the *condor_kbdd* is the primary way of monitoring the idle time of both the keyboard and mouse.

The *condor_kbdd* on Windows Platforms

Windows platforms need to use the *condor_kbdd* to monitor the idle time of both the keyboard and mouse. By adding `KBDD` to configuration variable , the *condor_master* daemon invokes the *condor_kbdd*, which then does the right thing to monitor activity given the version of Windows running.

With Windows Vista and more recent version of Windows, user sessions are moved out of session 0. Therefore, the *condor_startd* service is no longer able to listen to keyboard and mouse events. The *condor_kbdd* will run in an invisible window and should not be noticeable by the user, except for a listing in the task manager. When the user logs out, the program is terminated by Windows. This implementation also appears in versions of Windows that predate Vista, because it adds the capability of monitoring keyboard activity from multiple users.

To achieve the auto-start with user login, the HTCondor installer adds a *condor_kbdd* entry to the registry key at HKLM\Software\Microsoft\Windows\CurrentVersion\Run. On 64-bit versions of Vista and more recent Windows versions, the entry is actually placed in HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run.

In instances where the *condor_kbdd* is unable to connect to the *condor_startd*, it is likely because an exception was not properly added to the Windows firewall.

The *condor_kbdd* on Linux Platforms

On Linux platforms, great measures have been taken to make the *condor_kbdd* as robust as possible, but the X window system was not designed to facilitate such a need, and thus is not as efficient on machines where many users frequently log in and out on the console.

In order to work with X authority, which is the system by which X authorizes processes to connect to X servers, the *condor_kbdd* needs to run with super user privileges. Currently, the *condor_kbdd* assumes that X uses the `HOME` environment variable in order to locate a file named `.Xauthority`. This file contains keys necessary to connect to an X server. The keyboard daemon attempts to set `HOME` to various users' home directories in order to gain a connection to the X server and monitor events. This may fail to work if the keyboard daemon is not allowed to attach to the X server, and the state of a machine may be incorrectly set to idle when a user is, in fact, using the machine.

In some environments, the *condor_kbdd* will not be able to connect to the X server because the user currently logged into the system keeps their authentication token for using the X server in a place that no local user on the current machine can get to. This may be the case for files on AFS, because the user's `.Xauthority` file is in an AFS home directory.

There may also be cases where the *condor_kbdd* may not be run with super user privileges because of political reasons, but it is still desired to be able to monitor X activity. In these cases, change the XDM configuration in order to start up the *condor_kbdd* with the permissions of the logged in user. If running X11R6.3, the files to edit will probably be in `/usr/X11R6/lib/X11/xdm`. The `.xsession` file should start up the *condor_kbdd* at the end, and the `.Xreset` file should shut down the *condor_kbdd*. The **-l** option can be used to write the daemon's log file to a place where the user running the daemon has permission to write a file. The file's recommended location will be similar to `$HOME/.kbdd.log`, since this is a place where every user can write, and the file will not get in the way. The **-pidfile** and **-k** options allow for easy shut down of the *condor_kbdd* by storing the process ID in a file. It will be necessary to add lines to the XDM configuration similar to

```
$ condor_kbdd -l $HOME/.kbdd.log -pidfile $HOME/.kbdd.pid
```

This will start the *condor_kbdd* as the user who is currently logged in and write the log to a file in the directory `$HOME/.kbdd.log/`. This will also save the process ID of the daemon to `~/.kbdd.pid`, so that when the user logs out, XDM can do:

```
$ condor_kbdd -k $HOME/.kbdd.pid
```

This will shut down the process recorded in file `~/.kbdd.pid` and exit.

To see how well the keyboard daemon is working, review the log for the daemon and look for successful connections to the X server. If there are none, the *condor_kbdd* is unable to connect to the machine's X server.

5.21.3 Configuring The HTCondorView Server

The HTCondorView server is an alternate use of the *condor_collector* that logs information on disk, providing a persistent, historical database of pool state. This includes machine state, as well as the state of jobs submitted by users.

An existing *condor_collector* may act as the HTCondorView collector through configuration. This is the simplest situation, because the only change needed is to turn on the logging of historical information. The alternative of configuring a new *condor_collector* to act as the HTCondorView collector is slightly more complicated, while it offers the advantage that the same HTCondorView collector may be used for several pools as desired, to aggregate information into one place.

The following sections describe how to configure a machine to run a HTCondorView server and to configure a pool to send updates to it.

Configuring a Machine to be a HTCondorView Server

To configure the HTCondorView collector, a few configuration variables are added or modified for the *condor_collector* chosen to act as the HTCondorView collector. These configuration variables are described in *condor_collector Configuration File Entries*. Here are brief explanations of the entries that must be customized:

> The directory where historical data will be stored. This directory must be writable by whatever user the HTCondorView collector is running as (usually the user condor). There is a configurable limit to the maximum space required for all the files created by the HTCondorView server called ().
>
> NOTE: This directory should be separate and different from the `spool` or `log` directories already set up for HTCondor. There are a few problems putting these files into either of those directories.

> A boolean value that determines if the HTCondorView collector should store the historical information. It is `False` by default, and must be specified as `True` in the local configuration file to enable data collection.

Once these settings are in place in the configuration file for the HTCondorView server host, create the directory specified in and make it writable by the user the HTCondorView collector is running as. This is the same user that owns the `CollectorLog` file in the `log` directory. The user is usually condor.

If using the existing *condor_collector* as the HTCondorView collector, no further configuration is needed. To run a different *condor_collector* to act as the HTCondorView collector, configure HTCondor to automatically start it.

If using a separate host for the HTCondorView collector, to start it, add the value macro:*COLLECTOR* to macro:*DAEMON_LIST*, and restart HTCondor on that host. To run the HTCondorView collector on the same host

as another *condor_collector*, ensure that the two *condor_collector* daemons use different network ports. Here is an example configuration in which the main *condor_collector* and the HTCondorView collector are started up by the same *condor_master* daemon on the same machine. In this example, the HTCondorView collector uses port 12345.

```
VIEW_SERVER = $(COLLECTOR)
VIEW_SERVER_ARGS = -f -p 12345
VIEW_SERVER_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/ViewServerLog"
DAEMON_LIST = MASTER, NEGOTIATOR, COLLECTOR, VIEW_SERVER
```

For this change to take effect, restart the *condor_master* on this host. This may be accomplished with the *condor_restart* command, if the command is run with administrator access to the pool.

5.21.4 HTCondor's Dedicated Scheduling

The dedicated scheduler is a part of the *condor_schedd* that handles the scheduling of parallel jobs that require more than one machine concurrently running per job. MPI applications are a common use for the dedicated scheduler, but parallel applications which do not require MPI can also be run with the dedicated scheduler. All jobs which use the parallel universe are routed to the dedicated scheduler within the *condor_schedd* they were submitted to. A default HTCondor installation does not configure a dedicated scheduler; the administrator must designate one or more *condor_schedd* daemons to perform as dedicated scheduler.

Selecting and Setting Up a Dedicated Scheduler

We recommend that you select a single machine within an HTCondor pool to act as the dedicated scheduler. This becomes the machine from upon which all users submit their parallel universe jobs. The perfect choice for the dedicated scheduler is the single, front-end machine for a dedicated cluster of compute nodes. For the pool without an obvious choice for a access point, choose a machine that all users can log into, as well as one that is likely to be up and running all the time. All of HTCondor's other resource requirements for a access point apply to this machine, such as having enough disk space in the spool directory to hold jobs. See *Directories* for more information.

Configuration Examples for Dedicated Resources

Each execute machine may have its own policy for the execution of jobs, as set by configuration. Each machine with aspects of its configuration that are dedicated identifies the dedicated scheduler. And, the ClassAd representing a job to be executed on one or more of these dedicated machines includes an identifying attribute. An example configuration file with the following various policy settings is `/etc/examples/condor_config.local.dedicated.resource`.

Each execute machine defines the configuration variable , which identifies the dedicated scheduler it is managed by. The local configuration file contains a modified form of

```
DedicatedScheduler = "DedicatedScheduler@full.host.name"
STARTD_ATTRS = $(STARTD_ATTRS), DedicatedScheduler
```

Substitute the host name of the dedicated scheduler machine for the string "full.host.name".

If running personal HTCondor, the name of the scheduler includes the user name it was started as, so the configuration appears as:

```
DedicatedScheduler = "DedicatedScheduler@username@full.host.name"
STARTD_ATTRS = $(STARTD_ATTRS), DedicatedScheduler
```

All dedicated execute machines must have policy expressions which allow for jobs to always run, but not be preempted. The resource must also be configured to prefer jobs from the dedicated scheduler over all other jobs. Therefore, configuration gives the dedicated scheduler of choice the highest rank. It is worth noting that HTCondor puts no other requirements on a resource for it to be considered dedicated.

Job ClassAds from the dedicated scheduler contain the attribute `Scheduler`. The attribute is defined by a string of the form

```
Scheduler = "DedicatedScheduler@full.host.name"
```

The host name of the dedicated scheduler substitutes for the string full.host.name.

Different resources in the pool may have different dedicated policies by varying the local configuration.

Policy Scenario: Machine Runs Only Jobs That Require Dedicated Resources
One possible scenario for the use of a dedicated resource is to only run jobs that require the dedicated resource. To enact this policy, configure the following expressions:

```
START     = Scheduler =?= $(DedicatedScheduler)
SUSPEND   = False
CONTINUE  = True
PREEMPT   = False
KILL      = False
WANT_SUSPEND   = False
WANT_VACATE    = False
RANK      = Scheduler =?= $(DedicatedScheduler)
```

The expression specifies that a job with the `Scheduler` attribute must match the string corresponding `DedicatedScheduler` attribute in the machine ClassAd. The expression specifies that this same job (with the `Scheduler` attribute) has the highest rank. This prevents other jobs from preempting it based on user priorities. The rest of the expressions disable any other of the *condor_startd* daemon's pool-wide policies, such as those for evicting jobs when keyboard and CPU activity is discovered on the machine.

Policy Scenario: Run Both Jobs That Do and Do Not Require Dedicated Resources
While the first example works nicely for jobs requiring dedicated resources, it can lead to poor utilization of the dedicated machines. A more sophisticated strategy allows the machines to run other jobs, when no jobs that require dedicated resources exist. The machine is configured to prefer jobs that require dedicated resources, but not prevent others from running.

To implement this, configure the machine as a dedicated resource as above, modifying only the expression:

```
START = True
```

Policy Scenario: Adding Desktop Resources To The Mix
A third policy example allows all jobs. These desktop machines use a preexisting expression that takes the machine owner's usage into account for some jobs. The machine does not preempt jobs that must run on dedicated resources, while it may preempt other jobs as defined by policy. So, the default pool policy is used for starting and stopping jobs, while jobs that require a dedicated resource always start and are not preempted.

The , , , and macro:*RANK* policies are set in the global configuration. Locally, the configuration is modified to this hybrid policy by adding a second case.

```
SUSPEND    = Scheduler =!= $(DedicatedScheduler) && ($(SUSPEND))
PREEMPT    = Scheduler =!= $(DedicatedScheduler) && ($(PREEMPT))
RANK_FACTOR    = 1000000
RANK   = (Scheduler =?= $(DedicatedScheduler) * $(RANK_FACTOR)) \
```

(continues on next page)

(continued from previous page)

```
                + $(RANK)
START  = (Scheduler =?= $(DedicatedScheduler)) || ($(START))
```

Define RANK_FACTOR to be a larger value than the maximum value possible for the existing rank expression. is a floating point value, so there is no harm in assigning a very large value.

Preemption with Dedicated Jobs

The dedicated scheduler can be configured to preempt running parallel universe jobs in favor of higher priority parallel universe jobs. Note that this is different from preemption in other universes, and parallel universe jobs cannot be preempted either by a machine's user pressing a key or by other means.

By default, the dedicated scheduler will never preempt running parallel universe jobs. Two configuration variables control preemption of these dedicated resources: and . These variables have no default value, so if either are not defined, preemption will never occur. must evaluate to `True` for a machine to be a candidate for this kind of preemption. If more machines are candidates for preemption than needed to satisfy a higher priority job, the machines are sorted by , and only the highest ranked machines are taken.

Note that preempting one node of a running parallel universe job requires killing the entire job on all of its nodes. So, when preemption occurs, it may end up freeing more machines than are needed for the new job. Also, preempted jobs will be re-run, starting again from the beginning. Thus, the administrator should be careful when enabling preemption of these dedicated resources. Enable dedicated preemption with the configuration:

```
STARTD_JOB_ATTRS = JobPrio
SCHEDD_PREEMPTION_REQUIREMENTS = (My.JobPrio < Target.JobPrio)
SCHEDD_PREEMPTION_RANK = 0.0
```

In this example, preemption is enabled by user-defined job priority. If a set of machines is running a job at user priority 5, and the user submits a new job at user priority 10, the running job will be preempted for the new job. The old job is put back in the queue, and will begin again from the beginning when assigned to a newly acquired set of machines.

Grouping Dedicated Nodes into Parallel Scheduling Groups

In some parallel environments, machines are divided into groups, and jobs should not cross groups of machines. That is, all the nodes of a parallel job should be allocated to machines within the same group. The most common example is a pool of machine using InfiniBand switches. For example, each switch might connect 16 machines, and a pool might have 160 machines on 10 switches. If the InfiniBand switches are not routed to each other, each job must run on machines connected to the same switch. The dedicated scheduler's Parallel Scheduling Groups feature supports this operation.

Each *condor_startd* must define which group it belongs to by setting the variable in the configuration file, and advertising it into the machine ClassAd. The value of this variable is a string, which should be the same for all *condor_startd* daemons within a given group. The property must be advertised in the *condor_startd* ClassAd by appending `ParallelSchedulingGroup` to the configuration variable.

The submit description file for a parallel universe job which must not cross group boundaries contains

```
+WantParallelSchedulingGroups = True
```

The dedicated scheduler enforces the allocation to within a group.

5.21.5 Configuring HTCondor for Running Backfill Jobs

HTCondor can be configured to run backfill jobs whenever the *condor_startd* has no other work to perform. These jobs are considered the lowest possible priority, but when machines would otherwise be idle, the resources can be put to good use.

Currently, HTCondor only supports using the Berkeley Open Infrastructure for Network Computing (BOINC) to provide the backfill jobs. More information about BOINC is available at http://boinc.berkeley.edu.

The rest of this section provides an overview of how backfill jobs work in HTCondor, details for configuring the policy for when backfill jobs are started or killed, and details on how to configure HTCondor to spawn the BOINC client to perform the work.

Overview of Backfill jobs in HTCondor

Whenever a resource controlled by HTCondor is in the Unclaimed/Idle state, it is totally idle; neither the interactive user nor an HTCondor job is performing any work. Machines in this state can be configured to enter the Backfill state, which allows the resource to attempt a background computation to keep itself busy until other work arrives (either a user returning to use the machine interactively, or a normal HTCondor job). Once a resource enters the Backfill state, the *condor_startd* will attempt to spawn another program, called a backfill client, to launch and manage the backfill computation. When other work arrives, the *condor_startd* will kill the backfill client and clean up any processes it has spawned, freeing the machine resources for the new, higher priority task. More details about the different states an HTCondor resource can enter and all of the possible transitions between them are described in *Policy Configuration for Execution Points and for Access Points*, especially the *condor_startd Policy Configuration* and *condor_schedd Policy Configuration* sections.

At this point, the only backfill system supported by HTCondor is BOINC. The *condor_startd* has the ability to start and stop the BOINC client program at the appropriate times, but otherwise provides no additional services to configure the BOINC computations themselves. Future versions of HTCondor might provide additional functionality to make it easier to manage BOINC computations from within HTCondor. For now, the BOINC client must be manually installed and configured outside of HTCondor on each backfill-enabled machine.

Defining the Backfill Policy

There are a small set of policy expressions that determine if a *condor_startd* will attempt to spawn a backfill client at all, and if so, to control the transitions in to and out of the Backfill state. This section briefly lists these expressions. More detail can be found in *condor_startd Configuration File Macros*.

A boolean value to determine if any backfill functionality should be used. The default value is `False`.

A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported string is `"BOINC"`.

A boolean expression to control if an HTCondor resource should start a backfill client. This expression is only evaluated when the machine is in the Unclaimed/Idle state and the expression is `True`.

A boolean expression that is evaluated whenever an HTCondor resource is in the Backfill state. A value of

`True` indicates the machine should immediately kill the currently running backfill client and any other spawned processes, and return to the Owner state.

The following example shows a possible configuration to enable backfill:

```
# Turn on backfill functionality, and use BOINC
ENABLE_BACKFILL = TRUE
BACKFILL_SYSTEM = BOINC

# Spawn a backfill job if we've been Unclaimed for more than 5
# minutes
START_BACKFILL = $(StateTimer) > (5 * $(MINUTE))

# Evict a backfill job if the machine is busy (based on keyboard
# activity or cpu load)
EVICT_BACKFILL = $(MachineBusy)
```

Overview of the BOINC system

The BOINC system is a distributed computing environment for solving large scale scientific problems. A detailed explanation of this system is beyond the scope of this manual. Thorough documentation about BOINC is available at their website: http://boinc.berkeley.edu. However, a brief overview is provided here for sites interested in using BOINC with HTCondor to manage backfill jobs.

BOINC grew out of the relatively famous SETI@home computation, where volunteers installed special client software, in the form of a screen saver, that contacted a centralized server to download work units. Each work unit contained a set of radio telescope data and the computation tried to find patterns in the data, a sign of intelligent life elsewhere in the universe, hence the name: "Search for Extra Terrestrial Intelligence at home". BOINC is developed by the Space Sciences Lab at the University of California, Berkeley, by the same people who created SETI@home. However, instead of being tied to the specific radio telescope application, BOINC is a generic infrastructure by which many different kinds of scientific computations can be solved. The current generation of SETI@home now runs on top of BOINC, along with various physics, biology, climatology, and other applications.

The basic computational model for BOINC and the original SETI@home is the same: volunteers install BOINC client software, called the *boinc_client*, which runs whenever the machine would otherwise be idle. However, the BOINC installation on any given machine must be configured so that it knows what computations to work for instead of always working on a hard coded computation. The BOINC terminology for a computation is a project. A given BOINC client can be configured to donate all of its cycles to a single project, or to split the cycles between projects so that, on average, the desired percentage of the computational power is allocated to each project. Once the *boinc_client* starts running, it attempts to contact a centralized server for each project it has been configured to work for. The BOINC software downloads the appropriate platform-specific application binary and some work units from the central server for each project. Whenever the client software completes a given work unit, it once again attempts to connect to that project's central server to upload the results and download more work.

BOINC participants must register at the centralized server for each project they wish to donate cycles to. The process produces a unique identifier so that the work performed by a given client can be credited to a specific user. BOINC keeps track of the work units completed by each user, so that users providing the most cycles get the highest rankings, and therefore, bragging rights.

Because BOINC already handles the problems of distributing the application binaries for each scientific computation, the work units, and compiling the results, it is a perfect system for managing backfill computations in HTCondor. Many of the applications that run on top of BOINC produce their own application-specific checkpoints, so even if the *boinc_client* is killed, for example, when an HTCondor job arrives at a machine, or if the interactive user returns, an entire work unit will not necessarily be lost.

Installing the BOINC client software

In HTCondor Version 23.0.8, the *boinc_client* must be manually downloaded, installed and configured outside of HTCondor. Download the *boinc_client* executables at http://boinc.berkeley.edu/download.php.

Once the BOINC client software has been downloaded, the *boinc_client* binary should be placed in a location where the HTCondor daemons can use it. The path will be specified with the HTCondor configuration variable .

Additionally, a local directory on each machine should be created where the BOINC system can write files it needs. This directory must not be shared by multiple instances of the BOINC software. This is the same restriction as placed on the `spool` or `execute` directories used by HTCondor. The location of this directory is defined by . The directory must be writable by whatever user the *boinc_client* will run as. This user is either the same as the user the HTCondor daemons are running as, if HTCondor is not running as root, or a user defined via the configuration variable.

Finally, HTCondor administrators wishing to use BOINC for backfill jobs must create accounts at the various BOINC projects they want to donate cycles to. The details of this process vary from project to project. Beware that this step must be done manually, as the *boinc_client* can not automatically register a user at a given project, unlike the more fancy GUI version of the BOINC client software which many users run as a screen saver. For example, to configure machines to perform work for the Einstein@home project (a physics experiment run by the University of Wisconsin at Milwaukee), HTCondor administrators should go to http://einstein.phys.uwm.edu/create_account_form.php, fill in the web form, and generate a new Einstein@home identity. This identity takes the form of a project URL (such as http://einstein.phys.uwm.edu) followed by an account key, which is a long string of letters and numbers that is used as a unique identifier. This URL and account key will be needed when configuring HTCondor to use BOINC for backfill computations.

Configuring the BOINC client under HTCondor

After the *boinc_client* has been installed on a given machine, the BOINC projects to join have been selected, and a unique project account key has been created for each project, the HTCondor configuration needs to be modified.

Whenever the *condor_startd* decides to spawn the *boinc_client* to perform backfill computations, it will spawn a *condor_starter* to directly launch and monitor the *boinc_client* program. This *condor_starter* is just like the one used to invoke any other HTCondor jobs.

This *condor_starter* reads values out of the HTCondor configuration files to define the job it should run, as opposed to getting these values from a job ClassAd in the case of a normal HTCondor job. All of the configuration variables names for variables to control things such as the path to the *boinc_client* binary to use, the command-line arguments, and the initial working directory, are prefixed with the string `"BOINC_"`. Each of these variables is described as either a required or an optional configuration variable.

Required configuration variables:

The full path and executable name of the *boinc_client* binary to use.

The full path to the local directory where BOINC should run.

The HTCondor universe used for running the *boinc_client* program. This must be set to `vanilla` for BOINC to work under HTCondor.

What user the *boinc_client* program should be run as. This variable is only used if the HTCondor daemons are running as root. In this case, the *condor_starter* must be told what user identity to switch to before invoking

the *boinc_client*. This can be any valid user on the local system, but it must have write permission in whatever directory is specified by `BOINC_InitialDir`.

Optional configuration variables:

Command-line arguments that should be passed to the *boinc_client* program. For example, one way to specify the BOINC project to join is to use the **-attach_project** argument to specify a project URL and account key. For example:

```
BOINC_Arguments = --attach_project http://einstein.phys.uwm.edu [account_key]
```

Environment variables that should be set for the *boinc_client*.

Full path to the file where `stdout` from the *boinc_client* should be written. If this variable is not defined, `stdout` will be discarded.

Full path to the file where `stderr` from the *boinc_client* should be written. If this macro is not defined, `stderr` will be discarded.

The following example shows one possible usage of these settings:

```
# Define a shared macro that can be used to define other settings.
# This directory must be manually created before attempting to run
# any backfill jobs.
BOINC_HOME = $(LOCAL_DIR)/boinc

# Path to the boinc_client to use, and required universe setting
BOINC_Executable = /usr/local/bin/boinc_client
BOINC_Universe = vanilla

# What initial working directory should BOINC use?
BOINC_InitialDir = $(BOINC_HOME)

# Where to place stdout and stderr
BOINC_Output = $(BOINC_HOME)/boinc.out
BOINC_Error = $(BOINC_HOME)/boinc.err
```

If the HTCondor daemons reading this configuration are running as root, an additional variable must be defined:

```
# Specify the user that the boinc_client should run as:
BOINC_Owner = nobody
```

In this case, HTCondor would spawn the *boinc_client* as nobody, so the directory specified in `$(BOINC_HOME)` would have to be writable by the nobody user.

A better choice would probably be to create a separate user account just for running BOINC jobs, so that the local BOINC installation is not writable by other processes running as nobody. Alternatively, the `BOINC_Owner` could be set to daemon.

Attaching to a specific BOINC project

There are a few ways to attach an HTCondor/BOINC installation to a given BOINC project:

- Use the **-attach_project** argument to the *boinc_client* program, defined via the `BOINC_Arguments` variable. The *boinc_client* will only accept a single **-attach_project** argument, so this method can only be used to attach

to one project.

- The *boinc_cmd* command-line tool can perform various BOINC administrative tasks, including attaching to a BOINC project. Using *boinc_cmd*, the appropriate argument to use is called **-project_attach**. Unfortunately, the *boinc_client* must be running for *boinc_cmd* to work, so this method can only be used once the HTCondor resource has entered the Backfill state and has spawned the *boinc_client*.
- Manually create account files in the local BOINC directory. Upon start up, the *boinc_client* will scan its local directory (the directory specified with `BOINC_InitialDir`) for files of the form `account_[URL].xml`, for example, `account_einstein.phys.uwm.edu.xml`. Any files with a name that matches this convention will be read and processed. The contents of the file define the project URL and the authentication key. The format is:

```
<account>
  <master_url>[URL]</master_url>
  <authenticator>[key]</authenticator>
</account>
```

For example:

```
<account>
  <master_url>http://einstein.phys.uwm.edu</master_url>
  <authenticator>aaaa1111bbbb2222cccc3333</authenticator>
</account>
```

Of course, the <authenticator> tag would use the real authentication key returned when the account was created at a given project.

These account files can be copied to the local BOINC directory on all machines in an HTCondor pool, so administrators can either distribute them manually, or use symbolic links to point to a shared file system.

In the two cases of using command-line arguments for *boinc_client* or running the *boinc_cmd* tool, BOINC will write out the resulting account file to the local BOINC directory on the machine, and then future invocations of the *boinc_client* will already be attached to the appropriate project(s).

BOINC on Windows

The Windows version of BOINC has multiple installation methods. The preferred method of installation for use with HTCondor is the Shared Installation method. Using this method gives all users access to the executables. During the installation process

1. Deselect the option which makes BOINC the default screen saver
2. Deselect the option which runs BOINC on start up.
3. Do not launch BOINC at the conclusion of the installation.

There are three major differences from the Unix version to keep in mind when dealing with the Windows installation:

1. The Windows executables have different names from the Unix versions. The Windows client is called *boinc.exe*. Therefore, the configuration variable is written:

```
BOINC_Executable = C:\PROGRA~1\BOINC\boinc.exe
```

The Unix administrative tool *boinc_cmd* is called *boinccmd.exe* on Windows.

2. When using BOINC on Windows, the configuration variable will not be respected fully. To work around this difficulty, pass the BOINC home directory directly to the BOINC application via the configuration variable. For Windows, rewrite the argument line as:

```
BOINC_Arguments = --dir $(BOINC_HOME) \
          --attach_project http://einstein.phys.uwm.edu [account_key]
```

As a consequence of setting the BOINC home directory, some projects may fail with the authentication error:

```
Scheduler request failed: Peer
certificate cannot be authenticated
with known CA certificates.
```

To resolve this issue, copy the `ca-bundle.crt` file from the BOINC installation directory to `$(BOINC_HOME)`. This file appears to be project and machine independent, and it can therefore be distributed as part of an automated HTCondor installation.

3. The configuration variable behaves differently on Windows than it does on Unix. Its value may take one of two forms:

   - domain\user
   - user This form assumes that the user exists in the local domain (that is, on the computer itself).

   Setting this option causes the addition of the job attribute

```
RunAsUser = True
```

   to the backfill client. This further implies that the configuration variable be set to `True` to insure that the local *condor_starter* be able to run jobs in this manner. For more information on the `RunAsUser` attribute, see *Executing Jobs as the Submitting User*. For more information on the the `STARTER_ALLOW_RUNAS_OWNER` configuration variable, see *Shared File System Configuration File Macros*.

5.21.6 Per Job PID Namespaces

Per job PID namespaces provide enhanced isolation of one process tree from another through kernel level process ID namespaces. HTCondor may enable the use of per job PID namespaces for Linux RHEL 6, Debian 6, and more recent kernels.

Read about per job PID namespaces http://lwn.net/Articles/531419/.

The needed isolation of jobs from the same user that execute on the same machine as each other is already provided by the implementation of slot users as described in *User Accounts in HTCondor on Unix Platforms*. This is the recommended way to implement the prevention of interference between more than one job submitted by a single user. However, the use of a shared file system by slot users presents issues in the ownership of files written by the jobs.

The per job PID namespace provides a way to handle the ownership of files produced by jobs within a shared file system. It also isolates the processes of a job within its PID namespace. As a side effect and benefit, the clean up of processes for a job within a PID namespace is enhanced. When the process with PID = 1 is killed, the operating system takes care of killing all child processes.

To enable the use of per job PID namespaces, set the configuration to include

```
USE_PID_NAMESPACES = True
```

This configuration variable defaults to `False`, thus the use of per job PID namespaces is disabled by default.

5.21.7 Group ID-Based Process Tracking

One function that HTCondor often must perform is keeping track of all processes created by a job. This is done so that HTCondor can provide resource usage statistics about jobs, and also so that HTCondor can properly clean up any processes that jobs leave behind when they exit.

In general, tracking process families is difficult to do reliably. By default HTCondor uses a combination of process parent-child relationships, process groups, and information that HTCondor places in a job's environment to track process families on a best-effort basis. This usually works well, but it can falter for certain applications or for jobs that try to evade detection.

Jobs that run with a user account dedicated for HTCondor's use can be reliably tracked, since all HTCondor needs to do is look for all processes running using the given account. Administrators must specify in HTCondor's configuration what accounts can be considered dedicated via the setting. See *User Accounts in HTCondor on Unix Platforms* for further details.

Ideally, jobs can be reliably tracked regardless of the user account they execute under. This can be accomplished with group ID-based tracking. This method of tracking requires that a range of dedicated group IDs (GID) be set aside for HTCondor's use. The number of GIDs that must be set aside for an execute machine is equal to its number of execution slots. GID-based tracking is only available on Linux, and it requires that HTCondor daemons run as root.

GID-based tracking works by placing a dedicated GID in the supplementary group list of a job's initial process. Since modifying the supplementary group ID list requires root privilege, the job will not be able to create processes that go unnoticed by HTCondor.

Once a suitable GID range has been set aside for process tracking, GID-based tracking can be enabled via the parameter. The minimum and maximum GIDs included in the range are specified with the and settings. For example, the following would enable GID-based tracking for an execute machine with 8 slots.

```
USE_GID_PROCESS_TRACKING = True
MIN_TRACKING_GID = 750
MAX_TRACKING_GID = 757
```

If the defined range is too small, such that there is not a GID available when starting a job, then the *condor_starter* will fail as it tries to start the job. An error message will be logged stating that there are no more tracking GIDs.

GID-based process tracking requires use of the *condor_procd*. If is true, the *condor_procd* will be used regardless of the setting. Changes to and require a full restart of HTCondor.

5.21.8 Cgroup-Based Process Tracking

A new feature in Linux version 2.6.24 allows HTCondor to more accurately and safely manage jobs composed of sets of processes. This Linux feature is called Control Groups, or cgroups for short, and it is available starting with RHEL 6, Debian 6, and related distributions. Documentation about Linux kernel support for cgroups can be found in the Documentation directory in the kernel source code distribution. Another good reference is http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/index.html

The interface between the kernel cgroup functionality is via a (virtual) file system, usually mounted at `/sys/fs/cgroup`.

If your Linux distribution uses *systemd*, it will mount the cgroup file system, and the only remaining item is to set configuration variable , as described below.

When cgroups are correctly configured and running, the virtual file system mounted on `/sys/fs/cgroup` should have several subdirectories under it, and there should an `htcondor` subdirectory under the directory `/sys/fs/cgroup/cpu`, `/sys/fs/cgroup/memory` and some others.

The *condor_starter* daemon uses cgroups by default on Linux systems to accurately track all the processes started by a job, even when quickly-exiting parent processes spawn many child processes. As with the GID-based tracking, this is only implemented when a *condor_procd* daemon is running.

Kernel cgroups are named in a virtual file system hierarchy. HTCondor will put each running job on the execute node in a distinct cgroup. The name of this cgroup is the name of the execute directory for that *condor_starter*, with slashes replaced by underscores, followed by the name and number of the slot. So, for the memory controller, a job running on slot1 would have its cgroup located at `/sys/fs/cgroup/memory/htcondor/condor_var_lib_condor_execute_slot1/`. The `tasks` file in this directory will contain a list of all the processes in this cgroup, and many other files in this directory have useful information about resource usage of this cgroup. See the kernel documentation for full details.

Once cgroup-based tracking is configured, usage should be invisible to the user and administrator. The *condor_procd* log, as defined by configuration variable , will mention that it is using this method, but no user visible changes should occur, other than the impossibility of a quickly-forking process escaping from the control of the *condor_starter*, and the more accurate reporting of memory usage.

A cgroup-enabled HTCondor will install and handle a per-job (not per-process) Linux Out of Memory killer (OOM-Killer). When a job exceeds the memory provisioned by the *condor_startd*, the Linux kernel will send an OOM message to the *condor_starter*, and HTCondor will evict the job, and put it on hold. Sometimes, even when the job's memory usage is below the provisioned amount, if other, non-HTCondor processes, on the system are using too much memory, the linux kernel may choose to OOM-kill the job. In this case, HTCondor will log a message and evict the job, mark it as idle, so it can start again somewhere else.

5.21.9 Limiting Resource Usage Using Cgroups

While the method described to limit a job's resource usage is portable, and it should run on any Linux or BSD or Unix system, it suffers from one large flaw. The flaw is that resource limits imposed are per process, not per job. An HTCondor job is often composed of many Unix processes. If the method of limiting resource usage with a user job wrapper is used to impose a 2 Gigabyte memory limit, that limit applies to each process in the job individually. If a job created 100 processes, each using just under 2 Gigabytes, the job would continue without the resource limits kicking in. Clearly, this is not what the machine owner intends. Moreover, the memory limit only applies to the virtual memory size, not the physical memory size, or the resident set size. This can be a problem for jobs that use the `mmap` system call to map in a large chunk of virtual memory, but only need a small amount of memory at one time. Typically, the resource the administrator would like to control is physical memory, because when that is in short supply, the machine starts paging, and can become unresponsive very quickly.

The *condor_starter* can, using the Linux cgroup capability, apply resource limits collectively to sets of jobs, and apply limits to the physical memory used by a set of processes. The main downside of this technique is that it is only available on relatively new Unix distributions such as RHEL 6 and Debian 6. This technique also may require editing of system configuration files.

To enable cgroup-based limits, first ensure that cgroup-based tracking is enabled, as it is by default on supported systems, as described in section *3.14.13*. Once set, the *condor_starter* will create a cgroup for each job, and set attributes in that cgroup to control memory and cpu usage. These attributes are the cpu.shares attribute in the cpu controller, and two attributes in the memory controller, both memory.limit_in_bytes, and memory.soft_limit_in_bytes. The configuration variable controls this. If is set to the string `hard`, the hard limit will be set to the slot size, and the soft limit to 90% of the slot size.. If set to `soft`, the soft limit will be set to the slot size and the hard limit will be set to the memory size of the whole startd. By default, this whole size is the detected memory the size, minus RESERVED_MEMORY. Or, if is defined, that value is used..

No limits will be set if the value is `none`. The default is `none`. If the hard limit is in force, then the total amount of physical memory used by the sum of all processes in this job will not be allowed to exceed the limit. If the process goes above the hard limit, the job will be put on hold.

The memory size used in both cases is the machine ClassAd attribute `Memory`. Note that `Memory` is a static amount when using static slots, but it is dynamic when partitionable slots are used. That is, the limit is whatever the "Mem" column of condor_status reports for that slot.

If is set, HTCondor will also also use cgroups to limit the amount of swap space used by each job. By default, the maximum amount of swap space used by each slot is the total amount of Virtual Memory in the slot, minus the amount of physical memory. Note that HTCondor measures virtual memory in kbytes, and physical memory in megabytes. To prevent jobs with high memory usage from thrashing and excessive paging, and force HTCondor to put them on hold instead, you can tell condor that a job should never use swap, by setting DISABLE_SWAP_FOR_JOB to true (the default is false).

In addition to memory, the *condor_starter* can also control the total amount of CPU used by all processes within a job. To do this, it writes a value to the cpu.shares attribute of the cgroup cpu controller. The value it writes is copied from the `Cpus` attribute of the machine slot ClassAd multiplied by 100. Again, like the `Memory` attribute, this value is fixed for static slots, but dynamic under partitionable slots. This tells the operating system to assign cpu usage proportionally to the number of cpus in the slot. Unlike memory, there is no concept of `soft` or `hard`, so this limit only applies when there is contention for the cpu. That is, on an eight core machine, with only a single, one-core slot running, and otherwise idle, the job running in the one slot could consume all eight cpus concurrently with this limit in play, if it is the only thing running. If, however, all eight slots where running jobs, with each configured for one cpu, the cpu usage would be assigned equally to each job, regardless of the number of processes or threads in each job.

5.21.10 Concurrency Limits

Concurrency limits allow an administrator to limit the number of concurrently running jobs that declare that they use some pool-wide resource. This limit is applied globally to all jobs submitted from all schedulers across one HTCondor pool; the limits are not applied to scheduler, local, or grid universe jobs. This is useful in the case of a shared resource, such as an NFS or database server that some jobs use, where the administrator needs to limit the number of jobs accessing the server.

The administrator must predefine the names and capacities of the resources to be limited in the negotiator's configuration file. The job submitter must declare in the submit description file which resources the job consumes.

The administrator chooses a name for the limit. Concurrency limit names are case-insensitive. The names are formed from the alphabet letters 'A' to 'Z' and 'a' to 'z', the numerical digits 0 to 9, the underscore character '_' , and at most one period character. The names cannot start with a numerical digit.

For example, assume that there are 3 licenses for the X software, so HTCondor should constrain the number of running jobs which need the X software to 3. The administrator picks XSW as the name of the resource and sets the configuration

```
XSW_LIMIT = 3
```

where `XSW` is the invented name of this resource, and this name is appended with the string `_LIMIT`. With this limit, a maximum of 3 jobs declaring that they need this resource may be executed concurrently.

In addition to named limits, such as in the example named limit `XSW`, configuration may specify a concurrency limit for all resources that are not covered by specifically-named limits. The configuration variable sets this value. For example,

```
CONCURRENCY_LIMIT_DEFAULT = 1
```

will enforce a limit of at most 1 running job that declares a usage of an unnamed resource. If is omitted from the configuration, then no limits are placed on the number of concurrently executing jobs for which there is no specifically-named concurrency limit.

The job must declare its need for a resource by placing a command in its submit description file or adding an attribute to the job ClassAd. In the submit description file, an example job that requires the X software adds:

```
concurrency_limits = XSW
```

This results in the job ClassAd attribute

```
ConcurrencyLimits = "XSW"
```

Jobs may declare that they need more than one type of resource. In this case, specify a comma-separated list of resources:

```
concurrency_limits = XSW, DATABASE, FILESERVER
```

The units of these limits are arbitrary. This job consumes one unit of each resource. Jobs can declare that they use more than one unit with syntax that follows the resource name by a colon character and the integer number of resources. For example, if the above job uses three units of the file server resource, it is declared with

```
concurrency_limits = XSW, DATABASE, FILESERVER:3
```

If there are sets of resources which have the same capacity for each member of the set, the configuration may become tedious, as it defines each member of the set individually. A shortcut defines a name for a set. For example, define the sets called LARGE and SMALL:

```
CONCURRENCY_LIMIT_DEFAULT = 5
CONCURRENCY_LIMIT_DEFAULT_LARGE = 100
CONCURRENCY_LIMIT_DEFAULT_SMALL = 25
```

To use the set name in a concurrency limit, the syntax follows the set name with a period and then the set member's name. Continuing this example, there may be a concurrency limit named LARGE.SWLICENSE, which gets the capacity of the default defined for the LARGE set, which is 100. A concurrency limit named LARGE.DBSESSION will also have a limit of 100. A concurrency limit named OTHER.LICENSE will receive the default limit of 5, as there is no set named OTHER.

A concurrency limit may be evaluated against the attributes of a matched machine. This allows a job to vary what concurrency limits it requires based on the machine to which it is matched. To implement this, the job uses submit command **concurrency_limits_expr** instead of **concurrency_limits** . Consider an example in which execute machines are located on one of two local networks. The administrator sets a concurrency limit to limit the number of network intensive jobs on each network to 10. Configuration of each execute machine advertises which local network it is on. A machine on "NETWORK_A" configures

```
NETWORK = "NETWORK_A"
STARTD_ATTRS = $(STARTD_ATTRS) NETWORK
```

and a machine on "NETWORK_B" configures

```
NETWORK = "NETWORK_B"
STARTD_ATTRS = $(STARTD_ATTRS) NETWORK
```

The configuration for the negotiator sets the concurrency limits:

```
NETWORK_A_LIMIT = 10
NETWORK_B_LIMIT = 10
```

Each network intensive job identifies itself by specifying the limit within the submit description file:

```
concurrency_limits_expr = TARGET.NETWORK
```

The concurrency limit is applied based on the network of the matched machine.

An extension of this example applies two concurrency limits. One limit is the same as in the example, such that it is based on an attribute of the matched machine. The other limit is of a specialized application called "SWX" in this example. The negotiator configuration is extended to also include

```
SWX_LIMIT = 15
```

The network intensive job that also uses two units of the SWX application identifies the needed resources in the single submit command:

```
concurrency_limits_expr = strcat("SWX:2 ", TARGET.NETWORK)
```

Submit command **concurrency_limits_expr** may not be used together with submit command **concurrency_limits**.

Note that it is possible, under unusual circumstances, for more jobs to be started than should be allowed by the concurrency limits feature. In the presence of preemption and dropped updates from the *condor_startd* daemon to the *condor_collector* daemon, it is possible for the limit to be exceeded. If the limits are exceeded, HTCondor will not kill any job to reduce the number of running jobs to meet the limit.

5.21.11 The VM Universe

vm universe jobs may be executed on any execution site with Xen (via *libvirt*) or KVM. To do this, HTCondor must be informed of some details of the virtual machine installation, and the execution machines must be configured correctly.

What follows is not a comprehensive list of the options that help set up to use the **vm** universe; rather, it is intended to serve as a starting point for those users interested in getting **vm** universe jobs up and running quickly. Details of configuration variables are in the *Configuration File Entries Relating to Virtual Machines* section.

Begin by installing the virtualization package on all execute machines, according to the vendor's instructions. We have successfully used Xen and KVM.

For Xen, there are three things that must exist on an execute machine to fully support **vm** universe jobs.

1. A Xen-enabled kernel must be running. This running Xen kernel acts as Dom0, in Xen terminology, under which all VMs are started, called DomUs Xen terminology.
2. The *libvirtd* daemon must be available, and *Xend* services must be running.
3. The *pygrub* program must be available, for execution of VMs whose disks contain the kernel they will run.

For KVM, there are two things that must exist on an execute machine to fully support **vm** universe jobs.

1. The machine must have the KVM kernel module installed and running.
2. The *libvirtd* daemon must be installed and running.

Configuration is required to enable the execution of **vm** universe jobs. The type of virtual machine that is installed on the execute machine must be specified with the variable. For now, only one type can be utilized per machine. For instance, the following tells HTCondor to use KVM:

```
VM_TYPE = kvm
```

The location of the *condor_vm-gahp* and its log file must also be specified on the execute machine. On a Windows installation, these options would look like this:

```
VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp.exe
VM_GAHP_LOG = $(LOG)/VMGahpLog
```

Xen-Specific and KVM-Specific Configuration

Once the configuration options have been set, restart the *condor_startd* daemon on that host. For example:

```
$ condor_restart -startd leovinus
```

The *condor_startd* daemon takes a few moments to exercise the VM capabilities of the *condor_vm-gahp*, query its properties, and then advertise the machine to the pool as VM-capable. If the set up succeeded, then *condor_status* will reveal that the host is now VM-capable by printing the VM type and the version number:

```
$ condor_status -vm leovinus
```

After a suitable amount of time, if this command does not give any output, then the *condor_vm-gahp* is having difficulty executing the VM software. The exact cause of the problem depends on the details of the VM, the local installation, and a variety of other factors. We can offer only limited advice on these matters:

For Xen and KVM, the **vm** universe is only available when root starts HTCondor. This is a restriction currently imposed because root privileges are required to create a virtual machine on top of a Xen-enabled kernel. Specifically, root is needed to properly use the *libvirt* utility that controls creation and management of Xen and KVM guest virtual machines. This restriction may be lifted in future versions, depending on features provided by the underlying tool *libvirt*.

When a vm Universe Job Fails to Start

If a vm universe job should fail to launch, HTCondor will attempt to distinguish between a problem with the user's job description, and a problem with the virtual machine infrastructure of the matched machine. If the problem is with the job, the job will go on hold with a reason explaining the problem. If the problem is with the virtual machine infrastructure, HTCondor will reschedule the job, and it will modify the machine ClassAd to prevent any other vm universe job from matching. vm universe configuration is not slot-specific, so this change is applied to all slots.

When the problem is with the virtual machine infrastructure, these machine ClassAd attributes are changed:

- `HasVM` will be set to `False`
- `VMOfflineReason` will be set to a somewhat explanatory string
- `VMOfflineTime` will be set to the time of the failure
- `OfflineUniverses` will be adjusted to include `"VM"` and `13`

Since *condor_submit* adds `HasVM == True` to a vm universe job's requirements, no further vm universe jobs will match.

Once any problems with the infrastructure are fixed, to change the machine ClassAd attributes such that the machine will once again match to vm universe jobs, an administrator has three options. All have the same effect of setting the machine ClassAd attributes to the correct values such that the machine will not reject matches for vm universe jobs.

1. Restart the *condor_startd* daemon.
2. Submit a vm universe job that explicitly matches the machine. When the job runs, the code detects the running job and causes the attributes related to the vm universe to be set indicating that vm universe jobs can match with this machine.
3. Run the command line tool *condor_update_machine_ad* to set machine ClassAd attribute `HasVM` to `True`, and this will cause the other attributes related to the vm universe to be set indicating that vm universe jobs can match with this machine. See the *condor_update_machine_ad* manual page for examples and details.

5.21.12 Using HTCondor with AFS

Configuration variables that allow machines to interact with and use a shared file system are given at the *Shared File System Configuration File Macros* section.

Limitations with AFS occur because HTCondor does not currently have a way to authenticate itself to AFS. This is true of the HTCondor daemons that would like to authenticate as the AFS user condor, and of the *condor_shadow* which would like to authenticate as the user who submitted the job it is serving. Since neither of these things can happen yet, there are special things to do when interacting with AFS. Some of this must be done by the administrator(s) installing HTCondor. Other things must be done by HTCondor users who submit jobs.

AFS and HTCondor for Users

The *condor_shadow* daemon runs on the machine where jobs are submitted. It performs all file system access on behalf of the jobs. Because the *condor_shadow* daemon is not authenticated to AFS as the user who submitted the job, the *condor_shadow* daemon will not normally be able to write any output. Therefore the directories in which the job will be creating output files will need to be world writable; they need to be writable by non-authenticated AFS users. In addition, the program's `stdout`, `stderr`, log file, and any file the program explicitly opens will need to be in a directory that is world-writable.

An administrator may be able to set up special AFS groups that can make unauthenticated access to the program's files less scary. For example, there is supposed to be a way for AFS to grant access to any unauthenticated process on a given host. If set up, write access need only be granted to unauthenticated processes on the access point, as opposed to any unauthenticated process on the Internet. Similarly, unauthenticated read access could be granted only to processes running on the access point.

A solution to this problem is to not use AFS for output files. If disk space on the access point is available in a partition not on AFS, submit the jobs from there. While the *condor_shadow* daemon is not authenticated to AFS, it does run with the effective UID of the user who submitted the jobs. So, on a local (or NFS) file system, the *condor_shadow* daemon will be able to access the files, and no special permissions need be granted to anyone other than the job submitter. If the HTCondor daemons are not invoked as root however, the *condor_shadow* daemon will not be able to run with the submitter's effective UID, leading to a similar problem as with files on AFS.

AFS and HTCondor for Administrators

The largest result from the lack of authentication with AFS is that the directory defined by the configuration variable and its subdirectories `log` and `spool` on each machine must be either writable to unauthenticated users, or must not be on AFS. Making these directories writable a very bad security hole, so it is not a viable solution. Placing onto NFS is acceptable. To avoid AFS, place the directory defined for on a local partition on each machine in the pool. This implies running *condor_configure* to install the release directory and configure the pool, setting the variable to a local partition. When that is complete, log into each machine in the pool, and run *condor_init* to set up the local HTCondor directory.

The directory defined by , which holds all the HTCondor binaries, libraries, and scripts, can be on AFS. None of the HTCondor daemons need to write to these files. They only need to read them. So, the directory defined by only needs to be world readable in order to let HTCondor function. This makes it easier to upgrade the binaries to a newer version at a later date, and means that users can find the HTCondor tools in a consistent location on all the machines in the pool. Also, the HTCondor configuration files may be placed in a centralized location.

Finally, consider setting up some targeted AFS groups to help users deal with HTCondor and AFS better. This is discussed in the following manual subsection. In short, create an AFS group that contains all users, authenticated or not, but which is restricted to a given host or subnet. These should be made as host-based ACLs with AFS, but here at UW-Madison, we have had some trouble getting that working. Instead, we have a special group for all machines in

our department. The users here are required to make their output directories on AFS writable to any process running on any of our machines, instead of any process on any machine with AFS on the Internet.

CHAPTER

SIX

CLASSADS

This chapter presents HTCondor's ClassAd mechanism in three parts.

The first part may be of interest to advanced job submitters as well as HTCondor administrators: it describes how to write ClassAds and ClassAd expressions, including details of the ClassAd language syntax, evaluation semantics, and its built-in functions.

The second part is likely only of interest to HTCondor administrators: it describes the generic mechanism provided by HTCondor to transform ClassAds, as used in the schedd and the job routers, and as available via a command-line tool.

The third part describes how to format ClassAds for printing from command-line tools like *condor_q*, *condor_history*, and *condor_status*. Advanced users may specify their own custom formats, or adminstrators may set custom defaults.

6.1 HTCondor's ClassAd Mechanism

ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the HTCondor system. ClassAds are used extensively in the HTCondor system to represent jobs, resources, submitters and other HTCondor daemons. An understanding of this mechanism is required to harness the full flexibility of the HTCondor system.

A ClassAd is a set of uniquely named expressions. Each named expression is called an attribute. The following shows ten attributes, a portion of an example ClassAd.

```
MyType       = "Machine"
TargetType   = "Job"
Machine      = "froth.cs.wisc.edu"
Arch         = "INTEL"
OpSys        = "LINUX"
Disk         = 35882
Memory       = 128
KeyboardIdle = 173
LoadAvg      = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60
```

ClassAd expressions look very much like expressions in C, and are composed of literals and attribute references composed with operators and functions. The difference between ClassAd expressions and C expressions arise from the fact that ClassAd expressions operate in a much more dynamic environment. For example, an expression from a machine's ClassAd may refer to an attribute in a job's ClassAd, such as TARGET.Owner in the above example. The value and type of the attribute is not known until the expression is evaluated in an environment which pairs a specific job ClassAd with the machine ClassAd.

ClassAd expressions handle these uncertainties by defining all operators to be total operators, which means that they have well defined behavior regardless of supplied operands. This functionality is provided through two distinguished values, `UNDEFINED` and `ERROR`, and defining all operators so that they can operate on all possible values in the ClassAd system. For example, the multiplication operator which usually only operates on numbers, has a well defined behavior if supplied with values which are not meaningful to multiply. Thus, the expression 10 * "A string" evaluates to the value `ERROR`. Most operators are strict with respect to `ERROR`, which means that they evaluate to `ERROR` if any of their operands are `ERROR`. Similarly, most operators are strict with respect to `UNDEFINED`.

6.1.1 ClassAds: Old and New

ClassAds have existed for quite some time in two forms: Old and New. Old ClassAds were the original form and were used in HTCondor until HTCondor version 7.5.0. They were heavily tied to the HTCondor development libraries. New ClassAds added new features and were designed as a stand-alone library that could be used apart from HTCondor.

In HTCondor version 7.5.1, HTCondor switched to using the New ClassAd library for all use of ClassAds within HTCondor. The library is placed into a compatibility mode so that HTCondor 7.5.1 is still able to exchange ClassAds with older versions of HTCondor.

All user interaction with tools (such as *condor_q*) as well as output of tools is still compatible with Old ClassAds. Before HTCondor version 7.5.1, New ClassAds were used only in the Job Router. There are some syntax and behavior differences between Old and New ClassAds, all of which should remain invisible to users of HTCondor.

A complete description of New ClassAds can be found at http://htcondor.org/classad/classad.html, and in the ClassAd Language Reference Manual found on that web page.

Some of the features of New ClassAds that are not in Old ClassAds are lists, nested ClassAds, time values, and matching groups of ClassAds. HTCondor has avoided using these features, as using them makes it difficult to interact with older versions of HTCondor. But, users can start using them if they do not need to interact with versions of HTCondor older than 7.5.1.

The syntax varies slightly between Old and New ClassAds. Here is an example ClassAd presented in both forms. The Old form:

```
Foo = 3
Bar = "ab\"cd\ef"
Moo = Foo =!= Undefined
```

The New form:

```
[
Foo = 3;
Bar = "ab\"cd\\ef";
Moo = Foo isnt Undefined;
]
```

HTCondor will convert to and from Old ClassAd syntax as needed.

New ClassAd Attribute References

Expressions often refer to ClassAd attributes. These attribute references work differently in Old ClassAds as compared with New ClassAds. In New ClassAds, an unscoped reference is looked for only in the local ClassAd. An unscoped reference is an attribute that does not have a `MY.` or `TARGET.` prefix. The local ClassAd may be described by an example. Matchmaking uses two ClassAds: the job ClassAd and the machine ClassAd. The job ClassAd is evaluated to see if it is a match for the machine ClassAd. The job ClassAd is the local ClassAd. Therefore, in the `Requirements` attribute of the job ClassAd, any attribute without the prefix `TARGET.` is looked up only in the job ClassAd. With New ClassAd evaluation, the use of the prefix `MY.` is eliminated, as an unscoped reference can only refer to the local ClassAd.

The `MY.` and `TARGET.` scoping prefixes only apply when evaluating an expression within the context of two ClassAds. Two examples that exemplify this are matchmaking and machine policy evaluation. When evaluating an expression within the context of a single ClassAd, `MY.` and `TARGET.` are not defined. Using them within the context of a single ClassAd will result in a value of `Undefined`. Two examples that exemplify evaluating an expression within the context of a single ClassAd are during user job policy evaluation, and with the **-constraint** option to command-line tools.

New ClassAds have no `CurrentTime` attribute. If needed, use the time() function instead. In order to mimic Old ClassAd semantics in current versions of HTCondor, all ClassAds have an implicit `CurrentTime` attribute, with a value of time().

In current versions of HTCondor, New ClassAds will mimic the evaluation behavior of Old ClassAds. No configuration variables or submit description file contents should need to be changed. To eliminate this behavior and use only the semantics of New ClassAds, set the configuration variable `STRICT_CLASSAD_EVALUATION` to `True`. This permits testing expressions to see if any adjustment is required, before a future version of HTCondor potentially makes New ClassAds evaluation behavior the default or the only option.

6.1.2 ClassAd Syntax

ClassAd expressions are formed by composing literals, attribute references and other sub-expressions with operators and functions.

Composing Literals

Literals in the ClassAd language may be of integer, real, string, undefined or error types. The syntax of these literals is as follows:

Integer
: A sequence of continuous digits (i.e., [0-9]). Additionally, the keywords TRUE and FALSE (case insensitive) are syntactic representations of the integers 1 and 0 respectively.

Real
: Two sequences of continuous digits separated by a period (i.e., [0-9]+.[0-9]+).

String
: A double quote character, followed by a list of characters terminated by a double quote character. A backslash character inside the string causes the following character to be considered as part of the string, irrespective of what that character is.

Undefined
: The keyword `UNDEFINED` (case insensitive) represents the `UNDEFINED` value.

Error
: The keyword `ERROR` (case insensitive) represents the `ERROR` value.

Attributes

Every expression in a ClassAd is named by an attribute name. Together, the (name,expression) pair is called an attribute. An attribute may be referred to in other expressions through its attribute name.

Attribute names are sequences of alphabetic characters, digits and underscores, and may not begin with a digit. All characters in the name are significant, but case is not significant. Thus, Memory, memory and MeMoRy all refer to the same attribute.

An attribute reference consists of the name of the attribute being referenced, and an optional scope resolution prefix. The prefixes that may be used are `MY.` and `TARGET.`. The case used for these prefixes is not significant. The semantics of supplying a prefix are discussed in *ClassAd Evaluation Semantics*.

Expression Operators

The operators that may be used in ClassAd expressions are similar to those available in C. The available operators and their relative precedence is shown in the following example:

```
- (unary negation)   (high precedence)
*  /   %
+   - (addition, subtraction)
<   <=   >=   >
==  !=  =?=  is  =!=  isnt
&&
||                   (low precedence)
```

The operator with the highest precedence is the unary minus operator. The only operators which are unfamiliar are the =?=, is, =!= and isnt operators, which are discussed in *ClassAd Evaluation Semantics*.

Predefined Functions

Any ClassAd expression may utilize predefined functions. Function names are case insensitive. Parameters to functions and a return value from a function may be typed (as given) or not. Nested or recursive function calls are allowed.

Here are descriptions of each of these predefined functions. The possible types are the same as itemized in *ClassAd Syntax*. Where the type may be any of these literal types, it is called out as AnyType. Where the type is Integer, but only returns the value 1 or 0 (implying `True` or `False`), it is called out as Boolean. The format of each function is given as

```
ReturnType FunctionName(ParameterType parameter1, ParameterType parameter2, ...)
```

Optional parameters are given within square brackets.

`AnyType eval(AnyType Expr)`
: Evaluates `Expr` as a string and then returns the result of evaluating the contents of the string as a ClassAd expression. This is useful when referring to an attribute such as `slotX_State` where `X`, the desired slot number is an expression, such as `SlotID+10`. In such a case, if attribute `SlotID` is 5, the value of the attribute `slot15_State` can be referenced using the expression `eval(strcat("slot", SlotID+10,"_State"))`. Function strcat() calls function string() on the second parameter, which evaluates the expression, and then converts the integer

result 15 to the string "15". The concatenated string returned by strcat() is "slot15_State", and this string is then evaluated.

Note that referring to attributes of a job from within the string passed to eval() in the Requirements or Rank expressions could cause inaccuracies in HTCondor's automatic auto-clustering of jobs into equivalent groups for matchmaking purposes. This is because HTCondor needs to determine which ClassAd attributes are significant for matchmaking purposes, and indirect references from within the string passed to eval() will not be counted.

String unparse(Attribute attr)
This function looks up the value of the provided attribute and returns the unparsed version as a string. The attribute's value is not evaluated. If the attribute's value is x + 3, then the function would return the string "x + 3". If the provided attribute cannot be found, an empty string is returned.

This function returns ERROR if other than exactly 1 argument is given or the argument is not an attribute reference.

String unresolved(Attribute attr)
This function returns the external attribute references and unresolved attribute references of the expression that is the value of the provided attribute. If the provided attribute cannot be found, then undefined is returned.

For example, in a typical job ClassAd if the Requirements expression has the value OpSys == "LINUX" && TARGET.Arch == "ARM" && Cpus >= RequestCpus, then unresolved(Requirements) will return "Arch,Cpus,OpSys" because those will not be attributes of the job ClassAd.

Boolean unresolved(Attribute attr, String pattern)
This function returns True when at least one of the external or unresolved attribute references of the expression that is the value of the provided attribute matches the given Perl regular expression pattern. If none of the references match the pattern, then False is returned. If the provided attribute cannot be found, then undefined is returned.

For example, in a typical job ClassAd if the Requirements expression has the value OpSys == "LINUX" && Arch == "ARM", then unresolved(Requirements, "^OpSys") will return True, and unresolved(Requirements, "OpSys.+") will return False.

The intended use of this function is to make it easier to apply a submit transform to a job only when the job does not already reference a certain attribute. For instance

```
JOB_TRANSFORM_DefPlatform @=end
   # Apply this transform only when the job requirements does not reference OpSysAndVer␣
↪or OpSysName
   REQUIREMENTS ! unresolved(Requirements, "OpSys.+")
   # Add a clause to the job requirements to match only CentOs7 machines
   SET Requirements $(MY.Requirements) && OpSysAndVer == "CentOS7"
@end
```

AnyType ifThenElse(AnyType IfExpr,AnyType ThenExpr, AnyType ElseExpr)
A conditional expression is described by IfExpr. The following defines return values, when IfExpr evaluates to

- True. Evaluate and return the value as given by ThenExpr.
- False. Evaluate and return the value as given by ElseExpr.
- UNDEFINED. Return the value UNDEFINED.
- ERROR. Return the value ERROR.
- 0.0. Evaluate, and return the value as given by ElseExpr.
- non-0.0 Real values. Evaluate, and return the value as given by ThenExpr.

Where IfExpr evaluates to give a value of type String, the function returns the value ERROR. The implementation uses lazy evaluation, so expressions are only evaluated as defined.

This function returns ERROR if other than exactly 3 arguments are given.

Boolean isUndefined(AnyType Expr)
Returns True, if Expr evaluates to UNDEFINED. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isError(AnyType Expr)
Returns True, if Expr evaluates to ERROR. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isString(AnyType Expr)
Returns True, if the evaluation of Expr gives a value of type String. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isInteger(AnyType Expr)
Returns True, if the evaluation of Expr gives a value of type Integer. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isReal(AnyType Expr)
Returns True, if the evaluation of Expr gives a value of type Real. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isList(AnyType Expr)
Returns True, if the evaluation of Expr gives a value of type List. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isClassAd(AnyType Expr)
Returns True, if the evaluation of Expr gives a value of type ClassAd. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isBoolean(AnyType Expr)
Returns True, if the evaluation of Expr gives the integer value 0 or 1. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isAbstime(AnyType Expr)
Returns True, if the evaluation of Expr returns an abstime type. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean isReltime(AnyType Expr)
Returns True, if the evaluation of Expr returns an relative time type. Returns False in all other cases.

This function returns ERROR if other than exactly 1 argument is given.

Boolean member(AnyType m, ListType l)
Returns error if m does not evalute to a scalar, or l does not evaluate to a list. Otherwise the elements of l are evaluted in order, and if an element is equal to m in the sense of == the result of the function is True. Otherwise the function returns false.

Boolean anyCompare(string op, list l, AnyType t)
Returns error if op does not evalute to one of <, <=, ==, >, >=, !-, is or isnt. Returns error if l isn't a list,

or t isn't a scalar Otherwise the elements of l are evaluted and compared to t using the corresponding operator defined by op. If any of the members of l evaluate to true, the result is `True`. Otherwise the function returns `False`.

Boolean allCompare(string op, list l, AnyType t)
Returns error if op does not evalute to one of <, <=, ==, >, >=, `!-`, `is` or `isnt`. Returns error if l isn't a list, or t isn't a scalar Otherwise the elements of l are evaluted and compared to t using the corresponding operator defined by op. If all of the members of l evaluate to true, the result is `True`. Otherwise the function returns `False`.

Boolean IdenticalMember(AnyType m, ListType l)
Returns error if m does not evalute to a scalar, or l does not evaluate to a list. Otherwise the elements of l are evaluted in order, and if an element is equal to m in the sense of =?= the result of the function is `True`. Otherwise the function returns false.

Integer int(AnyType Expr)
Returns the integer value as defined by `Expr`. Where the type of the evaluated `Expr` is `Real`, the value is truncated (round towards zero) to an integer. Where the type of the evaluated `Expr` is `String`, the string is converted to an integer using a C-like atoi() function. When this result is not an integer, `ERROR` is returned. Where the evaluated `Expr` is `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Real real(AnyType Expr)
Returns the real value as defined by `Expr`. Where the type of the evaluated `Expr` is `Integer`, the return value is the converted integer. Where the type of the evaluated `Expr` is `String`, the string is converted to a real value using a C-like atof() function. When this result is not a real, `ERROR` is returned. Where the evaluated `Expr` is `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

String string(AnyType Expr)
Returns the string that results from the evaluation of `Expr`. Converts a non-string value to a string. Where the evaluated `Expr` is `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Bool bool(AnyType Expr)
Returns the boolean that results from the evaluation of `Expr`. Converts a non-boolean value to a bool. A string expression that evaluates to the string "true" yields true, and "false" returns

This function returns `ERROR` if other than exactly 1 argument is given.

AbsTime absTime(AnyType t [, int z])
Creates an AbsTime value corresponding to time t an time-zone offset z. If t is a String, then z must be omitted, and t is parsed as a specification as follows.

The operand t is parsed as a specification of an instant in time (date and time). This function accepts the canonical native representation of AbsTime values, but minor variations in format are allowed. The default format is yyyy-mm-ddThh:mm:sszzzzz where zzzzz is a time zone in the format +hh:mm or -hh:mm

If t and z are both omitted, the result is an AbsTime value representing the time and place where the function call is evaluated. Otherwise, t is converted to a Real by the function "real", and treated as a number of seconds from the epoch, Midnight January 1, 1970 UTC. If z is specified, it is treated as a number of seconds east of Greenwich. Otherwise, the offset is calculated from t according to the local rules for the place where the function is evaluated.

RelTime relTime(AnyType t)

If the operand t is a String, it is parsed as a specification of a time interval. This function accepts the canonical native representation of RelTime values, but minor variations in format are allowed.

Otherwise, t is converted to a Real by the function real, and treated as a number of seconds. The default string format is [-]days+hh:mm:ss.fff, where leading components and the fraction .fff are omitted if they are zero. In the default syntax, days is a sequence of digits starting with a non-zero digit, hh, mm, and ss are strings of exactly two digits (padded on the left with zeros if necessary) with values less than 24, 60, and 60, respectively and fff is a string of exactly three digits.

Integer floor(AnyType Expr)

Returns the integer that results from the evaluation of `Expr`, where the type of the evaluated `Expr` is `Integer`. Where the type of the evaluated `Expr` is not `Integer`, function `real(Expr)` is called. Its return value is then used to return the largest magnitude integer that is not larger than the returned value. Where `real(Expr)` returns `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Integer ceiling(AnyType Expr)

Returns the integer that results from the evaluation of `Expr`, where the type of the evaluated `Expr` is `Integer`. Where the type of the evaluated `Expr` is not `Integer`, function `real(Expr)` is called. Its return value is then used to return the smallest magnitude integer that is not less than the returned value. Where `real(Expr)` returns `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Integer pow(Integer base, Integer exponent) OR Real pow(Integer base, Integer exponent) OR Real pow(Real base, Real exponent)

Calculates `base` raised to the power of `exponent`. If `exponent` is an integer value greater than or equal to 0, and `base` is an integer, then an integer value is returned. If `exponent` is an integer value less than 0, or if either `base` or `exponent` is a real, then a real value is returned. An invocation with `exponent=0` or `exponent=0.0`, for any value of `base`, including 0 or 0.0, returns the value 1 or 1.0, type appropriate.

Integer quantize(AnyType a, Integer b) OR Real quantize(AnyType a, Real b) OR AnyType quantize(AnyType a, AnyType list b)

`quantize()` computes the quotient of `a/b`, in order to further compute `` ceiling(quotient) * b``. This computes and returns an integral multiple of `b` that is at least as large as `a`. So, when `b >= a`, the return value will be `b`. The return type is the same as that of `b`, where `b` is an Integer or Real.

When `b` is a list, `quantize()` returns the first value in the list that is greater than or equal to `a`. When no value in the list is greater than or equal to `a`, this computes and returns an integral multiple of the last member in the list that is at least as large as `a`.

This function returns `ERROR` if `a` or `b`, or a member of the list that must be considered is not an Integer or Real.

Here are examples:

```
8     = quantize(3, 8)
4     = quantize(3, 2)
0     = quantize(0, 4)
6.8   = quantize(1.5, 6.8)
7.2   = quantize(6.8, 1.2)
10.2  = quantize(10, 5.1)

4     = quantize(0, {4})
```

(continues on next page)

(continued from previous page)

```
2     = quantize(2, {1, 2, "A"})
3.0   = quantize(3, {1, 2, 0.5})
3.0   = quantize(2.7, {1, 2, 0.5})
ERROR = quantize(3, {1, 2, "A"})
```

Integer round(AnyType Expr)
Returns the integer that results from the evaluation of `Expr`, where the type of the evaluated `Expr` is `Integer`. Where the type of the evaluated `Expr` is not `Integer`, function `real(Expr)` is called. Its return value is then used to return the integer that results from a round-to-nearest rounding method. The nearest integer value to the return value is returned, except in the case of the value at the exact midpoint between two integer values. In this case, the even valued integer is returned. Where `real(Expr)` returns `ERROR` or `UNDEFINED`, or the integer value does not fit into 32 bits, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Integer random([AnyType Expr])
Where the optional argument `Expr` evaluates to type `Integer` or type `Real` (and called `x`), the return value is the integer or real `r` randomly chosen from the interval `0 <= r < x`. With no argument, the return value is chosen with `random(1.0)`. Returns `ERROR` in all other cases.

This function returns `ERROR` if greater than 1 argument is given.

Number sum([List l])
The elements of l are evaluated, producing a list l of values. Undefined values are removed. If the resulting l is composed only of numbers, the result is the sum of the values, as a Real if any value is Real, and as an Integer otherwise. If the list is empty, the result is 0. If the list has only Undefined values, the result is `UNDEFINED`. In other cases, the result is `ERROR`.

This function returns `ERROR` if greater than 1 argument is given.

Number avg([List l])
The elements of l are evaluated, producing a list l of values. Undefined values are removed. If the resulting l is composed only of numbers, the result is the average of the values, as a Real. If the list is empty, the result is 0. If the list has only Undefined values, the result is `UNDEFINED`. In other cases, the result is ERROR.

Number min([List l])
The elements of l are evaluated, producing a list l of values. Undefined values are removed. If the resulting l is composed only of numbers, the result is the minimum of the values, as a Real if any value is Real, and as an Integer otherwise. If the list is empty, the result is UNDEFINED. In other cases, the result is ERROR.

Number max([List l])
The elements of l are evaluated, producing a list l of values. Undefined values are removed. If the resulting l is composed only of numbers, the result is the maximum of the values, as a Real if any value is Real, and as an Integer otherwise. If the list is empty, the result is UNDEFINED. In other cases, the result is ERROR.

String strcat(AnyType Expr1 [, AnyType Expr2 ...])
Returns the string which is the concatenation of all arguments, where all arguments are converted to type `String` by function `string(Expr)`. Returns `ERROR` if any argument evaluates to `UNDEFINED` or `ERROR`.

String join(String sep, AnyType Expr1 [, AnyType Expr2 ...]) OR String join(String sep, List list OR String join(List list

Returns the string which is the concatenation of all arguments after the first one. The first argument is the separator, and it is inserted between each of the other arguments during concatenation. All arguments which are not undefined are converted to type `String` by function `string(Expr)` before concatenation. Undefined arguments are skipped. When there are exactly two arguments, If the second argument is a List, all members of the list are converted to strings and then joined using the separator. When there is only one argument, and the argument is a List, all members of the list are converted to strings and then concatenated.

Returns `ERROR` if any argument evaluates to `UNDEFINED` or `ERROR`.

For example:

```
"a, b, c" = join(", ", "a", "b", "c")
"abc"   = join(split("a b c"))
"a;b;c" = join(";", split("a b c"))
```

String substr(String s, Integer offset [, Integer length])

Returns the substring of `s`, from the position indicated by `offset`, with (optional) `length` characters. The first character within `s` is at offset 0. If the optional `length` argument is not present, the substring extends to the end of the string. If `offset` is negative, the value `(length - offset)` is used for the offset. If `length` is negative, an initial substring is computed, from the offset to the end of the string. Then, the absolute value of `length` characters are deleted from the right end of the initial substring. Further, where characters of this resulting substring lie outside the original string, the part that lies within the original string is returned. If the substring lies completely outside of the original string, the null string is returned.

This function returns `ERROR` if greater than 3 or less than 2 arguments are given.

Integer strcmp(AnyType Expr1, AnyType Expr2)

Both arguments are converted to type `String` by function `string(Expr)`. The return value is an integer that will be

- less than 0, if `Expr1` is lexicographically less than `Expr2`
- equal to 0, if `Expr1` is lexicographically equal to `Expr2`
- greater than 0, if `Expr1` is lexicographically greater than `Expr2`

Case is significant in the comparison. Where either argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than 2 arguments are given.

Integer stricmp(AnyType Expr1, AnyType Expr2)

This function is the same as `strcmp`, except that letter case is not significant.

Integer versioncmp(String left, String right)

This function version-compares two strings. It returns an integer

- less than zero if `left` is an earlier version than `right`
- zero if the strings are identical
- more than zero if `left` is a later version than `right`.

A version comparison is a lexicographic comparison unless the first difference between the two strings occurs in a string of digits, in which case, sort by the value of that number (assuming that more leading zeroes mean smaller numbers). Thus `7.x` is earlier than `7.y`, `7.9` is earlier than `7.10`, and the following sequence is in order: `000, 00, 01, 010, 09, 0, 1, 9, 10`.

`Boolean versionGT(String left, String right)` `Boolean versionLT(String left, String right)`
`Boolean versionGE(String left, String right)` `Boolean versionLE(String left, String right)`
`Boolean versionEQ(String left, String right)`

As `versioncmp()` (above), but for a specific comparison and returning a boolean. The two letter codes stand for "Greater Than", "Less Than", "Greater than or Equal", "Less than or Equal", and "EQual", respectively.

`Boolean version_in_range(String version, String min, String max)`

Equivalent to `versionLE(min, version) && versionLE(version, max)`.

String toUpper(AnyType Expr)
The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all lower case letters converted to upper case. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

String toLower(AnyType Expr)
The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all upper case letters converted to lower case. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Integer size(AnyType Expr)
If Expr evaluates to a string, return the number of characters in the string. If Expr evaluate to a list, return the number of elements in the list. If Expr evaluate to a classad, return the number of entries in the ad. Otherwise, `ERROR` is returned.

List split(String s [, String tokens])
Returns a list of the substrings of `s` that have been split up by using any of the characters within string `tokens`. If `tokens` is not specified, then all white space characters are used to delimit the string.

List splitUserName(String Name)
Returns a list of two strings. Where `Name` includes an `@` character, the first string in the list will be the substring that comes before the `@` character, and the second string in the list will be the substring that comes after. Thus, if `Name` is `"user@domain"`, then the returned list will be {"user", "domain"}. If there is no `@` character in `Name`, then the first string in the list will be `Name`, and the second string in the list will be the empty string. Thus, if `Name` is `"username"`, then the returned list will be {"username", ""}.

List splitSlotName(String Name)
Returns a list of two strings. Where `Name` includes an `@` character, the first string in the list will be the substring that comes before the `@` character, and the second string in the list will be the substring that comes after. Thus, if `Name` is `"slot1@machine"`, then the returned list will be {"slot1", "machine"}. If there is no `@` character in

Name, then the first string in the list will be the empty string, and the second string in the list will be Name, Thus, if Name is "machinename", then the returned list will be {"", "machinename"}.

Integer time()
Returns the current coordinated universal time. This is the time, in seconds, since midnight of January 1, 1970.

String formatTime([Integer time] [, String format])
Returns a formatted string that is a representation of time. The argument time is interpreted as coordinated universal time in seconds, since midnight of January 1, 1970. If not specified, time will default to the current time.

The argument format is interpreted similarly to the format argument of the ANSI C strftime function. It consists of arbitrary text plus placeholders for elements of the time. These placeholders are percent signs (%) followed by a single letter. To have a percent sign in the output, use a double percent sign (%%). If format is not specified, it defaults to %c.

Because the implementation uses strftime() to implement this, and some versions implement extra, non-ANSI C options, the exact options available to an implementation may vary. An implementation is only required to implement the ANSI C options, which are:

%a
abbreviated weekday name

%A
full weekday name

%b
abbreviated month name

%B
full month name

%c
local date and time representation

%d
day of the month (01-31)

%H
hour in the 24-hour clock (0-23)

%I
hour in the 12-hour clock (01-12)

%j
day of the year (001-366)

%m
month (01-12)

%M
minute (00-59)

%p
local equivalent of AM or PM

%S
second (00-59)

%U
week number of the year (Sunday as first day of week) (00-53)

%w
weekday (0-6, Sunday is 0)

%W
week number of the year (Monday as first day of week) (00-53)

%x
local date representation

%X
local time representation

%y
year without century (00-99)

%Y
year with century

%Z
time zone name, if any

String interval(Integer seconds)
Uses `seconds` to return a string of the form `days+hh:mm:ss`. This represents an interval of time. Leading values that are zero are omitted from the string. For example, `seconds` of 67 becomes "1:07". A second example, `seconds` of 1472523 = 17*24*60*60 + 1*60*60 + 2*60 + 3, results in the string "17+1:02:03".

String evalInEachContext(Expression expr, List contexts)
This function evaluates its first argument as an expression in the context of each Classad in the second argument and returns a list that is the result of each evaluation. The first argument should be an expression. If the second argument does not evaluate to a list of ClassAds, `ERROR` is returned.

For example:

```
{true, false} = evalInEachContext(Prio > 2, { [Prio=3;], [Prio=1;] })
{3, 1} = evalInEachContext(Prio, { [Prio=3;], [Prio=1;] })
ERROR = evalInEachContext(Prio > 2, { [Prio=3;], UNDEFINED })
ERROR = evalInEachContext(Prio > 2, UNDEFINED)
```

String countMatches(Expression expr, List contexts)
This function evaluates its first argument as an expression in the context of each Classad in the second argument and returns a count of the results that evaluated to `True`. The first argument should be an expression. The second argument should be a list of ClassAds or a list of attribute references to classAds, or should evaluate to a list of ClassAds. This function will always return a integer value when the first argument is defined and the second argument is not `ERROR`.

For example:

```
1 = countMatches(Prio > 2, { [Prio=3;], [Prio=1;] })
1 = countMatches(Prio > 2, { [Prio=3;], UNDEFINED })
0 = countMatches(Prio > 2, UNDEFINED)
```

AnyType debug(AnyType expression)
This function evaluates its argument, and it returns the result. Thus, it is a no-operation. However, a side-effect

of the function is that information about the evaluation is logged to the evaluating program's log file, at the D_FULLDEBUG debug level. This is useful for determining why a given ClassAd expression is evaluating the way it does. For example, if a *condor_startd* START expression is unexpectedly evaluating to UNDEFINED, then wrapping the expression in this debug() function will log information about each component of the expression to the log file, making it easier to understand the expression.

String envV1ToV2(String old_env)
This function converts a set of environment variables from the old HTCondor syntax to the new syntax. The single argument should evaluate to a string that represents a set of environment variables using the old HTCondor syntax (usually stored in the job ClassAd attribute Env). The result is the same set of environment variables using the new HTCondor syntax (usually stored in the job ClassAd attribute Environment). If the argument evaluates to UNDEFINED, then the result is also UNDEFINED.

String mergeEnvironment(String env1 [, String env2, ...])
This function merges multiple sets of environment variables into a single set. If multiple arguments include the same variable, the one that appears last in the argument list is used. Each argument should evaluate to a string which represents a set of environment variables using the new HTCondor syntax or UNDEFINED, which is treated like an empty string. The result is a string that represents the merged set of environment variables using the new HTCondor syntax (suitable for use as the value of the job ClassAd attribute Environment).

For the following functions, a delimiter is represented by a string. Each character within the delimiter string delimits individual strings within a list of strings that is given by a single string. The default delimiter contains the comma and space characters. A string within the list is ended (delimited) by one or more characters within the delimiter string.

Integer stringListSize(String list [, String delimiter])
Returns the number of elements in the string list, as delimited by the optional delimiter string. Returns ERROR if either argument is not a string.

This function returns ERROR if other than 1 or 2 arguments are given.

Integer stringListSum(String list [, String delimiter]) OR Real stringListSum(String list [, String delimiter])
Sums and returns the sum of all items in the string list, as delimited by the optional delimiter string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR.

Real stringListAvg(String list [, String delimiter])
Sums and returns the real-valued average of all items in the string list, as delimited by the optional delimiter string. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value 0.0.

Integer stringListMin(String list [, String delimiter]) OR Real stringListMin(String list [, String delimiter])
Finds and returns the minimum value from all items in the string list, as delimited by the optional delimiter string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value UNDEFINED.

Integer stringListMax(String list [, String delimiter]) OR Real stringListMax(String list [, String delimiter])
Finds and returns the maximum value from all items in the string list, as delimited by the optional delimiter

string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value UNDEFINED.

Boolean stringListMember(String x, String list [, String delimiter])
Returns TRUE if item x is in the string list, as delimited by the optional delimiter string. Returns FALSE if item x is not in the string list. Comparison is done with strcmp(). The return value is ERROR, if any of the arguments are not strings.

Boolean stringListIMember(String x, String list [, String delimiter])
Same as stringListMember(), but comparison is done with stricmp(), so letter case is not relevant.

Integer stringListsIntersect(String list1, String list2 [, String delimiter])
Returns TRUE if the lists contain any matching elements, and returns FALSE if the lists do not contain any matching elements. Returns ERROR if either argument is not a string or if an incorrect number of arguments are given.

Boolean stringListSubsetMatch(String list1, String list2 [, String delimiter])
Returns TRUE if all item in the string list1 are also in the string list2, as delimited by the optional delimiter string. Returns FALSE if list1 has any items that are not in list2. Both lists are treated as sets. Empty items and duplicate items are ignored. The return value is TRUE if list1 is UNDEFINED or empty and list2 is any string value. The return value is FALSE if list1 is any string vlaue and list2 is UNDEFINED. The return value is UNDEFINED if both list1 and list2 are UNDEFINED. The return value is ERROR, if any of the arguments are not either strings or UNDEFINED

Boolean stringListISubsetMatch(String list1, String list2 [, String delimiter])
Same as stringListSubsetMatch(), but the sets are case-insensitive.

The following three functions utilize regular expressions as defined and supported by the PCRE library. See http://www.pcre.org for complete documentation of regular expressions.

The options argument to these functions is a string of special characters that modify the use of the regular expressions. Inclusion of characters other than these as options are ignored.

I or i
Ignore letter case.

M or m
Modifies the interpretation of the caret (^) and dollar sign ($) characters. The caret character matches the start of a string, as well as after each newline character. The dollar sign character matches before a newline character.

S or s
The period matches any character, including the newline character.

F or f
When doing substitution, return the full target string with substitutions applied. Normally, only the substitute text is returned.

G or g
When doing substitution, apply the substitution for every matching portion of the target string (that doesn't overlap a previous match).

Boolean regexp(String pattern, String target [, String options])
Uses the regular expression given by string `pattern` to scan through the string `target`. Returns `TRUE` when `target` matches the regular expression given by `pattern`. Returns `FALSE` otherwise. If any argument is not a string, or if `pattern` does not describe a valid regular expression, returns `ERROR`.

Boolean regexpMember(String pattern, List targetStrings [, String options])
Uses the description of a regular expression given by string `pattern` to scan through a List of string n `targetStrings`. Returns `TRUE` when `target` matches a regular expression given by `pattern`. If no strings match, and at least one item in targetString evaluated to undefined, returns undefined. If any item in targetString before a match evaluated to neither a string nor undefined, returns `ERROR`.

String regexps
`(String pattern, String target, String substitute [ , String options ])` Uses the regular expression given by string `pattern` to scan through the string `target`. When `target` matches the regular expression given by `pattern`, the string `substitute` is returned, with backslash expansion performed. If any argument is not a string, returns `ERROR`.

String replace
`(String pattern, String target, String substitute [ , String options ])` Uses the regular expression given by string `pattern` to scan through the string `target`. Returns a modified version of `target`, where the first substring that matches `pattern` is replaced by the string `substitute`, with backslash expansion performed. Equivalent to `regexps()` with the `f` option. If any argument is not a string, returns `ERROR`.

String replaceall
`(String pattern, String target, String substitute [ , String options ])` Uses the regular expression given by string `pattern` to scan through the string `target`. Returns a modified version of `target`, where every substring that matches `pattern` is replaced by the string `substitute`, with backslash expansion performed. Equivalent to `regexps()` with the `fg` options. If any argument is not a string, returns `ERROR`.

Boolean stringList_regexpMember
`(String pattern, String list [ , String delimiter ] [ , String options ])` Uses the description of a regular expression given by string `pattern` to scan through the list of strings in `list`. Returns `TRUE` when one of the strings in `list` is a regular expression as described by `pattern`. The optional `delimiter` describes how the list is delimited, and string `options` modifies how the match is performed. Returns `FALSE` if `pattern` does not match any entries in `list`. The return value is `ERROR`, if any of the arguments are not strings, or if `pattern` is not a valid regular expression.

String userHome(String userName [, String default])
Returns the home directory of the given user as configured on the current system (determined using the getpwdnam() call). (Returns `default` if the `default` argument is passed and the home directory of the user is not defined.)

List userMap(String mapSetName, String userName)
Map an input string using the given mapping set. Returns a string containing the list of groups to which the user belongs separated by commas or undefined if the user was not found in the map file.

String userMap(String mapSetName, String userName, String preferredGroup)
Map an input string using the given mapping set. Returns a string, which is the preferred group if the user is in that group; otherwise it is the first group to which the user belongs, or undefined if the user belongs to no groups.

String userMap(String mapSetName, String userName, String preferredGroup, String defaultGroup)
Map an input string using the given mapping set. Returns a string, which is the preferred group if the user is

in that group; the first group to which the user belongs, if any; and the default group if the user belongs to no groups.

The maps for the `userMap()` function are defined by the following configuration macros: `<SUBSYS>_CLASSAD_USER_MAP_NAMES`, `CLASSAD_USER_MAPFILE_<name>` and `CLASSAD_USER_MAPDATA_<name>` (see the *HTCondor-wide Configuration File Entries* section).

6.1.3 ClassAd Evaluation Semantics

The ClassAd mechanism's primary purpose is for matching entities that supply constraints on candidate matches. The mechanism is therefore defined to carry out expression evaluations in the context of two ClassAds that are testing each other for a potential match. For example, the *condor_negotiator* evaluates the `Requirements` expressions of machine and job ClassAds to test if they can be matched. The semantics of evaluating such constraints is defined below.

Evaluating Literals

Literals are self-evaluating, Thus, integer, string, real, undefined and error values evaluate to themselves.

Attribute References

Since the expression evaluation is being carried out in the context of two ClassAds, there is a potential for name space ambiguities. The following rules define the semantics of attribute references made by ClassAd A that is being evaluated in a context with another ClassAd B:

1. If the reference is prefixed by a scope resolution prefix,
    - If the prefix is `MY.`, the attribute is looked up in ClassAd A. If the named attribute does not exist in A, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
    - Similarly, if the prefix is `TARGET.`, the attribute is looked up in ClassAd B. If the named attribute does not exist in B, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
2. If the reference is not prefixed by a scope resolution prefix,
    - If the attribute is defined in A, the value of the reference is the value of the expression bound to the attribute name in A.
    - Otherwise, if the attribute is defined in B, the value of the reference is the value of the expression bound to the attribute name in B.
    - Otherwise, if the attribute is defined in the ClassAd environment, the value from the environment is returned. This is a special environment, to be distinguished from the Unix environment. Currently, the only attribute of the environment is `CurrentTime`, which evaluates to the integer value returned by the system call `time(2)`.
    - Otherwise, the value of the reference is `UNDEFINED`.
3. Finally, if the reference refers to an expression that is itself in the process of being evaluated, there is a circular dependency in the evaluation. The value of the reference is `ERROR`.

ClassAd Operators

All operators in the ClassAd language are total, and thus have well defined behavior regardless of the supplied operands. Furthermore, most operators are strict with respect to ERROR and UNDEFINED, and thus evaluate to ERROR or UNDEFINED if either of their operands have these exceptional values.

- **Arithmetic operators:**
  1. The operators *, /, + and - operate arithmetically only on integers and reals.
  2. Arithmetic is carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is an integer and the other real.
  3. The operators are strict with respect to both UNDEFINED and ERROR.
  4. If either operand is not a numerical type, the value of the operation is ERROR.
- **Comparison operators:**
  1. The comparison operators ==, !=, <=, <, >= and > operate on integers, reals and strings.
  2. String comparisons are case insensitive for most operators. The only exceptions are the operators =?= and =!=, which do case sensitive comparisons assuming both sides are strings.
  3. Comparisons are carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is a real, and the other an integer. Strings may not be converted to any other type, so comparing a string and an integer or a string and a real results in ERROR.
  4. The operators ==, !=, <=, <, >=, and > are strict with respect to both UNDEFINED and ERROR.
  5. In addition, the operators =?=, is, =!=, and isnt behave similar to == and !=, but are not strict. Semantically, the =?= and is test if their operands are "identical," i.e., have the same type and the same value. For example, 10 == UNDEFINED and UNDEFINED == UNDEFINED both evaluate to UNDEFINED, but 10 =?= UNDEFINED and UNDEFINED is UNDEFINED evaluate to FALSE and TRUE respectively. The =!= and isnt operators test for the "is not identical to" condition.

     =?= and is have the same behavior as each other. And isnt and =!= behave the same as each other. The ClassAd unparser will always use =?= in preference to is and =!= in preference to isnt when printing out ClassAds.
- **Logical operators:**
  1. The logical operators && and || operate on integers and reals. The zero value of these types are considered FALSE and non-zero values TRUE.
  2. The operators are not strict, and exploit the "don't care" properties of the operators to squash UNDEFINED and ERROR values when possible. For example, UNDEFINED && FALSE evaluates to FALSE, but UNDEFINED || FALSE evaluates to UNDEFINED.
  3. Any string operand is equivalent to an ERROR operand for a logical operator. In other words, TRUE && "foobar" evaluates to ERROR.
- **The Ternary operator:**
  1. The Ternary operator (expr1 ? expr2 : expr3) operate with expressions. If all three expressions are given, the operation is strict.
  2. However, if the middle expression is missing, eg. expr1 ?: expr3, then, when expr1 is defined, that defined value is returned. Otherwise, when expr1 evaluated to UNDEFINED, the value of expr3 is evaluated and returned. This can be a convenient shortcut for writing what would otherwise be a much longer classad expression.

Expression Examples

The =?= operator is similar to the == operator. It checks if the left hand side operand is identical in both type and value to the the right hand side operand, returning TRUE when they are identical.

Caution: For strings, the comparison is case-insensitive with the == operator and case-sensitive with the =?= operator. A key point in understanding is that the =?= operator only produces evaluation results of TRUE and FALSE, where the == operator may produce evaluation results TRUE, FALSE, UNDEFINED, or ERROR.

Table 4.1 presents examples that define the outcome of the == operator. Table 4.2 presents examples that define the outcome of the =?= operator.

expression	evaluated result
(10 == 10)	TRUE
(10 == 5)	FALSE
(10 == "ABC")	ERROR
"ABC" == "abc"	TRUE
(10 == UNDEFINED)	UNDEFINED
(UNDEFINED == UNDEFINED)	UNDEFINED

Table 4.1: Evaluation examples for the == operator

expression	evaluated result
(10 =?= 10)	TRUE
(10 =?= 5)	FALSE
(10 =?= "ABC")	FALSE
"ABC" =?= "abc"	FALSE
(10 =?= UNDEFINED)	FALSE
(UNDEFINED =?= UNDEFINED)	TRUE

Table 4.2: Evaluation examples for the =?= operator

The =!= operator is similar to the != operator. It checks if the left hand side operand is not identical in both type and value to the the right hand side operand, returning FALSE when they are identical.

Caution: For strings, the comparison is case-insensitive with the != operator and case-sensitive with the =!= operator. A key point in understanding is that the =!= operator only produces evaluation results of TRUE and FALSE, where the != operator may produce evaluation results TRUE, FALSE, UNDEFINED, or ERROR.

Table 4.3 presents examples that define the outcome of the != operator. Table 4.4 presents examples that define the outcome of the =!= operator.

expression	evaluated result
(10 != 10)	FALSE
(10 != 5)	TRUE
(10 != "ABC")	ERROR
"ABC" != "abc"	FALSE
(10 != UNDEFINED)	UNDEFINED
(UNDEFINED != UNDEFINED)	UNDEFINED

Table 4.3: Evaluation examples for the != operator

expression	evaluated result
(10 =!= 10)	FALSE
(10 =!= 5)	TRUE
(10 =!= "ABC")	TRUE
"ABC" =!= "abc"	TRUE
(10 =!= UNDEFINED)	TRUE
(UNDEFINED =!= UNDEFINED)	FALSE

Table 4.4: Evaluation examples for the =!= operator

6.1.4 Old ClassAds in the HTCondor System

The simplicity and flexibility of ClassAds is heavily exploited in the HTCondor system. ClassAds are not only used to represent machines and jobs in the HTCondor pool, but also other entities that exist in the pool such as submitters of jobs and master daemons. Since arbitrary expressions may be supplied and evaluated over these ClassAds, users have a uniform and powerful mechanism to specify constraints over these ClassAds. These constraints can take the form of `Requirements` expressions in resource and job ClassAds, or queries over other ClassAds.

Constraints and Preferences

The `requirements` and `rank` expressions within the submit description file are the mechanism by which users specify the constraints and preferences of jobs. For machines, the configuration determines both constraints and preferences of the machines.

For both machine and job, the `rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, a job ClassAd may contain the following expressions:

```
Requirements = (Arch == "INTEL") && (OpSys == "LINUX")
Rank         = TARGET.Memory + TARGET.Mips
```

In this case, the job requires a 32-bit Intel processor running a Linux operating system. Among all such computers, the customer prefers those with large physical memories and high MIPS ratings. Since the `Rank` is a user-specified metric, any expression may be used to specify the perceived desirability of the match. The *condor_negotiator* daemon runs algorithms to deliver the best resource (as defined by the `rank` expression), while satisfying other required criteria.

Similarly, the machine may place constraints and preferences on the jobs that it will run by setting the machine's configuration. For example,

```
Friend        = Owner == "tannenba" || Owner == "wright"
ResearchGroup = Owner == "jbasney" || Owner == "raman"
Trusted       = Owner != "rival" && Owner != "riffraff"
START         = Trusted && ( ResearchGroup || LoadAvg < 0.3 && KeyboardIdle > 15*60 )
RANK          = Friend + ResearchGroup*10
```

The above policy states that the computer will never run jobs owned by users rival and riffraff, while the computer will always run a job submitted by members of the research group. Furthermore, jobs submitted by friends are preferred to other foreign jobs, and jobs submitted by the research group are preferred to jobs submitted by friends.

Note: Because of the dynamic nature of ClassAd expressions, there is no a priori notion of an integer-valued expression, a real-valued expression, etc. However, it is intuitive to think of the `Requirements` and `Rank` expressions as integer-valued and real-valued expressions, respectively. If the actual type of the expression is not of the expected type, the value is assumed to be zero.

Querying with ClassAd Expressions

The flexibility of this system may also be used when querying ClassAds through the *condor_status* and *condor_q* tools which allow users to supply ClassAd constraint expressions from the command line.

Needed syntax is different on Unix and Windows platforms, due to the interpretation of characters in forming command-line arguments. The expression must be a single command-line argument, and the resulting examples differ for the platforms. For Unix shells, single quote marks are used to delimit a single argument. For a Windows command window, double quote marks are used to delimit a single argument. Within the argument, Unix escapes the double quote mark by prepending a backslash to the double quote mark. Windows escapes the double quote mark by prepending another double quote mark. There may not be spaces in between.

Here are several examples. To find all computers which have had their keyboards idle for more than 60 minutes and have more than 4000 MB of memory, the desired ClassAd expression is

```
KeyboardIdle > 60*60 && Memory > 4000
```

On a Unix platform, the command appears as

```
$ condor_status -const 'KeyboardIdle > 60*60 && Memory > 4000'

Name               OpSys   Arch   State     Activity LoadAv Mem  ActvtyTime
100
slot1@altair.cs.wi LINUX   X86_64 Owner     Idle     0.000 8018 13+00:31:46
slot2@altair.cs.wi LINUX   X86_64 Owner     Idle     0.000 8018 13+00:31:47
...
...
slot1@athena.stat. LINUX   X86_64 Unclaimed Idle     0.000 7946  0+00:25:04
slot2@athena.stat. LINUX   X86_64 Unclaimed Idle     0.000 7946  0+00:25:05
...
...
```

The Windows equivalent command is

```
> condor_status -const "KeyboardIdle > 60*60 && Memory > 4000"
```

Here is an example for a Unix platform that utilizes a regular expression ClassAd function to list specific information. A file contains ClassAd information. *condor_advertise* is used to inject this information, and *condor_status* constrains the search with an expression that contains a ClassAd function.

```
$ cat ad
MyType = "Generic"
FauxType = "DBMS"
Name = "random-test"
Machine = "f05.cs.wisc.edu"
MyAddress = "<128.105.149.105:34000>"
DaemonStartTime = 1153192799
UpdateSequenceNumber = 1
```

(continues on next page)

(continued from previous page)

```
$ condor_advertise UPDATE_AD_GENERIC ad

$ condor_status -any -constraint 'FauxType=="DBMS" && regexp("random.*", Name, "i")'

MyType              TargetType          Name

Generic             None                random-test
```

The ClassAd expression describing a machine that advertises a Windows operating system:

```
OpSys == "WINDOWS"
```

Here are three equivalent ways on a Unix platform to list all machines advertising a Windows operating system. Spaces appear in these examples to show where they are permitted.

```
$ condor_status -constraint ' OpSys == "WINDOWS"  '
```

```
$ condor_status -constraint OpSys==\"WINDOWS\"
```

```
$ condor_status -constraint "OpSys==\"WINDOWS\""
```

The equivalent command on a Windows platform to list all machines advertising a Windows operating system must delimit the single argument with double quote marks, and then escape the needed double quote marks that identify the string within the expression. Spaces appear in this example where they are permitted.

```
> condor_status -constraint " OpSys == ""WINDOWS"" "
```

6.1.5 Extending ClassAds with User-written Functions

The ClassAd language provides a rich set of functions. It is possible to add new functions to the ClassAd language without recompiling the HTCondor system or the ClassAd library. This requires implementing the new function in the C++ programming language, compiling the code into a shared library, and telling HTCondor where in the file system the shared library lives.

While the details of the ClassAd implementation are beyond the scope of this document, the ClassAd source distribution ships with an example source file that extends ClassAds by adding two new functions, named todays_date() and double(). This can be used as a model for users to implement their own functions. To deploy this example extension, follow the following steps on Linux:

- Download the ClassAd source distribution from http://www.cs.wisc.edu/condor/classad.
- Unpack the tarball.
- Inspect the source file `shared.cpp`. This one file contains the whole extension.
- Build `shared.cpp` into a shared library. On Linux, the command line to do so is

```
$ g++ -DWANT_CLASSAD_NAMESPACE -I. -shared -o shared.so \
  -Wl,-soname,shared.so -o shared.so -fPIC shared.cpp
```

- Copy the file `shared.so` to a location that all of the HTCondor tools and daemons can read.

```
$ cp shared.so `condor_config_val LIBEXEC`
```

- Tell HTCondor to load the shared library into all tools and daemons, by setting the CLASSAD_USER_LIBS configuration variable to the full name of the shared library. In this case,

```
CLASSAD_USER_LIBS = $(LIBEXEC)/shared.so
```

- Restart HTCondor.
- Test the new functions by running

```
$ condor_status -format "%s\n" todays_date()
```

6.2 ClassAd Transforms

HTCondor has a general purpose language for transforming ClassAds, this language is used by the *condor_schedd* for submit transforms, and as of version 8.9.7 by the job router for routes and pre and post route transforms.

There is also a stand alone tool `condor_transform_ads` than can read ClassAds from a file or pipe, transform them, and write the resulting ClassAds to a file or pipe.

The transform language is build on the same basic macro expansion engine use by HTCondor configuration and by *condor_submit* and shares many of the same features such as `$()` macro expansion and if statements.

This transform language is a superset of an earlier transform language based on New ClassAds. The *condor_schedd* and *condor_job_router* will still allow the earlier transform language, and they will automatically convert configuration from earlier New ClassAds style transforms to the to the native transform language when they read the configuration.

6.2.1 General Concepts

Transforms consists of a sequence of lines containing `key=value` pairs or transform commands such as `SET`. Transform commands execute in order from top to bottom and may make use of macro values set by earlier statements using `$(var)` macro substitution. Unlike configuration files, Transform commands will use the value of `$(var)` defined at the time, rather than the last value defined in the configuration file.

If/else statements and macro functions such as `$INT(var)` can be used in transforms, but *include* may not be used.

A macro expansion of the form `$(MY.<attr>)` will expand as the value of the attribute `<attr>` of the ClassAd that is being transformed. Expansion will expand simple string values without quotes but will not evaluate expressions. Use `$STRING(MY.<attr>)` or `$INT(MY.<attr>)` if you need to evaluate the ClassAd attribute before expanding it.

The existence of an attribute in the ClassAd being transformed can be tested by using `if defined MY.<attr>`

In the definitions below.

`<attr>` must be a valid ClassAd attribute name

`<newattr>` must be a valid ClassAd attribute name

`<expr>` must be a valid ClassAd expression after `$()` macro expansion. Don't forget to quote string values!

`<var>` must be a valid macro name

`<regex>` is a regular expression

`<attrpat>` is a regular expression substitution pattern, which may include capture groups `\0`, `\1`, etc.

6.2.2 Transform Commands

<var> = <value>
Sets the temporary macro variable <var> to <value>. This is the same sort of macro assignment used in configuration and submit files, the value is everything after the = until then end of the line with leading and trailing whitespace removed. Variables set in this way do not directly affect the resulting transformed ClassAd, but they can be used later in the transform by $(var) macro expansion. In the *condor_job_router* some macro variable names will affect the way the router behaves. For a list of macro variable names have have special meaning to the *condor_job_router* see the *Routing Table Entry Commands and Macro values* section.

REQUIREMENTS <expr>
Apply the transform only if the expression given by <expr> evaluates to true when evaluated against the untransformed ClassAd.

SET <attr> <expr>
Sets the ClassAd value of <attr> to <expr> in the ClassAd being transformed.

DEFAULT <attr> <expr>
Sets the ClassAd value of <attr> to <expr> in the ClassAd being transformed if that ClassAd does not currently have <attr> or if it is currently set to undefined. This is equivalent to

```
if ! defined MY.<Attr>
  SET <Attr> <value>
endif
```

EVALSET <attr> <expr>
Evaluate <expr> and set the ClassAd value of <attr> to the result of the evaluation. Use this when the ClassAd value of <attr> must be a simple value rather than expression, or when you need to capture the result of evaluating at transform time. Note that it is usually better to use SET with macro expansions when you want to modify a ClassAd attribute as part of a transform.

EVALMACRO <var> <expr>
Evaluate <expr> and set the temporary macro variable <var> to the result of evaluation. $(var) can the be used in later transform statements such as SET or if.

COPY <attr> <newattr>
Copies the ClassAd value of <attr> to a new ClassAd attribute <newattr>. This will result in two attributes that have the same value at this step of the transform.

COPY /<regex>/ <attrpat>
Copies all ClassAd attributes that have names matching the regular expression <regex> to new attribute names. The new attribute names are defined by <attrpat> which may have regular expression capture groups to substitute portions of the original attribute name. \0 Is the entire attribute name, and \1 is the first capture, etc. For example

```
# copy all attributes whose names begin with Resource to new attribute with names␣
↪that begin with OriginalResource
COPY /Resource(.+)/ OriginalResource\1
```

RENAME <attr> <newattr>
Renames the attribute <attr> to a new attribute name <newattr>. This is the equivalent of a COPY statement followed by a DELETE statement.

RENAME /<regex>/ <attrpat>
Renames all ClassAd attributes that match the regular expression <regex> to new attribute names given by the substitution pattern <attrpat>.

DELETE <attr>
Deletes the ClassAd attribute <attr> from the transformed ClassAd.

DELETE /<regex>/
Deletes all ClassAd attributes whose names match the regular expression <regex> from the transformed ClassAd.

6.3 Print Formats

Many HTCondor tools that work with ClassAds use a formatting engine called the ClassAd pretty printer. Tools that have a **-format** or **-autoformat** argument use those arguments to configure the ClassAd pretty printer, and then use the pretty printer to produce output from ClassAds.

The *condor_q*, *condor_history* and *condor_status* tools, as well as others that have a **-print-format** or **-pr** argument can configure the ClassAd pretty using a file. The syntax of this file is described below.

Not all tools support all of the print format options.

6.3.1 Syntax

A print format file consists of a heading line and zero or more formatting lines followed by optional constraint, sort and summary lines. These sections of the format file begin with the keywords SELECT, WHERE, GROUP, or SUMMARY which must be in that order if they appear. These keywords must be all uppercase and must be the first word on the line.

A line beginning with # is treated as a comment

A custom print format file must begin with the SELECT keyword. The SELECT keyword can be followed by options to qualify the type of query, the global formatting options and whether or not there will be column headings. The prototype for the SELECT line is:

SELECT [FROM AUTOCLUSTER | UNIQUE] [BARE | NOTITLE | NOHEADER | NOSUMMARY] [LABEL [SEPARATOR <string>]] [<separators>]

The first two optional keywords indicate the query type. These options work only in *condor_q*.

FROM AUTOCLUSTER
Used with condor_q to query the schedd's default autocluster set.

UNIQUE
Used with condor_q to ask the *condor_schedd* to count unique values. This option tells the schedd to building a new FROM AUTOCLUSTER set using the given attributes

The next set of optional keywords enable or disable various things that are normally printed before or after the classad output.

NOTITLE
Disables the title on tools that have a title, like the Schedd name from *condor_q*.

NOHEADER
Disables column headers.

NOSUMMARY
Disables the summary output such as the totals by job stats at the bottom of normal *condor_q* output.

BARE
Shorthand for NOTITLE NOHEADER NOSUMMARY

In the descriptions below <string> is text. If the text starts with a single quote, then it continues to the next single quote. If it starts with a doublequote, it continues to the next doublequote. If it starts with neither, then it continues until the next space or tab. a n, r or t inside the string will be converted into a newline, carriage return or tab character respectively.

LABEL [SEPARATOR <string>]
Use item labels rather than column headers. The separator between the label and the value will be = unless the SEPARATOR is used to define a different one.

RECORDPREFIX <string>
The value of <string> will be printed before each ClassAd. The default is to print nothing.

RECORDSUFFIX <string>
The value of <string> will be printed after each ClassAd. The default is to print the newline character.

FIELDPREFIX <string>
The value of <string> will be printed before each attribute or expression. The default is to print nothing.

FIELDSUFFIX <string>
The value of <string> will be printed after each attribute or expression. The default is to print a single space.

After the SELECT line, there should be zero or more formatting lines one line for each field in the output. Each formatting line is a ClassAd attribute or expression followed by zero or more keywords that control formatting, the first valid keyword ends the expression. Keywords are all uppercase and space delimited. The prototype for each formatting line is:

<expr> [AS <label>] [PRINTF <format-string> | PRINTAS <function-name> [ALWAYS] | WIDTH [AUTO | [-]<INT>]] [FIT | TRUNCATE] [LEFT | RIGHT] [NOPREFIX] [NOSUFFIX]

AS <string>
defines the label or column heading. if the formatting line has no AS keyword, then <expr> will be used as the label or column heading

PRINTF <string>
<string> should be a c++ printf format string, the same as used by the **-format** command line arguments for tools

PRINTAS <function>
Format using the built-in function. The Valid function names for PRINTAS are defined by the code and differ between the various tools, refer to the table at the end of this page.

WIDTH [-]<int>
Align the data to the given width, negative values left align.

WIDTH AUTO
Use a width sized to fit the largest item.

FIT
Adjust column width to fit the data, normally used with WIDTH AUTO

TRUNCATE
If the data is larger than the given width, truncate it

LEFT
Left align the data to the given width

RIGHT
Rigth align the data to the given width

NOPREFIX
Do not include the FIELDPREFIX string for this field

NOSUFFIX
Do not include the FIELDSUFFIX string for this field

OR <char>[<char>]
if the field data is undefined, print <char>, if <char> is doubled, fill the column with <char>. Allowed values for <char> are space or one of the following ?*.-_#0

After the field formatting lines, there may be sections in the file that define a query constraint, sorting and grouping and the summary line. These sections can be multiple lines, but must begin with a keyword.

WHERE <constraint-expr>
Display only ClassAds where the expression <constraint-expr> evaluates to true.

GROUP BY <sort-expr> [ASCENDING | DECENDING]
Sort the ClassAds by evaluating <sort-expr>. If multiple sort keys are desired, the GROUP BY line can be followed by lines containing additional expressions, for example

```
GROUP BY
  Owner
  ClusterId  DECENDING
```

SUMMARY [STANDARD | NONE]
Enable or disable the summary totals. The summary can also be disabled using NOSUMMARY or BARE keywords on the SELECT line.

6.3.2 Examples

This print format file produces the default -nobatch output of *condor_q*

```
# queue.cpf
# produce the standard output of condor_q
SELECT
   ClusterId     AS "    ID"  NOSUFFIX WIDTH AUTO
   ProcId        AS " "       NOPREFIX       PRINTF ".%-3d"
   Owner         AS "OWNER"          WIDTH -14   PRINTAS OWNER
   QDate         AS "  SUBMITTED"    WIDTH 11    PRINTAS QDATE
   RemoteUserCpu AS "    RUN_TIME"   WIDTH 12    PRINTAS CPU_TIME
   JobStatus     AS ST                             PRINTAS JOB_STATUS
   JobPrio       AS PRI
   ImageSize     AS SIZE             WIDTH 6     PRINTAS MEMORY_USAGE
   Cmd           AS CMD                          PRINTAS JOB_DESCRIPTION
SUMMARY STANDARD
```

This print format file produces only totals

```
# q_totals.cpf
# show only totals with condor_q
SELECT NOHEADER NOTITLE
SUMMARY STANDARD
```

This print format file shows typical fields of the Schedd autoclusters.

```
# negotiator_autocluster.cpf
SELECT FROM AUTOCLUSTER
   Owner         AS OWNER         WIDTH -14   PRINTAS OWNER
```

(continues on next page)

(continued from previous page)

```
    JobCount       AS COUNT                    PRINTF %5d
    AutoClusterId  AS " ID"          WIDTH 3
    JobUniverse    AS UNI                      PRINTF %3d
    RequestMemory  AS REQ_MEMORY     WIDTH 10   PRINTAS READABLE_MB
    RequestDisk    AS REQUEST_DISK   WIDTH 12   PRINTAS READABLE_KB
    JobIDs         AS JOBIDS
GROUP BY Owner
```

This print format file shows the use of `SELECT UNIQUE`

```
# count_jobs_by_owner.cpf
# aggregate by the given attributes, return unique values plus count and jobids.
# This query builds an autocluster set in the schedd on the fly using all of the␣
↪displayed attributes
# And all of the GROUP BY attributes (except JobCount and JobIds)
SELECT UNIQUE NOSUMMARY
    Owner          AS OWNER      WIDTH -20
    JobUniverse    AS "UNIVERSE "    PRINTAS JOB_UNIVERSE
    JobStatus      AS STATUS     PRINTAS JOB_STATUS_RAW
    RequestCpus    AS CPUS
    RequestMemory  AS MEMORY
    JobCount       AS COUNT      PRINTF  %5d
    JobIDs
GROUP BY
    Owner
```

6.3.3 PRINTAS functions for *condor_q*

Some of the tools that interpret a print format file have specialized formatting functions for certain ClassAd attributes. These are selected by using the `PRINTAS` keyword followed by the function name. Available function names depend on the tool. Some functions implicitly use the value of certain attributes, often multiple attributes. The list for *condor_q* is.

BATCH_NAME
: Used for the `BATCH_NAME` column of the default output of *condor_q*. This function constructs a batch name string using value of the `JobBatchName` attribute if it exists, otherwise it constructs a batch name from `JobUniverse`, `ClusterId`, `DAGManJobId`, and `DAGNodeName`.

BUFFER_IO_MISC
: Used for the `MISC` column of the `-io` output of *condor_q*. This function constructs an IO string that varies by `JobUniverse`. The output for Standard universe jobs refers to `FileSeekCount`, `BufferSize` and `BufferBlockSize`. For all other jobs it refers to `TransferrringInput`, `TransferringOutput` and `TransferQueued`.

CPU_TIME
: Used for the `RUN_TIME` or `CPU_TIME` column of the default *condor_q* output. The result of the function depends on wether the `-currentrun` argument is used with *condor_q*. If `RemoteUserCpu` is undefined, this function returns undefined. It returns the value of `RemoteUserCpu` if it is non-zero. Otherwise it reports the amount of time that the *condor_shadow* has been alive. If the `-currentrun` argument is used with *condor_q*, this will be the shadow lifetime for the current run only. If it is not, then the result is the sum of `RemoteWallClockTime` and the current shadow lifetime. The result is then formatted using the `%T` format.

CPU_UTIL
: Used for the `CPU_UTIL` column of the default *condor_q* output. This function returns `RemoteUserCpu` divided

by CommittedTime if CommittedTime is non-zero. It returns undefined if CommittedTime is undefined, zero or negative. The result is then formatted using the %.1f format.

DAG_OWNER
Used for the OWNER column of default *condor_q* output. This function returns the value of the Owner attribute when the -dag option is not passed to *condor_q*. When the -dag option is passed, it returns the value of DAGNodeName for jobs that have a DAGManJobId defined, and Owner for all other jobs.

GRID_JOB_ID
Used for the GRID_JOB_ID column of the -grid output of *condor_q*. This function extracts and returns the job id from the GridJobId attribute.

GRID_RESOURCE
Used for the GRID->MANAGER HOST column of the -grid output of *condor_q*. This funciton extracts and returns the manager and host from the GridResource attribute. For ec2 jobs the host will be the value of EC2RemoteVirtualMachineName attribute.

GRID_STATUS
Used for the STATUS column of the -grid output of *condor_q*. This function renders the status of grid jobs from the GridJobStatus attribute. If the attribute has a string value it is reported unmodified. Otherwise, if GridJobStatus is an integer, it is presumed to be a condor job status and converted to a string.

JOB_DESCRIPTION
Used for the CMD column of the default output of *condor_q*. This function renders a job description from the MATCH_EXP_JobDescription, JobDescription or Cmd and Args or Arguments job attributes.

JOB_FACTORY_MODE
Used for the MODE column of the -factory output of *condor_q*. This function renders an integer value into a string value using the conversion for JobMaterializePaused modes.

JOB_ID
Used for the ID column of the default output of *condor_q*. This function renders a string job id from the ClusterId and ProcId attributes of the job.

JOB_STATUS
Used for the ST column of the default output of *condor_q*. This function renders a one or two character job status from the JobStatus, TransferringInput, TransferringOutput, TransferQueued and LastSuspensionTime attributes of the job.

JOB_STATUS_RAW
This function converts an integer to a string using the conversion for JobStatus values.

JOB_UNIVERSE
Used for the UNIVERSE column of the -idle and -autocluster output of *condor_q*. This funciton converts an integer to a string using the conversion for JobUniverse values. Values that are outside the range of valid universes are rendered as Unknown.

MEMORY_USAGE
Used for the SIZE column of the default output of *condor_q*. This function renders a memory usage value in megabytes the MemoryUsage or ImageSize attributes of the job.

OWNER
Used for the OWNER column of the default output of *condor_q*. This function renders an Owner string from the Owner attribute of the job. Prior to 8.9.9, this function would modify the result based on the NiceUser attribute of the job, but it no longer does so.

QDATE
Used for the SUBMITTED column of the default output of *condor_q*. This function converts a Unix timestamp to a string date and time with 2 digit month, day, hour and minute values.

READABLE_BYTES
Used for the INPUT and OUTPUT columns of the -io output of *condor_q* This function renders a numeric byte value into a string with an appropriate B, KB, MB, GB, or TB suffix.

READABLE_KB
This function renders a numeric Kibibyte value into a string with an appropriate B, KB, MB, GB, or TB suffix. Use this for Job attributes that are valued in Kb, such as DiskUsage.

READABLE_MB
This function renders a numeric Mibibyte value into a string with an appropriate B, KB, MB, GB, or TB suffix. Use this for Job attributes that are valued in Mb, such as MemoryUsage.

REMOTE_HOST
Used for the HOST(S) column of the -run output of *condor_q*. This function extracts the host name from a job attribute appropriate to the JobUniverse value of the job. For Local and Scheduler universe jobs, the Schedd that was queried is used using a variable internal to *condor_q*. For grid uiniverse jobs, the EC2RemoteVirtualMachineName or GridResources attributes are used. for all other universes the RemoteHost job attribute is used.

STDU_GOODPUT
Used for the GOODPUT column of the -goodput output of *condor_q*. This function renders a floating point goodput time in seconds from the JobStatus, CommittedTime, ShadowBDay, LastCkptTime, and RemoteWallClockTime attributes.

STDU_MPBS
Used for the Mb/s column of the -goodput output of *condor_q*. This function renders a Megabytes per second goodput value from the BytesSent, BytesRecvd job attributes and total job execution time as calculated by the STDU_GOODPUT output.

6.3.4 PRINTAS functions for *condor_status*

ACTIVITY_CODE
Render a two character machine state and activity code from the State and Activity attributes of the machine ad. The letter codes for State are:

~	None
O	Owner
U	Unclaimed
M	Matched
C	Claimed
P	Preempting
S	Shutdown
X	Delete
F	Backfill
D	Drained
#	<undefined>
?	<error>

The letter codes for Activity are:

0	None
i	Idle
b	Busy
r	Retiring
v	Vacating
s	Suspended
b	Benchmarking
k	Killing
#	<undefined>
?	<error>

For example if `State` is Claimed and `Activity` is Idle, then this function returns Ci.

ACTIVITY_TIME
Used for the `ActvtyTime` column of the default output of *condor_status*. The funciton renders the given Unix timestamp as an elapsed time since the `MyCurrentTime` or `LastHeardFrom` attribute.

CONDOR_PLATFORM
Used for the optional `Platform` column of the `-master` output of *condor_status*. This function extracts the Arch and Opsys information from the given string.

CONDOR_VERSION
Used for the `Version` column of the `-master` output of *condor_status*. This function extract the version number and build id from the given string.

DATE
This function converts a Unix timestamp to a string date and time with 2 digit month, day, hour and minute values.

DUE_DATE
This function converts an elapsed time to a Unix timestamp by adding the `LastHeardFrom` attribute to it, and then converts it to a string date and time with 2 digit month, day, hour and minute values.

ELAPSED_TIME
Used in multiple places, for insance the `Uptime` column of the `-master` output of *condor_status*. This function converts a Unix timestamp to an elapsed time by subtracting it from the `LastHeardFrom` attribute, then formats it as a human readable elapsed time.

LOAD_AVG
Used for the `LoadAv` column of the default output of *condor_status* Render the given floating point value using `%.3f` format.

PLATFORM
Used for the `Platform` column of the `-compact` output of *condor_status*. Render a compact platform name from the value of the `OpSys`, `OpSysAndVer`, `OpSysShortName` and `Arch` attributes.

READABLE_KB
This function renders a numeric Kibibyte value into a string with an appropriate B, KB, MB, GB, or TB suffix. Use this for Job attributes that are valued in Kb, such as `DiskUsage`.

READABLE_MB
This function renders a numeric Mibibyte value into a string with an appropriate B, KB, MB, GB, or TB suffix. Use this for Job attributes that are valued in Mb, such as `MemoryUsage`.

STRINGS_FROM_LIST
Used for the `Offline Universes` column of the `-offline` output of *condor_status*. This function converts a ClassAd list into a string containing a comma separated list of items.

TIME
: Used for the `KbdIdle` column of the default output of *condor_status*. This function converts a numeric time in seconds into a string time including number of days, hours, minutes and seconds.

UNIQUE
: Used for the `Users` column of the compact `-claimed` output of *condor_status* This function converts a classad list into a string containing a comma separate list of unique items.

CHAPTER

SEVEN

DAGMAN WORKFLOWS

7.1 DAGMan Introduction

DAGMan is a HTCondor tool that allows multiple jobs to be organized in **workflows**, represented as a directed acyclic graph (DAG). A DAGMan workflow automatically submits jobs in a particular order, such that certain jobs need to complete before others start running. This allows the outputs of some jobs to be used as inputs for others, and makes it easy to replicate a workflow multiple times in the future.

7.1.1 Describing Workflows with DAGMan

A DAGMan workflow is described in a **DAG input file**. The input file specifies the nodes of the DAG as well as the dependencies that order the DAG.

A **node** within a DAG represents a unit of work. It contains the following:

- **Job**: An HTCondor job, defined in a submit file.
- **PRE script** (optional): A script that runs before the job starts. Typically used to verify that all inputs are valid.
- **POST script** (optional): A script that runs after the job finishes. Typically used to verify outputs and clean up temporary files.

The following diagram illustrates the elements of a node – every node must contain a job, with an optional pre and an optional post script.

An **edge** in DAGMan describes a dependency between two nodes. DAG edges are directional; each has a **parent** and a **child**, where the parent node must finish running before the child starts. Any node can have an unlimited number of parents and children.

7.1.2 Example: Diamond DAG

A simple diamond-shaped DAG, as shown in the following image is presented as a starting point for examples. This diamond DAG contains 4 nodes.

A very simple DAG input file for this diamond-shaped DAG is:

```
# File name: diamond.dag

JOB  A  A.sub
```

(continues on next page)

(continued from previous page)

```
JOB  B  B.sub
JOB  C  C.sub
JOB  D  D.sub
PARENT A CHILD B C
PARENT B C CHILD D
```

A set of basic commands appearing in a DAG input file is described below.

7.1.3 JOB

The **JOB** command specifies an HTCondor job. The syntax used for each *JOB* command is:

```
JOB JobName SubmitDescriptionFileName [DIR directory] [NOOP] [DONE]
```

A *JOB* entry maps a *JobName* to an HTCondor submit description file. The *JobName* uniquely identifies nodes within the DAG input file and in output messages. Each node name, given by *JobName*, within the DAG must be unique.

The values defined for *JobName* and *SubmitDescriptionFileName* are case sensitive, as file names in a file system are case sensitive. The *JobName* can be any string that contains no white space, except for the strings *PARENT* and *CHILD* (in upper, lower, or mixed case). *JobName* also cannot contain special characters (‘.’, ‘+’) which are reserved for system use.

The optional *DIR* keyword specifies a working directory for this node, from which the HTCondor job will be submitted, and from which a *PRE* and/or *POST* script will be run. If a relative directory is specified, it is relative to the current working directory as the DAG is submitted. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*.

The optional *NOOP* keyword identifies that the HTCondor job within the node is not to be submitted to HTCondor. This is useful for debugging a complex DAG structure, by marking jobs as *NOOP* to verify that the control flow through the DAG is correct. The *NOOP* keywords are then removed before submitting the DAG. Any PRE and POST scripts for jobs specified with *NOOP* are executed; to avoid running the PRE and POST scripts, comment them out. Even though the job specified with *NOOP* is not submitted, its submit description file must still exist.

The optional *DONE* keyword identifies a node as being already completed. This is mainly used by Rescue DAGs generated by DAGMan itself, in the event of a failure to complete the workflow. Users should generally not use the *DONE* keyword. The *NOOP* keyword is more flexible in avoiding the execution of a job within a node.

7.1.4 PARENT/CHILD Relationships

The **PARENT ... CHILD ...** command specifies the dependencies within the DAG. Nodes are parents and/or children within the DAG. A parent node must be completed successfully before any of its children may be started. A child node may only be started once all its parents have successfully completed.

The syntax used for each dependency (PARENT/CHILD) command is

```
PARENT ParentJobName [ParentJobName2 ... ] CHILD  ChildJobName [ChildJobName2 ... ]
```

The *PARENT* keyword is followed by one or more *ParentJobName(s)*. The *CHILD* keyword is followed by one or more *ChildJobName(s)*. Each child job depends on every parent job within the line. A single line in the input file can specify the dependencies from one or more parents to one or more children. The diamond-shaped DAG example may specify the dependencies with

```
PARENT A CHILD B C
PARENT B C CHILD D
```

An alternative specification for the diamond-shaped DAG may specify some or all of the dependencies on separate lines:

```
PARENT A CHILD B C
PARENT B CHILD D
PARENT C CHILD D
```

As a further example, the line

```
PARENT p1 p2 CHILD c1 c2
```

produces four dependencies:

1. p1 to c1
2. p2 to c1
3. p1 to c2
4. p2 to c2

7.1.5 Node Job Submit File Contents

SUBMIT-DESCRIPTION command

In addition to declaring inline submit descriptions as part of a job, they can be declared independently of jobs using the *SUBMIT-DESCRIPTION* command. This can be helpful to reduce the size and readability of a .dag file when many nodes are running the same job.

A *SUBMIT-DESCRIPTION* can be defined using the following syntax:

```
SUBMIT-DESCRIPTION DescriptionName {
    # submit attributes go here
}
```

An independently declared submit description must have a unique name that is not used by any of the jobs. It can then be linked to a job as follows:

```
JOB JobName DescriptionName
```

For example, the previous diamond.dag example could be written as follows:

```
# File name: diamond.dag

SUBMIT-DESCRIPTION DiamondDesc {
    executable   = /path/diamond.exe
    output       = diamond.out.$(cluster)
    error        = diamond.err.$(cluster)
    log          = diamond_condor.log
```

(continues on next page)

(continued from previous page)

```
    universe     = vanilla
}

JOB A DiamondDesc
JOB B DiamondDesc
JOB C DiamondDesc
JOB D DiamondDesc

PARENT A CHILD B C
PARENT B C CHILD D
```

Inline Submit Descriptions

Instead of using a submit description file, you can alternatively include an inline submit description directly inside the .dag file. An inline submit description should be wrapped in { and } braces, with each argument appearing on a separate line, just like the contents of a regular submit file. Using the previous diamond-shaped DAG example, the diamond.dag file would look like this:

```
# File name: diamond.dag

JOB  A  {
    executable   = /path/diamond.exe
    output       = diamond.out.$(cluster)
    error        = diamond.err.$(cluster)
    log          = diamond_condor.log
    universe     = vanilla
}
JOB  B  {
    executable   = /path/diamond.exe
    output       = diamond.out.$(cluster)
    error        = diamond.err.$(cluster)
    log          = diamond_condor.log
    universe     = vanilla
}
JOB  C  {
    executable   = /path/diamond.exe
    output       = diamond.out.$(cluster)
    error        = diamond.err.$(cluster)
    log          = diamond_condor.log
    universe     = vanilla
}
JOB  D  {
    executable   = /path/diamond.exe
    output       = diamond.out.$(cluster)
    error        = diamond.err.$(cluster)
    log          = diamond_condor.log
    universe     = vanilla
}
PARENT A CHILD B C
PARENT B C CHILD D
```

This can be helpful when trying to manage lots of submit descriptions, so they can all be described in the same file

instead of needed to regularly shift between many files.

The main drawback of using inline submit descriptions is that they do not support the `queue` statement or any variations thereof. Any job described inline in the .dag file will only have a single instance submitted.

External File Descriptions

Each node in a DAG may use a unique submit description file. A key limitation is that each HTCondor submit description file must submit jobs described by a single cluster number; DAGMan cannot deal with a submit description file producing multiple job clusters.

Consider again the diamond-shaped DAG example, where each node job uses the same submit description file.

```
# File name: diamond.dag

JOB  A  diamond_job.sub
JOB  B  diamond_job.sub
JOB  C  diamond_job.sub
JOB  D  diamond_job.sub
PARENT A CHILD B C
PARENT B C CHILD D
```

Here is a sample HTCondor submit description file for this DAG:

```
# File name: diamond_job.sub

executable   = /path/diamond.exe
output       = diamond.out.$(cluster)
error        = diamond.err.$(cluster)
log          = diamond_condor.log
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

Since each node uses the same HTCondor submit description file, this implies that each node within the DAG runs the same job. The `$(Cluster)` macro produces unique file names for each job's output.

The job ClassAd attribute `DAGParentNodeNames` is also available for use within the submit description file. It defines a comma separated list of each *JobName* which is a parent node of this job's node. This attribute may be used in the **arguments** command for all but scheduler universe jobs. For example, if the job has two parents, with *JobName*s B and C, the submit description file command

```
arguments = $$([DAGParentNodeNames])
```

will pass the string `"B,C"` as the command line argument when invoking the job.

DAGMan supports jobs with queues of multiple procs, so for example:

```
queue 500
```

will queue 500 procs as expected.

7.2 Scripts

The optional *SCRIPT* command specifies processing that is done either before a job within a node is submitted, after a job within a node completes its execution, or when a job goes on hold. All scripts run on the Access Point and not the Execution Point where the node job is likely to run.

7.2.1 PRE and POST scripts

Processing done before a job is submitted is called a *PRE* script. Processing done after a job completes its execution is called a *POST* script. Note that the executable specified does not necessarily have to be a shell script (Unix) or batch file (Windows); but it should be relatively light weight because it will be run directly on the access point, not submitted as an HTCondor job.

The syntax used for *PRE* or *POST* commands is

```
SCRIPT [DEFER status time] PRE <JobName | ALL_NODES> ExecutableName [arguments]
```

```
SCRIPT [DEFER status time] POST <JobName | ALL_NODES> ExecutableName [arguments]
```

The *SCRIPT* command can use the *PRE* or *POST* keyword, which specifies the relative timing of when the script is to be run. The *JobName* identifies the node to which the script is attached. The *ExecutableName* specifies the executable (e.g., shell script or batch file) to be executed, and may not contain spaces. The optional *arguments* are command line arguments to the script, and spaces delimiting the arguments. Both *ExecutableName* and optional *arguments* are case sensitive.

A PRE script is commonly used to place files in a staging area for the jobs to use. A POST script is commonly used to clean up or remove files once jobs are finished running. An example uses PRE and POST scripts to stage files that are stored on tape. The PRE script reads compressed input files from the tape drive, uncompresses them, and places the resulting files in the current directory. The HTCondor jobs can then use these files, producing output files. The POST script compresses the output files, writes them out to the tape, and then removes both the staged files and the output files.

7.2.2 HOLD scripts

Additionally, the *SCRIPT* command can take a *HOLD* keyword, which indicates an executable to be run when a job goes on hold. These are typically used to notify a user when something goes wrong with their jobs.

The syntax used for a *HOLD* command is

```
SCRIPT [DEFER status time] HOLD <JobName | ALL_NODES> ExecutableName [arguments]
```

Unlike *PRE* and *POST* scripts, *HOLD* scripts are not considered part of the DAG workflow and are run on a best-effort basis. If one does not complete successfully, it has no effect on the overall workflow and no error will be reported.

7.2.3 DEFER retries

The optional *DEFER* keyword causes a retry of only the script, if the execution of the script exits with the exit code given by *status*. The retry occurs after at least *time* seconds, rather than being considered failed. While waiting for the retry, the script does not count against a *maxpre* or *maxpost* limit. The ordering of the *DEFER* keyword within the *SCRIPT* specification is fixed. It must come directly after the *SCRIPT* keyword; this is done to avoid backward compatibility issues for any DAG with a *JobName* of DEFER.

7.2.4 Scripts as part of a DAG workflow

Scripts are executed on the access point; the access point is not necessarily the same machine upon which the node's job is run. Further, a single cluster of HTCondor jobs may be spread across several machines.

If the PRE script fails, then the HTCondor job associated with the node is not submitted, and (as of version 8.5.4) the POST script is not run either (by default). However, if the job is submitted, and there is a POST script, the POST script is always run once the job finishes. (The behavior when the PRE script fails may be changed to run the POST script by setting configuration variable to `True` or by passing the **-AlwaysRunPost** argument to *condor_submit_dag*.)

7.2.5 Examples that use PRE or POST scripts

Examples use the diamond-shaped DAG. A first example uses a PRE script to expand a compressed file needed as input to each of the HTCondor jobs of nodes B and C. The DAG input file:

```
# File name: diamond.dag

JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
SCRIPT PRE  B  pre.sh $JOB .gz
SCRIPT PRE  C  pre.sh $JOB .gz
PARENT A CHILD B C
PARENT B C CHILD D
```

The script `pre.sh` uses its command line arguments to form the file name of the compressed file. The script contains

```
#!/bin/sh
gunzip ${1}${2}
```

Therefore, the PRE script invokes

```
gunzip B.gz
```

for node B, which uncompresses file `B.gz`, placing the result in file `B`.

A second example uses the `$RETURN` macro. The DAG input file contains the POST script specification:

```
SCRIPT POST A stage-out job_status $RETURN
```

If the HTCondor job of node A exits with the value -1, the POST script is invoked as

```
stage-out job_status -1
```

The slightly different example POST script specification in the DAG input file

```
SCRIPT POST A stage-out job_status=$RETURN
```

invokes the POST script with

```
stage-out job_status=$RETURN
```

This example shows that when there is no space between the = sign and the variable `$RETURN`, there is no substitution of the macro's value.

7.2.6 Special script argument macros

The five macros `$JOB`, `$RETRY`, `$MAX_RETRIES`, `$DAG_STATUS` and `$FAILED_COUNT` can be used within the DAG input file as arguments passed to a PRE or POST script. An additional three macros `$JOBID`, `$RETURN`, and `$PRE_SCRIPT_RETURN` can be used as arguments to POST scripts. The use of these variables is limited to being used as an individual command line *argument* to the script, surrounded by spaces, in order to cause the substitution of the variable's value.

The special macros are as follows:

- `$JOB` evaluates to the (case sensitive) string defined for *JobName*.
- `$RETRY` evaluates to an integer value set to 0 the first time a node is run, and is incremented each time the node is retried. See *Retrying Failed Nodes* for the description of how to cause nodes to be retried.
- `$MAX_RETRIES` evaluates to an integer value set to the maximum number of retries for the node. See *Retrying Failed Nodes* for the description of how to cause nodes to be retried. If no retries are set for the node, `$MAX_RETRIES` will be set to 0.
- `$JOBID` (for POST scripts only) evaluates to a representation of the HTCondor job ID of the node job. It is the value of the job ClassAd attribute `ClusterId`, followed by a period, and then followed by the value of the job ClassAd attribute `ProcId`. An example of a job ID might be 1234.0. For nodes with multiple jobs in the same cluster, the `ProcId` value is the one of the last job within the cluster.
- `$RETURN` (for POST scripts only) variable evaluates to the return value of the HTCondor job, if there is a single job within a cluster. With multiple jobs within the same cluster, there are two cases to consider. In the first case, all jobs within the cluster are successful; the value of `$RETURN` will be 0, indicating success. In the second case, one or more jobs from the cluster fail. When *condor_dagman* sees the first terminated event for a job that failed, it assigns that job's return value as the value of `$RETURN`, and it attempts to remove all remaining jobs within the cluster. Therefore, if multiple jobs in the cluster fail with different exit codes, a race condition determines which exit code gets assigned to `$RETURN`.

  A job that dies due to a signal is reported with a `$RETURN` value representing the additive inverse of the signal number. For example, SIGKILL (signal 9) is reported as -9. A job whose batch system submission fails is reported as -1001. A job that is externally removed from the batch system queue (by something other than *condor_dagman*) is reported as -1002.
- `$PRE_SCRIPT_RETURN` (for POST scripts only) variable evaluates to the return value of the PRE script of a node, if there is one. If there is no PRE script, this value will be -1. If the node job was skipped because of failure of the PRE script, the value of `$RETURN` will be -1004 and the value of `$PRE_SCRIPT_RETURN` will be the exit value of the PRE script; the POST script can use this to see if the PRE script exited with an error condition, and assign success or failure to the node, as appropriate.
- `$DAG_STATUS` is the status of the DAG. Note that this macro's value and definition is unrelated to the attribute named `DagStatus` as defined for use in a node status file. This macro's value is the same as the job ClassAd attribute `DAG_Status` that is defined within the *condor_dagman* job's ClassAd. This macro may have the following values:

  - 0: OK
  - 1: error; an error condition different than those listed here
  - 2: one or more nodes in the DAG have failed
  - 3: the DAG has been aborted by an ABORT-DAG-ON specification
  - 4: removed; the DAG has been removed by *condor_rm*
  - 5: cycle; a cycle was found in the DAG
  - 6: halted; the DAG has been halted (see *Suspending a Running DAG*)
- `$FAILED_COUNT` is defined by the number of nodes that have failed in the DAG.

7.3 Node Success/Failure

Progress towards completion of the DAG is based upon the success of the nodes within the DAG. The success of a node is based upon the success of the job(s), PRE script, and POST script. A job, PRE script, or POST script with an exit value not equal to 0 is considered failed. **The exit value of whatever component of the node was run last determines the success or failure of the node.**

Table 2.1 lists the definition of node success and failure for all variations of script and job success and failure, when is set to `False`. In this table, a dash (-) represents the case where a script does not exist for the DAG, **S** represents success, and **F** represents failure.

PRE	JOB	POST	Node
-	S	-	**S**
-	F	-	**F**
-	S	S	**S**
-	S	F	**F**
-	F	S	**S**
-	F	F	**F**
S	S	-	**S**
S	F	-	**F**
S	S	S	**S**
S	S	F	**F**
S	F	S	**S**
S	F	F	**F**
F	not run	-	**F**
F	not run	not run	**F**

Table 2.1: Node **S**uccess or **F**ailure definition with `DAGMAN_ALWAYS_RUN_POST = False` (the default).

Table 2.2 lists the definition of node success and failure only for the cases where the PRE script fails, when is set to `True`.

PRE	JOB	POST	Node
F	not run	-	**F**
F	not run	S	**S**
F	not run	F	**F**

Table 2.2: Node **S**uccess or **F**ailure definition with `DAGMAN_ALWAYS_RUN_POST = True`.

7.3.1 PRE_SKIP

The behavior of DAGMan with respect to node success or failure can be changed with the addition of a *PRE_SKIP* command. A *PRE_SKIP* line within the DAG input file uses the syntax:

```
PRE_SKIP <JobName | ALL_NODES> non-zero-exit-code
```

The PRE script of a node identified by *JobName* that exits with the value given by *non-zero-exit-code* skips the remainder of the node entirely. Neither the job associated with the node nor the POST script will be executed, and the node will be marked as successful.

7.3.2 Retrying Failed Nodes

DAGMan can retry any failed node in a DAG by specifying the node in the DAG input file with the *RETRY* command. The use of retry is optional. The syntax for retry is

```
RETRY <JobName | ALL_NODES> NumberOfRetries [UNLESS-EXIT value]
```

where *JobName* identifies the node. *NumberOfRetries* is an integer number of times to retry the node after failure. The implied number of retries for any node is 0, the same as not having a retry line in the file. Retry is implemented on nodes, not parts of a node.

The diamond-shaped DAG example may be modified to retry node C:

```
# File name: diamond.dag

JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
RETRY  C 3
```

If node C is marked as failed for any reason, then it is started over as a first retry. The node will be tried a second and third time, if it continues to fail. If the node is marked as successful, then further retries do not occur.

Retry of a node may be short circuited using the optional keyword *UNLESS-EXIT*, followed by an integer exit value. If the node exits with the specified integer exit value, then no further processing will be done on the node.

The macro `$(RETRY)` evaluates to an integer value, set to 0 first time a node is run, and is incremented each time for each time the node is retried. The macro `$(MAX_RETRIES)` is the value set for *NumberOfRetries*. These macros may be used as arguments passed to a PRE or POST script.

7.3.3 Stopping the DAG on Node Failure

The *ABORT-DAG-ON* command provides a way to abort the entire DAG if a given node returns a specific exit code. The syntax for *ABORT-DAG-ON* is

```
ABORT-DAG-ON <JobName | ALL_NODES> AbortExitValue [RETURN DAGReturnValue]
```

If the return value of the node specified by *JobName* matches *AbortExitValue*, the DAG is immediately aborted. A DAG abort differs from a node failure, in that a DAG abort causes all nodes within the DAG to be stopped immediately. This includes removing the jobs in nodes that are currently running. A node failure differs, as it would allow the DAG to continue running, until no more progress can be made due to dependencies.

The behavior differs based on the existence of PRE and/or POST scripts. If a PRE script returns the *AbortExitValue* value, the DAG is immediately aborted. If the HTCondor job within a node returns the *AbortExitValue* value, the DAG is aborted if the node has no POST script. If the POST script returns the *AbortExitValue* value, the DAG is aborted.

An abort overrides node retries. If a node returns the abort exit value, the DAG is aborted, even if the node has retry specified.

When a DAG aborts, by default it exits with the node return value that caused the abort. This can be changed by using the optional *RETURN* keyword along with specifying the desired *DAGReturnValue*. The DAG abort return value can be used for DAGs within DAGs, allowing an inner DAG to cause an abort of an outer DAG.

A DAG return value other than 0, 1, or 2 will cause the *condor_dagman* job to stay in the queue after it exits and get retried, unless the `on_exit_remove` expression in the `.condor.sub` file is manually modified.

Adding *ABORT-DAG-ON* for node C in the diamond-shaped DAG

```
# File name: diamond.dag

JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
RETRY  C 3
ABORT-DAG-ON C 10 RETURN 1
```

causes the DAG to be aborted, if node C exits with a return value of 10. Any other currently running nodes, of which only node B is a possibility for this particular example, are stopped and removed. If this abort occurs, the return value for the DAG is 1.

7.4 File Paths in DAGs

condor_dagman assumes that all relative paths in a DAG input file and the associated HTCondor submit description files are relative to the current working directory when *condor_submit_dag* is run. This works well for submitting a single DAG. It presents problems when multiple independent DAGs are submitted with a single invocation of *condor_submit_dag*. Each of these independent DAGs would logically be in its own directory, such that it could be run or tested independent of other DAGs. Thus, all references to files will be designed to be relative to the DAG's own directory.

Consider an example DAG within a directory named `dag1`. There would be a DAG input file, named `one.dag` for this example. Assume the contents of this DAG input file specify a node job with

```
JOB A  A.submit
```

Further assume that partial contents of submit description file `A.submit` specify

```
executable = programA
input      = A.input
```

Directory contents are

```
dag1/
├── A.input
├── A.submit
├── one.dag
└── programA
```

All file paths are correct relative to the `dag1` directory. Submission of this example DAG sets the current working directory to `dag1` and invokes *condor_submit_dag*:

```
$ cd dag1
$ condor_submit_dag one.dag
```

Expand this example such that there are now two independent DAGs, and each is contained within its own directory. For simplicity, assume that the DAG in `dag2` has remarkably similar files and file naming as the DAG in `dag1`. Assume that the directory contents are

```
parent/
├── dag1
│   ├── A.input
│   ├── A.submit
│   ├── one.dag
│   └── programA
└── dag2
    ├── B.input
    ├── B.submit
    ├── programB
    └── two.dag
```

The goal is to use a single invocation of *condor_submit_dag* to run both dag1 and dag2. The invocation

```
$ cd parent
$ condor_submit_dag dag1/one.dag dag2/two.dag
```

does not work. Path names are now relative to `parent`, which is not the desired behavior.

The solution is the *-usedagdir* command line argument to *condor_submit_dag*. This feature runs each DAG as if *condor_submit_dag* had been run in the directory in which the relevant DAG file exists. A working invocation is

```
$ cd parent
$ condor_submit_dag -usedagdir dag1/one.dag dag2/two.dag
```

Output files will be placed in the correct directory, and the `.dagman.out` file will also be in the correct directory. A Rescue DAG file will be written to the current working directory, which is the directory when *condor_submit_dag* is invoked. The Rescue DAG should be run from that same current working directory. The Rescue DAG includes all the path information necessary to run each node job in the proper directory.

Use of *-usedagdir* does not work in conjunction with a JOB node specification within the DAG input file using the *DIR* keyword. Using both will be detected and generate an error.

7.5 Running and Managing DAGMan

Once once a workflow has been setup in a `.dag` file, all that is left is to submit the prepared workflow. A key concept to understand regarding the submission and management of a DAGMan workflow is that the DAGMan process itself is ran as a HTCondor job (often referred to as the DAGMan proper job) that will in turn manage and submit all the various jobs and scripts defined in the workflow.

7.5.1 DAG Submission

A DAG is submitted using the tool *condor_submit_dag*. The manual page for *condor_submit_dag* details the command. The simplest of DAG submissions has the syntax

```
$ condor_submit_dag DAGInputFileName
```

and the current working directory contains the DAG input file.

The diamond-shaped DAG example may be submitted with

```
$ condor_submit_dag diamond.dag
```

Do not submit the same DAG, with same DAG input file, from within the same directory, such that more than one of this same DAG is running at the same time. It will fail in an unpredictable manner, as each instance of this same DAG will attempt to use the same file to enforce dependencies.

To increase robustness and guarantee recoverability, the *condor_dagman* process is run as an HTCondor job. As such, it needs a submit description file. *condor_submit_dag* generates this needed submit description file, naming it by appending `.condor.sub` to the name of the DAG input file. This submit description file may be edited if the DAG is submitted with

```
$ condor_submit_dag -no_submit diamond.dag
```

causing *condor_submit_dag* to create the submit description file, but not submit *condor_dagman* to HTCondor. To submit the DAG, once the submit description file is edited, use

```
$ condor_submit diamond.dag.condor.sub
```

Since the *condor_dagman* process is an actual HTCondor job, the job Cluster Id produced for this DAGMan proper job is used to help mark all jobs ran by DAGMan. This is done by adding the job classad attribute `DAGManJobId` for all submitted jobs to the produced Job Id.

7.5.2 DAG Monitoring

After submission, the progress of the DAG can be monitored by looking at the job event log file(s), observing the e-mail that job submission to HTCondor causes, or by using *condor_q*.

Using just *condor_q* while a DAGMan workflow is running will display condensed information regarding the overall workflow progress under the DAGMan proper job as follows:

```
$ condor_q
$ OWNER   BATCH_NAME          SUBMITTED   DONE  RUN  IDLE  TOTAL  JOB_IDS
$ Cole    diamond.dag+1024    1/1 12:34   1     2    -     4      1025.0 ... 1026.0
```

Using *condor_q* with the *-dag* and *-nobatch* flags will display information about the DAGMan proper job and all jobs currently submitted/running as part of the DAGMan workflow as follows:

```
$ condor_q -dag -nobatch
$ ID       OWNER/NODENAME  SUBMITTED    RUN_TIME ST PRI SIZE CMD
$ 1024.0   Cole            1/1 12:34  0+01:13:19 R  0   0.4  condor_dagman ...
$ 1025.0    |-Node_B       1/1 13:44  0+00:03:19 R  0   0.4  diamond.sh ...
$ 1026.0    |-Node_C       1/1 13:45  0+00:02:19 R  0   0.4  diamond.sh ...
```

In addition to basic job management, the DAGMan proper job holds a lot of extra information within its job classad that can queried with the *-l* or the more recommended *-af <Attributes>* flags for *condor_q* in association with the DAGMan proper Job Id.

```
$ condor_q <dagman-job-id> -af Attribute-1 ... Attribute-N
$ condor_q -l <dagman-job-id>
```

There is also a large amount of information logged in an extra file. The name of this extra file is produced by appending `.dagman.out` to the name of the DAG input file; for example, if the DAG input file is `diamond.dag`, this extra file is named `diamond.dag.dagman.out`. The `.dagman.out` file is an important resource for debugging; save this file if a problem occurs. The `dagman.out` is appended to, rather than overwritten, with each new DAGMan run.

Status Information for the DAG in a ClassAd

The *condor_dagman* job places information about the status of the DAG into its own job ClassAd. The attributes are fully described in *Job ClassAd Attributes*. The attributes are

DAG Info	DAG_Status	DAG_InRecovery
	DAG_AdUpdateTime	
Node Info	DAG_NodesTotal	DAG_NodesDone
	DAG_NodesPrerun	DAG_NodesPostrun
	DAG_NodesReady	DAG_NodesUnready
	DAG_NodesFailed	DAG_NodesFutile
	DAG_NodesQueued	
DAG Process Info	DAG_JobsSubmitted	DAG_JobsCompleted
	DAG_JobsIdle	DAG_JobsRunning
	DAG_JobsHeld	

Note that most of this information is also available in the `dagman.out` file.

7.5.3 Editing a Running DAG

Certain properties of a running DAG can be changed after the workflow has been started. The values of these properties are published in the *condor_dagman* job ad; changing any of these properties using *condor_qedit* will also update the internal DAGMan value.

Currently, you can change the following attributes:

Attribute Name	Attribute Description
DAGMan_MaxJobs	Maximum number of running jobs
DAGMan_MaxIdle	Maximum number of idle jobs
DAGMan_MaxPreScripts	Maximum number of running PRE scripts
DAGMan_MaxPostScripts	Maximum number of running POST scripts

To edit one of these properties, use the *condor_qedit* tool with the job ID of the *condor_dagman* job, for example:

```
$ condor_qedit <dagman-job-id> DAGMan_MaxJobs 1000
```

To view all the properties of a *condor_dagman* job:

```
$ condor_q -l <dagman-job-id> | grep DAG
```

7.5.4 Removing a DAG

To remove an entire DAG, consisting of the *condor_dagman* job, plus any jobs submitted to HTCondor, remove the *condor_dagman* job by running *condor_rm*. For example,

```
$ condor_q -nobatch
-- Submitter: user.cs.wisc.edu : <128.105.175.125:36165> : user.cs.wisc.edu
 ID      OWNER          SUBMITTED     RUN_TIME ST PRI SIZE CMD
  9.0   taylor         10/12 11:47   0+00:01:32 R  0   8.7  condor_dagman -f ...
 11.0   taylor         10/12 11:48   0+00:00:00 I  0   3.6  B.out
 12.0   taylor         10/12 11:48   0+00:00:00 I  0   3.6  C.out

    3 jobs; 2 idle, 1 running, 0 held

$ condor_rm 9.0
```

When a *condor_dagman* job is removed, all node jobs (including sub-DAGs) of that *condor_dagman* will be removed by the *condor_schedd*. As of version 8.5.8, the default is that *condor_dagman* itself also removes the node jobs (to fix a race condition that could result in "orphaned" node jobs). (The *condor_schedd* has to remove the node jobs to deal with the case of removing a *condor_dagman* job that has been held.)

The previous behavior of *condor_dagman* itself not removing the node jobs can be restored by setting the configuration macro to `False`. This will decrease the load on the *condor_schedd*, at the cost of allowing the possibility of "orphaned" node jobs.

A removed DAG will be considered failed unless the DAG has a FINAL node that succeeds.

In the case where a machine is scheduled to go down, DAGMan will clean up memory and exit. However, it will leave any submitted jobs in the HTCondor queue.

7.5.5 Suspending a Running DAG

It may be desired to temporarily suspend a running DAG. For example, the load may be high on the access point, and therefore it is desired to prevent DAGMan from submitting any more jobs until the load goes down. There are two ways to suspend (and resume) a running DAG.

- Use *condor_hold*/*condor_release* on the *condor_dagman* job.

  After placing the *condor_dagman* job on hold, no new node jobs will be submitted, and no PRE or POST scripts will be run. Any node jobs already in the HTCondor queue will continue undisturbed. Any running PRE or POST scripts will be killed. If the *condor_dagman* job is left on hold, it will remain in the HTCondor queue after all of the currently running node jobs are finished. To resume the DAG, use *condor_release* on the *condor_dagman* job.

  Note that while the *condor_dagman* job is on hold, no updates will be made to the `dagman.out` file.

- Use a DAG halt file.

  The second way of suspending a DAG uses the existence of a specially-named file to change the state of the DAG. When in this halted state, no PRE scripts will be run, and no node jobs will be submitted. Running node jobs will continue undisturbed. A halted DAG will still run POST scripts, and it will still update the `dagman.out` file. This differs from behavior of a DAG that is held. Furthermore, a halted DAG will not remain in the queue indefinitely; when all of the running node jobs have finished, DAGMan will create a Rescue DAG and exit.

  To resume a halted DAG, remove the halt file.

  The specially-named file must be placed in the same directory as the DAG input file. The naming is the same as the DAG input file concatenated with the string `.halt`. For example, if the DAG input file is `test1.dag`, then `test1.dag.halt` will be the required name of the halt file.

  As any DAG is first submitted with *condor_submit_dag*, a check is made for a halt file. If one exists, it is removed.

Note that neither condor_hold nor a DAG halt is propagated to sub-DAGs. In other words, if you *condor_hold* or create a halt file for a DAG that has sub-DAGs, any sub-DAGs that are already in the queue will continue to submit node jobs.

A *condor_hold* or DAG halt does, however, apply to splices, because they are merged into the parent DAG and controlled by a single *condor_dagman* instance.

7.6 DAG Save Point Files

A DAG can be set up to write the current progress of the DAG at specified nodes to a save point file. These files are written the first time the designated node starts running. Meaning any retries won't save the DAG progress again. The save point file is written in the exact same format as a partial Rescue DAG except that all node retry values will be reset to their max value. The DAG save point file can then be specified when re-running a DAG to start the DAG at a certain point of progress.

To specify a save point file use the DAG submit description keyword `SAVE_POINT_FILE` followed by the name of the node designated as the save point to write a save file, and optionally a filename. If a filename is not specified the file will be written as `[Node Name]-[DAG filename].save` where the DAG filename is the DAG file that the save file declaration was read from.

If the specified save point filename includes a path then DAGMan will attempt to write the file to that location. If the *condor_submit_dag* `useDagDir` flag is used and a path is specified for a save point then the file will be written to that path relative to a DAG's working directory. Any save point files without a specified path will be written to a sub-directory called `save_files` created near all other DAGMan procuded files (i.e. `.condor.sub`, `.dagman.out`, etc.).

```
# File: savepointEx.dag
JOB A node.sub
JOB B node.sub
JOB C node.sub
JOB D node.sub

PARENT A B C CHILD D

#SAVE_POINT_FILE NodeName Filename
SAVE_POINT_FILE A
SAVE_POINT_FILE B Node-B_custom.save
SAVE_POINT_FILE C ../example/subdir/Node-C_custom.save
SAVE_POINT_FILE D ./Node-D_custom.save
```

Given the above example DAG file, if `condor_submit_dag savepointEx.dag` was ran from the below directory `my_work` then the produced files appear in the directory tree as follows:

```
Directory Tree Visualized
└─Home
    ├─example
    │   └─subdir
    │       └─Node-C_custom.save
    └─my_work
        ├─savepointEx.dag
        ├─savepointEx.dag.condor.sub
        ├─savepointEx.dag.dagman.out
        ├─...
        ├─Node-D_custom.save
        └─save_files
              ├─ A-savepointEx.dag.save
              └─ Node-B_custom.save
```

Once a DAG has ran and produce save point files, the DAG can then be re-run from a save file by passing a filename via the `-load_save` flag for *condor_submit_dag*. If the save point file is passed with a specified path then DAGMan will attempt to read the file from that path. If just a save point filename is given then DAGMan will assume the file is located in the``save_files`` directory. The path to save point files will be checked relative to the current working directory that *condor_submit_dag* was ran from.

When DAGMan writes save point files, if a save file with the same name already exists then DAGMan will rotate the file to `[filename].old` before writing the new save. Any already existing "old" save files will be removed prior to rotation and saving. So, if the above example DAG was re-run with `condor_submit_dag -load_save ./Node-D_custom.save savepointEx.dag` from the same directory then once node D starts the previous save would become `Node-D_custom.save.old`. This behavior does not just effect save point files when re-running a DAG. If a DAG was set up as follows:

```
# File: progressSavefile.dag
JOB A node.sub
JOB B node.sub
JOB C node.sub
...
SAVE_POINT_FILE A dag-progress.save
SAVE_POINT_FILE B dag-progress.save
SAVE_POINT_FILE C dag-progress.save
```

Then assuming the parent/child relationships is `A->B->C`, the first save written at the start of node A will be written to `dag-progress.save`. Then when node B starts the present `dag-progress.save` will become `dag-progress.save.old` and a new `dag-progress.save` will be written. Finally, once node C starts `dag-progress.save.old` will be deleted, the present `dag-progress.save` will become `dag-progress.save.old` and a new `dag-progress.save` will be written. Allowing a single save file that progresses with the DAG to be created.

7.7 Resubmitting a Failed DAG

When debugging a DAG in which something has gone wrong, a first determination is whether a resubmission will use a Rescue DAG or benefit from recovery. The existence of a Rescue DAG means that recovery would be inappropriate. A Rescue DAG is has a file name ending in `.rescue<XXX>`, where `<XXX>` is replaced by a 3-digit number.

Determine if a DAG ever completed (independent of whether it was successful or not) by looking at the last lines of the `.dagman.out` file. If there is a line similar to

```
(condor_DAGMAN) pid 445 EXITING WITH STATUS 0
```

then the DAG completed. This line explains that the *condor_dagman* job finished normally. If there is no line similar to this at the end of the `.dagman.out` file, and output from *condor_q* shows that the *condor_dagman* job for the DAG being debugged is not in the queue, then recovery is indicated.

7.7.1 The Rescue DAG

Any time a DAG exits unsuccessfully, DAGMan generates a Rescue DAG. The Rescue DAG records the state of the DAG, with information such as which nodes completed successfully, and the Rescue DAG will be used when the DAG is again submitted. With the Rescue DAG, nodes that have already successfully completed are not re-run.

There are a variety of circumstances under which a Rescue DAG is generated. If a node in the DAG fails, the DAG does not exit immediately; the remainder of the DAG is continued until no more forward progress can be made based on the DAG's dependencies. At this point, DAGMan produces the Rescue DAG and exits. A Rescue DAG is produced on Unix platforms if the *condor_dagman* job itself is removed with *condor_rm*. On Windows, a Rescue DAG is not generated in this situation, but re-submitting the original DAG will invoke a lower-level recovery functionality, and it will produce similar behavior to using a Rescue DAG. A Rescue DAG is produced when a node sets and triggers an *ABORT-DAG-ON* event with a non-zero return value. A zero return value constitutes successful DAG completion, and therefore a Rescue DAG is not generated.

By default, if a Rescue DAG exists, it will be used when the DAG is submitted specifying the original DAG input file. If more than one Rescue DAG exists, the newest one will be used. By using the Rescue DAG, DAGMan will avoid re-running nodes that completed successfully in the previous run. **Note that passing the -force option to condor_submit_dag or condor_dagman will cause condor_dagman to not use any existing rescue DAG. This means that previously-completed node jobs will be re-run.**

The granularity defining success or failure in the Rescue DAG is the node. For a node that fails, all parts of the node will be re-run, even if some parts were successful the first time. For example, if a node's PRE script succeeds, but then the node's HTCondor job cluster fails, the entire node, including the PRE script, will be re-run. A job cluster may result in the submission of multiple HTCondor jobs. If one of the jobs within the cluster fails, the node fails. Therefore, the Rescue DAG will re-run the entire node, implying the submission of the entire cluster of jobs, not just the one(s) that failed.

Statistics about the failed DAG execution are presented as comments at the beginning of the Rescue DAG input file.

Rescue DAG Naming

The file name of the Rescue DAG is obtained by appending the string .rescue<XXX> to the original DAG input file name. Values for <XXX> start at 001 and continue to 002, 003, and beyond. The configuration variable sets a maximum value for <XXX>. If you hit the limit, the last Rescue DAG file is overwritten if the DAG fails again.

If a Rescue DAG exists when the original DAG is re-submitted, the Rescue DAG with the largest magnitude value for <XXX> will be used, and its usage is implied.

Example

Here is an example showing file naming and DAG submission for the case of a failed DAG. The initial DAG is submitted with

```
$ condor_submit_dag my.dag
```

A failure of this DAG results in the Rescue DAG named `my.dag.rescue001`. The DAG is resubmitted using the same command:

```
$ condor_submit_dag my.dag
```

This resubmission of the DAG uses the Rescue DAG file `my.dag.rescue001`, because it exists. Failure of this Rescue DAG results in another Rescue DAG called `my.dag.rescue002`. If the DAG is again submitted, using the same command as with the first two submissions, but not repeated here, then this third submission uses the Rescue DAG file `my.dag.rescue002`, because it exists, and because the value 002 is larger in magnitude than 001.

Using an Older Rescue DAG

To explicitly specify a particular Rescue DAG, use the optional command-line argument *-dorescuefrom* with *condor_submit_dag*. Note that this will have the side effect of renaming existing Rescue DAG files with larger magnitude values of <XXX>. Each renamed file has its existing name appended with the string `.old`. For example, assume that `my.dag` has failed 4 times, resulting in the Rescue DAGs named `my.dag.rescue001`, `my.dag.rescue002`, `my.dag.rescue003`, and `my.dag.rescue004`. A decision is made to re-run using `my.dag.rescue002`. The submit command is

```
$ condor_submit_dag -dorescuefrom 2 my.dag
```

The DAG specified by the DAG input file `my.dag.rescue002` is submitted. The existing Rescue DAG `my.dag.rescue003` is renamed to be `my.dag.rescue003.old`, while the existing Rescue DAG `my.dag.rescue004` is renamed to be `my.dag.rescue004.old`.

Special Cases

Note that if multiple DAG input files are specified on the *condor_submit_dag* command line, a single Rescue DAG encompassing all of the input DAGs is generated. A DAG file containing splices also produces a single Rescue DAG file. On the other hand, a DAG containing sub-DAGs will produce a separate Rescue DAG for each sub-DAG that is queued (and for the top-level DAG).

If the Rescue DAG file is generated before all retries of a node are completed, then the Rescue DAG file will also contain *RETRY* entries. The number of retries will be set to the appropriate remaining number of retries. The configuration variable controls whether or not node retries are reset in a Rescue DAG.

Partial versus Full Rescue DAGs

As of HTCondor version 7.7.2, the Rescue DAG file is a partial DAG file, not a complete DAG input file as in the past.

A partial Rescue DAG file contains only information about which nodes are done and the number of retries remaining for nodes with retries. It does not contain information such as the actual DAG structure and the specification of the submit description file for each node job. Partial Rescue DAGs are automatically parsed in combination with the original DAG input file, which contains information about the DAG structure. This updated implementation means that a change in the original DAG input file, such as specifying a different submit description file for a node job, will take effect when running the partial Rescue DAG. In other words, you can fix mistakes in the original DAG file while still gaining the benefit of using the Rescue DAG.

To use a partial Rescue DAG, you must re-run *condor_submit_dag* on the original DAG file, not the Rescue DAG file.

Note that the existence of a DONE specification in a partial Rescue DAG for a node that no longer exists in the original DAG input file is a warning, as opposed to an error, unless the configuration variable is set to a value of 1 or higher (which is now the default). Comment out the line with *DONE* in the partial Rescue DAG file to avoid a warning or error.

The previous (prior to version 7.7.2) behavior of producing full DAG input file as the Rescue DAG is obtained by setting the configuration variable to `False`. **Note that the option to generate full Rescue DAGs is likely to disappear some time during the 8.3 series.**

To run a full Rescue DAG, either one left over from an older version of DAGMan, or one produced by setting to `False`, directly specify the full Rescue DAG file on the command line instead of the original DAG file. For example:

```
$ condor_submit_dag my.dag.rescue002
```

Attempting to re-submit the original DAG file, if the Rescue DAG file is a complete DAG, will result in a parse failure.

Rescue for Parse Failure

Starting in HTCondor version 7.5.5, passing the **-DumpRescue** option to either *condor_dagman* or *condor_submit_dag* causes *condor_dagman* to output a Rescue DAG file, even if the parsing of a DAG input file fails. In this parse failure case, *condor_dagman* produces a specially named Rescue DAG containing whatever it had successfully parsed up until the point of the parse error. This Rescue DAG may be useful in debugging parse errors in complex DAGs, especially ones using splices. This incomplete Rescue DAG is not meant to be used when resubmitting a failed DAG. Note that this incomplete Rescue DAG generated by the **-DumpRescue** option is a full DAG input file, as produced by versions of HTCondor prior to HTCondor version 7.7.2. It is not a partial Rescue DAG file, regardless of the value of the configuration variable .

To avoid confusion between this incomplete Rescue DAG generated in the case of a parse failure and a usable Rescue DAG, a different name is given to the incomplete Rescue DAG. The name appends the string `.parse_failed` to the original DAG input file name. Therefore, if the submission of a DAG with

```
$ condor_submit_dag my.dag
```

has a parse failure, the resulting incomplete Rescue DAG will be named `my.dag.parse_failed`.

To further prevent one of these incomplete Rescue DAG files from being used, a line within the file contains the single command *REJECT*. This causes *condor_dagman* to reject the DAG, if used as a DAG input file. This is done because the incomplete Rescue DAG may be a syntactically correct DAG input file. It will be incomplete relative to the original DAG, such that if the incomplete Rescue DAG could be run, it could erroneously be perceived as having successfully executed the desired workflow, when, in fact, it did not.

7.7.2 DAG Recovery

DAG recovery restores the state of a DAG upon resubmission. Recovery is accomplished by reading the `.nodes.log` file that is used to enforce the dependencies of the DAG. The DAG can then continue towards completion.

Recovery is different than a Rescue DAG. Recovery is appropriate when no Rescue DAG has been created. There will be no Rescue DAG if the machine running the *condor_dagman* job crashes, or if the *condor_schedd* daemon crashes, or if the *condor_dagman* job crashes, or if the *condor_dagman* job is placed on hold.

Much of the time, when a not-completed DAG is re-submitted, it will automatically be placed into recovery mode due to the existence and contents of a lock file created as the DAG is first run. In recovery mode, the `.nodes.log` is used to identify nodes that have completed and should not be re-submitted.

DAGMan can be told to work in recovery mode by including the **-DoRecovery** option on the command line, as in the example

```
$ condor_submit_dag diamond.dag -DoRecovery
```

where `diamond.dag` is the name of the DAG input file.

7.8 Node Priorities

7.8.1 Setting Priorities for Nodes

The *PRIORITY* command assigns a priority to a DAG node (and to the HTCondor job(s) associated with the node). The syntax for *PRIORITY* is

```
PRIORITY <JobName | ALL_NODES> PriorityValue
```

The priority value is an integer (which can be negative). A larger numerical priority is better. The default priority is 0.

The node priority affects the order in which nodes that are ready (all of their parent nodes have finished successfully) at the same time will be submitted. The node priority also sets the node job's priority in the queue (that is, its `JobPrio` attribute), which affects the order in which jobs will be run once they are submitted (see *Job Priority* for more information). The node priority only affects the order of job submission within a given DAG; but once jobs are submitted, their `JobPrio` value affects the order in which they will be run relative to all jobs submitted by the same user.

Sub-DAGs can have priorities, just as "regular" nodes can. (The priority of a sub-DAG will affect the priorities of its nodes: see "effective node priorities" below.) Splices cannot be assigned a priority, but individual nodes within a splice can be assigned priorities.

Note that node priority does not override the DAG dependencies. Also note that node priorities are not guarantees of the relative order in which nodes will be run, even among nodes that become ready at the same time - so node priorities should not be used as a substitute for parent/child dependencies. In other words, priorities should be used when it is preferable, but not required, that some jobs run before others. (The order in which jobs are run once they are submitted can be affected by many things other than the job's priority; for example, whether there are machines available in the pool that match the job's requirements.)

PRE scripts can affect the order in which jobs run, so DAGs containing PRE scripts may not submit the nodes in exact priority order, even if doing so would satisfy the DAG dependencies.

Node priority is most relevant if node submission is throttled (via the *-maxjobs* or *-maxidle* command-line arguments or the or configuration variables), or if there are not enough resources in the pool to immediately run all submitted

node jobs. This is often the case for DAGs with large numbers of "sibling" nodes, or DAGs running on heavily-loaded pools.

Example

Adding *PRIORITY* for node C in the diamond-shaped DAG:

```
# File name: diamond.dag

JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
RETRY  C 3
PRIORITY C 1
```

This will cause node C to be submitted (and, mostly likely, run) before node B. Without this priority setting for node C, node B would be submitted first because the "JOB" statement for node B comes earlier in the DAG file than the "JOB" statement for node C.

7.8.2 Effective node priorities

The "effective" priority for a node (the priority controlling the order in which nodes are actually submitted, and which is assigned to JobPrio) is the sum of the explicit priority (specified in the DAG file) and the priority of the DAG itself. DAG priorities also default to 0, so they are most relevant for sub-DAGs (although a top-level DAG can be submitted with a non-zero priority by specifying a **-priority** value on the *condor_submit_dag* command line). **This algorithm for calculating effective priorities is a simplification introduced in version 8.5.7 (a node's effective priority is no longer dependent on the priorities of its parents).**

Here is an example to clarify:

```
# File name: priorities.dag

JOB A A.sub
SUBDAG EXTERNAL B SD.dag
PARENT A CHILD B
PRIORITY A 60
PRIORITY B 100
```

```
# File name: SD.dag

JOB SA SA.sub
JOB SB SB.sub
PARENT SA CHILD SB
PRIORITY SA 10
PRIORITY SB 20
```

In this example (assuming that priorities.dag is submitted with the default priority of 0), the effective priority of node A will be 60, and the effective priority of sub-DAG B will be 100. Therefore, the effective priority of node SA will be 110 and the effective priority of node SB will be 120.

The effective priorities listed above are assigned by DAGMan. There is no way to change the priority in the submit description file for a job, as DAGMan will override any **priority** command placed in a submit description file (unless

the effective node priority is 0; in this case, any priority specified in the submit file will take effect).

7.9 Single Submission of Multiple, Independent DAGs

A single use of *condor_submit_dag* may execute multiple, independent DAGs. Each independent DAG has its own, distinct DAG input file. These DAG input files are command-line arguments to *condor_submit_dag*.

Internally, all of the independent DAGs are combined into a single, larger DAG, with no dependencies between the original independent DAGs. As a result, any generated Rescue DAG file represents all of the original independent DAGs with a single DAG. The file name of this Rescue DAG is based on the DAG input file listed first within the command-line arguments. For example, assume that three independent DAGs are submitted with

```
$ condor_submit_dag A.dag B.dag C.dag
```

The first listed is `A.dag`. The remainder of the specialized file name adds a suffix onto this first DAG input file name, `A.dag`. The suffix is `_multi.rescue<XXX>`, where `<XXX>` is substituted by the 3-digit number of the Rescue DAG created as defined in *The Rescue DAG* section. The first time a Rescue DAG is created for the example, it will have the file name `A.dag_multi.rescue001`.

Other files such as `dagman.out` and the lock file also have names based on this first DAG input file.

The success or failure of the independent DAGs is well defined. When multiple, independent DAGs are submitted with a single command, the success of the composite DAG is defined as the logical AND of the success of each independent DAG. This implies that failure is defined as the logical OR of the failure of any of the independent DAGs.

By default, DAGMan internally renames the nodes to avoid node name collisions. If all node names are unique, the renaming of nodes may be disabled by setting the configuration variable to `False`

7.10 Composing workflows from multiple DAG files

The organization and dependencies of the jobs within a DAG are the keys to its utility. Some workflows are naturally constructed hierarchically, such that a node within a DAG is also a DAG (instead of a "simple" HTCondor job). HTCondor DAGMan handles this situation easily, and allows DAGs to be nested to any depth.

There are two ways that DAGs can be nested within other DAGs:

1. Sub-DAGs
2. Splices.

With Sub-DAGs, each DAG has its own *condor_dagman* job, which then becomes a node job within the higher-level DAG. With splices, on the other hand, the nodes of the spliced DAG are directly incorporated into the higher-level DAG. Therefore, splices do not result in additional *condor_dagman* instances.

A weakness in scalability exists when submitting external Sub-DAGs, because each executing independent DAG requires its own instance of *condor_dagman* to be running. The outer DAG has an instance of *condor_dagman*, and each named SUBDAG has an instance of *condor_dagman* while it is in the HTCondor queue. The scaling issue presents itself when a workflow contains hundreds or thousands of Sub-DAGs that are queued at the same time. (In this case, the resources (especially memory) consumed by the multiple *condor_dagman* instances can be a problem.) Further, there may be many Rescue DAGs created if a problem occurs. (Note that the scaling issue depends only on how many Sub-DAGs are queued at any given time, not the total number of Sub-DAGs in a given workflow; division of a large workflow into sequential Sub-DAGs can actually enhance scalability.) To alleviate these concerns, the DAGMan language introduces the concept of graph splicing.

Because splices are simpler in some ways than sub-DAGs, they are generally preferred unless certain features are needed that are only available with Sub-DAGs. This document: https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=SubDagsVsSplices explains the pros and cons of splices and external sub-DAGs, and should help users decide which alternative is better for their application.

Note that Sub-DAGs and splices can be combined in a single workflow, and can be nested to any depth (but be sure to avoid recursion, which will cause problems!).

7.10.1 A DAG Within a DAG Is a SUBDAG

As stated above, the SUBDAG EXTERNAL command causes the specified DAG file to be run by a separate instance of *condor_dagman*, with the *condor_dagman* job becoming a node job within the higher-level DAG.

The syntax for the SUBDAG command is

```
SUBDAG EXTERNAL JobName DagFileName [DIR directory] [NOOP] [DONE]
```

The optional specifications of **DIR**, **NOOP**, and **DONE**, if used, must appear in this order within the entry. **NOOP** and **DONE** for **SUBDAG** nodes have the same effect that they do for **JOB** nodes.

A **SUBDAG** node is essentially the same as any other node, except that the DAG input file for the inner DAG is specified, instead of the HTCondor submit file. The keyword **EXTERNAL** means that the SUBDAG is run within its own instance of *condor_dagman*.

Since more than one DAG is being discussed, here is terminology introduced to clarify which DAG is which. Reuse the example diamond-shaped DAG as given in previous examples. Assume that node B of this diamond-shaped DAG will itself be a DAG. The DAG of node B is called a SUBDAG, inner DAG, or lower-level DAG. The diamond-shaped DAG is called the outer or top-level DAG.

Work on the inner DAG first. Here is a very simple linear DAG input file used as an example of the inner DAG.

```
# File name: inner.dag

JOB  X  X.sub
JOB  Y  Y.sub
JOB  Z  Z.sub
PARENT X CHILD Y
PARENT Y CHILD Z
```

The HTCondor submit description file, used by *condor_dagman*, corresponding to `inner.dag` will be named `inner.dag.condor.sub`. The DAGMan submit description file is always named `<DAG file name>.condor.sub`. Each DAG or SUBDAG results in the submission of *condor_dagman* as an HTCondor job, and *condor_submit_dag* creates this submit description file.

The preferred specification of the DAG input file for the outer DAG is

```
# File name: diamond.dag

JOB  A  A.submit
SUBDAG EXTERNAL  B  inner.dag
JOB  C  C.submit
JOB  D  D.submit
PARENT A CHILD B C
PARENT B C CHILD D
```

Within the outer DAG's input file, the **SUBDAG** command specifies a special case of a **JOB** node, where the job is itself a DAG.

One of the benefits of using the SUBDAG feature is that portions of the overall workflow can be constructed and modified during the execution of the DAG (a SUBDAG file doesn't have to exist until just before it is submitted). A drawback can be that each SUBDAG causes its own distinct job submission of *condor_dagman*, leading to a larger number of jobs, together with their potential need of carefully constructed policy configuration to throttle node submission or execution (because each SUBDAG has its own throttles).

Here are details that affect SUBDAGs:

- Nested DAG Submit Description File Generation

  There are three ways to generate the <DAG file name>.condor.sub file of a SUBDAG:

  - **Lazily** (the default in HTCondor version 7.5.2 and later versions)
  - **Eagerly** (the default in HTCondor versions 7.4.1 through 7.5.1)
  - **Manually** (the only way prior to version HTCondor version 7.4.1)

  When the <DAG file name>.condor.sub file is generated **lazily**, this file is generated immediately before the SUBDAG job is submitted. Generation is accomplished by running

```
$ condor_submit_dag -no_submit
```

  on the DAG input file specified in the **SUBDAG** entry. This is the default behavior. There are advantages to this lazy mode of submit description file creation for the SUBDAG:

  - The DAG input file for a SUBDAG does not have to exist until the SUBDAG is ready to run, so this file can be dynamically created by earlier parts of the outer DAG or by the PRE script of the node containing the SUBDAG.
  - It is now possible to have SUBDAGs within splices. That is not possible with eager submit description file creation, because *condor_submit_dag* does not understand splices.

  The main disadvantage of lazy submit file generation is that a syntax error in the DAG input file of a SUBDAG will not be discovered until the outer DAG tries to run the inner DAG.

  When <DAG file name>.condor.sub files are generated **eagerly**, *condor_submit_dag* runs itself recursively (with the *-no_submit* option) on each SUBDAG, so all of the <DAG file name>.condor.sub files are generated before the top-level DAG is actually submitted. To generate the <DAG filename>.condor.sub files eagerly, pass the *-do_recurse* flag to *condor_submit_dag*; also set the configuration variable to `False`, so that *condor_dagman* does not re-run *condor_submit_dag* at run time thereby regenerating the submit description files.

  To generate the .condor.sub files **manually**, run

```
$ condor_submit_dag -no_submit
```

  on each lower-level DAG file, before running *condor_submit_dag* on the top-level DAG file; also set the configuration variable to `False`, so that *condor_dagman* does not re-run *condor_submit_dag* at run time. The main reason for generating the <DAG file name>.condor.sub files manually is to set options for the lower-level DAG that one would not otherwise be able to set An example of this is the *-insert_sub_file* option. For instance, using the given example do the following to manually generate HTCondor submit description files:

```
$ condor_submit_dag -no_submit -insert_sub_file fragment.sub inner.dag
$ condor_submit_dag diamond.dag
```

Note that most *condor_submit_dag* command-line flags have corresponding configuration variables, so we encourage the use of per-DAG configuration files, especially in the case of nested DAGs. This is the easiest way to set different options for different DAGs in an overall workflow.

It is possible to combine more than one method of generating the `<DAG file name>.condor.sub` files. For example, one might pass the *-do_recurse* flag to *condor_submit_dag*, but leave the `DAGMAN_GENERATE_SUBDAG_SUBMITS` configuration variable set to the default of `True`. Doing this would provide the benefit of an immediate error message at submit time, if there is a syntax error in one of the inner DAG input files, but the lower-level `<DAG file name>.condor.sub` files would still be regenerated before each nested DAG is submitted.

The values of the following command-line flags are passed from the top-level *condor_submit_dag* instance to any lower-level *condor_submit_dag* instances. This occurs whether the lower-level submit description files are generated lazily or eagerly:

- **-verbose**
- **-force**
- **-notification**
- **-allowlogerror**
- **-dagman**
- **-usedagdir**
- **-outfile_dir**
- **-oldrescue**
- **-autorescue**
- **-dorescuefrom**
- **-allowversionmismatch**
- **-no_recurse/do_recurse**
- **-update_submit**
- **-import_env**
- **-include_env**
- **-insert_env**
- **-suppress_notification**
- **-priority**
- **-dont_use_default_node_log**

The values of the following command-line flags are preserved in any already-existing lower-level DAG submit description files:

- **-maxjobs**
- **-maxidle**
- **-maxpre**
- **-maxpost**
- **-debug**

Other command-line arguments are set to their defaults in any lower-level invocations of *condor_submit_dag*.

The **-force** option will cause existing DAG submit description files to be overwritten without preserving any existing values.

- Submission of the outer DAG

  The outer DAG is submitted as before, with the command

```
$ condor_submit_dag diamond.dag
```

- Interaction with Rescue DAGs

  The use of new-style Rescue DAGs is now the default. With new-style rescue DAGs, the appropriate rescue DAG(s) will be run automatically if there is a failure somewhere in the workflow. For example (given the DAGs in the example at the beginning of the SUBDAG section), if one of the nodes in `inner.dag` fails, this will produce a Rescue DAG for `inner.dag` (named `inner.dag.rescue.001`). Then, since `inner.dag` failed, node B of `diamond.dag` will fail, producing a Rescue DAG for `diamond.dag` (named `diamond.dag.rescue.001`, etc.). If the command

```
$ condor_submit_dag diamond.dag
```

  is re-run, the most recent outer Rescue DAG will be run, and this will re-run the inner DAG, which will in turn run the most recent inner Rescue DAG.

- File Paths

  Remember that, unless the DIR keyword is used in the outer DAG, the inner DAG utilizes the current working directory when the outer DAG is submitted. Therefore, all paths utilized by the inner DAG file must be specified accordingly.

7.10.2 DAG Splicing

As stated above, the SPLICE command causes the nodes of the spliced DAG to be directly incorporated into the higher-level DAG (the DAG containing the SPLICE command).

The syntax for the *SPLICE* command is

```
SPLICE SpliceName DagFileName [DIR directory]
```

A splice is a named instance of a subgraph which is specified in a separate DAG file. The splice is treated as an entity for dependency specification in the including DAG. (Conceptually, a splice is treated as a node within the DAG containing the SPLICE command, although there are some limitations, which are discussed below. This means, for example, that splices can have parents and children.) A splice can also be incorporated into an including DAG without any dependencies; it is then considered a disjoint DAG within the including DAG.

The same DAG file can be reused as differently named splices, each one incorporating a copy of the dependency graph (and nodes therein) into the including DAG.

The nodes within a splice are scoped according to a hierarchy of names associated with the splices, as the splices are parsed from the top level DAG file. The scoping character to describe the inclusion hierarchy of nodes into the top level dag is '+'. (In other words, if a splice named "SpliceX" contains a node named "NodeY", the full node name once the DAGs are parsed is "SpliceX+NodeY". This character is chosen due to a restriction in the allowable characters which may be in a file name across the variety of platforms that HTCondor supports. In any DAG input file, all splices must have unique names, but the same splice name may be reused in different DAG input files.

HTCondor does not detect nor support splices that form a cycle within the DAG. A DAGMan job that causes a cyclic inclusion of splices will eventually exhaust available memory and crash.

The *SPLICE* command in a DAG input file creates a named instance of a DAG as specified in another file as an entity which may have *PARENT* and *CHILD* dependencies associated with other splice names or node names in the including DAG file.

The following series of examples illustrate potential uses of splicing. To simplify the examples, presume that each and every job uses the same, simple HTCondor submit description file:

```
# BEGIN SUBMIT FILE simple-job.sub
executable   = /bin/echo
arguments    = OK
universe     = vanilla
output       = $(jobname).out
error        = $(jobname).err
log          = submit.log
notification = NEVER

request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
# END SUBMIT FILE simple-job.sub
```

Simple SPLICE Example

This first simple example splices a diamond-shaped DAG in between the two nodes of a top level DAG. Here is the DAG input file for the diamond-shaped DAG:

```
# BEGIN DAG FILE diamond.dag
JOB A simple-job.sub
VARS A jobname="$(JOB)"

JOB B simple-job.sub
VARS B jobname="$(JOB)"

JOB C simple-job.sub
VARS C jobname="$(JOB)"

JOB D simple-job.sub
VARS D jobname="$(JOB)"

PARENT A CHILD B C
PARENT B C CHILD D
# END DAG FILE diamond.dag
```

The top level DAG incorporates the diamond-shaped splice:

```
# BEGIN DAG FILE toplevel.dag
JOB X simple-job.sub
VARS X jobname="$(JOB)"

```

(continues on next page)

(continued from previous page)

```
JOB Y simple-job.sub
VARS Y jobname="$(JOB)"

# This is an instance of diamond.dag, given the symbolic name DIAMOND
SPLICE DIAMOND diamond.dag

# Set up a relationship between the nodes in this dag and the splice

PARENT X CHILD DIAMOND
PARENT DIAMOND CHILD Y

# END DAG FILE toplevel.dag
```

The following example illustrates the resulting top level DAG and the dependencies produced. Notice the naming of nodes scoped with the splice name. This hierarchy of splice names assures unique names associated with all nodes.

Fig. 1: The diamond-shaped DAG spliced between two nodes.

SPLICING one DAG Twice Example

The next example illustrates the starting point for a more complex example. The DAG input file `X.dag` describes this X-shaped DAG. The completed example displays more of the spatial constructs provided by splices. Pay particular attention to the notion that each named splice creates a new graph, even when the same DAG input file is specified.

```
# BEGIN DAG FILE X.dag

JOB A simple-job.sub
VARS A jobname="$(JOB)"

JOB B simple-job.sub
VARS B jobname="$(JOB)"

JOB C simple-job.sub
VARS C jobname="$(JOB)"

JOB D simple-job.sub
VARS D jobname="$(JOB)"

JOB E simple-job.sub
VARS E jobname="$(JOB)"

JOB F simple-job.sub
VARS F jobname="$(JOB)"

JOB G simple-job.sub
VARS G jobname="$(JOB)"

# Make an X-shaped dependency graph
PARENT A B C CHILD D
PARENT D CHILD E F G
```

(continues on next page)

(continued from previous page)

```
# END DAG FILE X.dag
```

Fig. 2: The X-shaped DAG.

File `s1.dag` continues the example, presenting the DAG input file that incorporates two separate splices of the X-shaped DAG. The next description illustrates the resulting DAG.

```
# BEGIN DAG FILE s1.dag

JOB A simple-job.sub
VARS A jobname="$(JOB)"

JOB B simple-job.sub
VARS B jobname="$(JOB)"

# name two individual splices of the X-shaped DAG
SPLICE X1 X.dag
SPLICE X2 X.dag

# Define dependencies
# A must complete before the initial nodes in X1 can start
PARENT A CHILD X1
# All final nodes in X1 must finish before
# the initial nodes in X2 can begin
PARENT X1 CHILD X2
# All final nodes in X2 must finish before B may begin.
PARENT X2 CHILD B

# END DAG FILE s1.dag
```

Fig. 3: The DAG described by s1.dag.

Disjointed SPLICE Example

The top level DAG in the hierarchy of this complex example is described by the DAG input file `toplevel.dag`, which illustrates the final DAG. Notice that the DAG has two disjoint graphs in it as a result of splice S3 not having any dependencies associated with it in this top level DAG.

```
# BEGIN DAG FILE toplevel.dag

JOB A simple-job.sub
VARS A jobname="$(JOB)"

JOB B simple-job.sub
VARS B jobname="$(JOB)"

JOB C simple-job.sub
VARS C jobname="$(JOB)"
```

(continues on next page)

(continued from previous page)

```
JOB D simple-job.sub
VARS D jobname="$(JOB)"

# a diamond-shaped DAG
PARENT A CHILD B C
PARENT B C CHILD D

# This splice of the X-shaped DAG can only run after
# the diamond dag finishes
SPLICE S2 X.dag
PARENT D CHILD S2

# Since there are no dependencies for S3,
# the following splice is disjoint
SPLICE S3 s1.dag

# END DAG FILE toplevel.dag
```

Fig. 4: The complex splice example DAG.

Splice DIR option

The *DIR* option specifies a working directory for a splice, from which the splice will be parsed and the jobs within the splice submitted. The directory associated with the splice's *DIR* specification will be propagated as a prefix to all nodes in the splice and any included splices. If a node already has a *DIR* specification, then the splice's *DIR* specification will be a prefix to the node's, separated by a directory separator character. Jobs in included splices with an absolute path for their *DIR* specification will have their *DIR* specification untouched. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*.

A "full" rescue DAG generated by a DAG run with the *-usedagdir* argument will contain DIR specifications, so such a rescue DAG must be run without the *-usedagdir* argument. (Note that "full" rescue DAGs are no longer the default.)

Splice Limitations

Limitation: splice DAGS do not produce rescue DAGs

Because the nodes of a splice are directly incorporated into the DAG containing the SPLICE command, splices do not generate their own rescue DAGs, unlike SUBDAG EXTERNALs. However, all progress for nodes in the splice DAG will be written in the parent DAGs rescue DAG file.

Limitation: splice DAGs must exist at submit time

Unlike the DAG files referenced in a SUBDAG EXTERNAL command, DAG files referenced in a SPLICE command must exist when the DAG containing the SPLICE command is submitted. (Note that, if a SPLICE is contained within a sub-DAG, the splice DAG must exist at the time that the sub-DAG is submitted, not when the top-most DAG is submitted, so the splice DAG can be created by a part of the workflow that runs before the relevant sub-DAG.)

Limitation: Splices and PRE or POST Scripts

A PRE or POST script may not be specified for a splice (however, nodes within a spliced DAG can have PRE and POST scripts). The reason for this is that, when the DAG is parsed, the splices are also parsed and the splice nodes are directly

incorporated into the DAG containing the SPLICE command. Therefore, once parsing is complete, there are no actual nodes corresponding to the splice itself to which to "attach" the PRE or POST scripts.

To achieve the desired effect of having a PRE script associated with a splice, introduce a new NOOP node into the DAG with the splice as a dependency. Attach the PRE script to the NOOP node.

```
# BEGIN DAG FILE example1.dag

# Names a node with no associated node job, a NOOP node
# Note that the file noop.submit does not need to exist
JOB OnlyPreNode noop.sub NOOP

# Attach a PRE script to the NOOP node
SCRIPT PRE OnlyPreNode prescript.sh

# Define the splice
SPLICE TheSplice thenode.dag

# Define the dependency
PARENT OnlyPreNode CHILD TheSplice

# END DAG FILE example1.dag
```

The same technique is used to achieve the effect of having a POST script associated with a splice. Introduce a new NOOP node into the DAG as a child of the splice, and attach the POST script to the NOOP node.

```
# BEGIN DAG FILE example2.dag

# Names a node with no associated node job, a NOOP node
# Note that the file noop.submit does not need to exist.
JOB OnlyPostNode noop.sub NOOP

# Attach a POST script to the NOOP node
SCRIPT POST OnlyPostNode postscript.sh

# Define the splice
SPLICE TheSplice thenode.dag

# Define the dependency
PARENT TheSplice CHILD OnlyPostNode

# END DAG FILE example2.dag
```

Limitation: Splices and the RETRY of a Node, use of VARS, or use of PRIORITY

A RETRY, VARS or PRIORITY command cannot be specified for a SPLICE; however, individual nodes within a spliced DAG can have a RETRY, VARS or PRIORITY specified.

Here is an example showing a DAG that will not be parsed successfully:

```
# top level DAG input file
JOB    A a.sub
SPLICE B b.dag
PARENT A CHILD B
```

(continues on next page)

(continued from previous page)

```
# cannot work, as B is not a node in the DAG once
# splice B is incorporated
RETRY B 3
VARS  B dataset="10"
PRIORITY B 20
```

The following example will work:

```
# top level DAG input file
JOB    A a.sub
SPLICE B b.dag
PARENT A  CHILD B

# file: b.dag
JOB   X x.sub
RETRY X 3
VARS  X dataset="10"
PRIORITY X 20
```

When RETRY is desired on an entire subgraph of a workflow, sub-DAGs (see above) must be used instead of splices.

Here is the same example, now defining job B as a SUBDAG, and effecting RETRY on that SUBDAG.

```
# top level DAG input file
JOB    A a.sub
SUBDAG EXTERNAL B b.dag
PARENT A  CHILD B

RETRY B 3
```

Limitation: The Interaction of Categories and MAXJOBS with Splices

Categories normally refer only to nodes within a given splice. All of the assignments of nodes to a category, and the setting of the category throttle, should be done within a single DAG file. However, it is now possible to have categories include nodes from within more than one splice. To do this, the category name is prefixed with the + (plus) character. This tells DAGMan that the category is a cross-splice category. Towards deeper understanding, what this really does is prevent renaming of the category when the splice is incorporated into the upper-level DAG. The MAXJOBS specification for the category can appear in either the upper-level DAG file or one of the splice DAG files. It probably makes the most sense to put it in the upper-level DAG file.

Here is an example which applies a single limitation on submitted jobs, identifying the category with `+init`.

```
# relevant portion of file name: upper.dag

SPLICE A splice1.dag
SPLICE B splice2.dag

MAXJOBS +init 2
```

```
# relevant portion of file name: splice1.dag

JOB C C.sub
CATEGORY C +init
```

(continues on next page)

(continued from previous page)

```
JOB D D.sub
CATEGORY D +init
```

```
# relevant portion of file name: splice2.dag

JOB X X.sub
CATEGORY X +init
JOB Y Y.sub
CATEGORY Y +init
```

For both global and non-global category throttles, settings at a higher level in the DAG override settings at a lower level. In this example:

```
# relevant portion of file name: upper.dag

SPLICE A lower.dag

MAXJOBS A+catX 10
MAXJOBS +catY 2


# relevant portion of file name: lower.dag

MAXJOBS catX 5
MAXJOBS +catY 1
```

the resulting throttle settings are 2 for the `+catY` category and 10 for the `A+catX` category in splice. Note that non-global category names are prefixed with their splice name(s), so to refer to a non-global category at a higher level, the splice name must be included.

7.11 DAGMan Throttling

Submit machines with limited resources are supported by command line options that place limits on the submission and handling of HTCondor jobs and PRE and POST scripts. Presented here are descriptions of the command line options to *condor_submit_dag*. These same limits can be set in configuration. Each limit is applied within a single DAG.

7.11.1 Throttling at DAG Submission

- **Total nodes/clusters:** The **-maxjobs** option specifies the maximum number of clusters that *condor_dagman* can submit at one time. Since each node corresponds to a single cluster, this limit restricts the number of nodes that can be submitted (in the HTCondor queue) at a time. It is commonly used when there is a limited amount of input file staging capacity. As a specific example, consider a case where each node represents a single HTCondor proc that requires 4 MB of input files, and the proc will run in a directory with a volume of 100 MB of free space. Using the argument **-maxjobs 25** guarantees that a maximum of 25 clusters, using a maximum of 100 MB of space, will be submitted to HTCondor at one time. (See the *condor_submit_dag* manual page) for more information. Also see the equivalent configuration option.
- **Idle procs:** The number of idle procs within a given DAG can be limited with the optional command line argument **-maxidle**. *condor_dagman* will not submit any more node jobs until the number of idle procs in the

DAG goes below this specified value, even if there are ready nodes in the DAG. This allows *condor_dagman* to submit jobs in a way that adapts to the load on the HTCondor pool at any given time. If the pool is lightly loaded, *condor_dagman* will end up submitting more jobs; if the pool is heavily loaded, *condor_dagman* will submit fewer jobs. (See the *condor_submit_dag* manual page for more information.) Also see the equivalent configuration option.

- **PRE/POST scripts:** Since PRE and POST scripts run on the submit machine, it may be desirable to limit the number of PRE or POST scripts running at one time. The optional **-maxpre** command line argument limits the number of PRE scripts that may be running at one time, and the optional **-maxpost** command line argument limits the number of POST scripts that may be running at one time. (See the *condor_submit_dag* manual page for more information.) Also see the equivalent and configuration options.

7.11.2 Throttling Nodes by Category

In order to limit the number of submitted job clusters within a DAG, the nodes may be placed into categories by assignment of a name. Then, a maximum number of submitted clusters may be specified for each category.

The *CATEGORY* command assigns a category name to a DAG node. The syntax for *CATEGORY* is

```
CATEGORY <JobName | ALL_NODES> CategoryName
```

Category names cannot contain white space.

The *MAXJOBS* command limits the number of submitted job clusters on a per category basis. The syntax for *MAXJOBS* is

```
MAXJOBS CategoryName MaxJobsValue
```

If the number of submitted job clusters for a given category reaches the limit, no further job clusters in that category will be submitted until other job clusters within the category terminate. If MAXJOBS is not set for a defined category, then there is no limit placed on the number of submissions within that category.

Note that a single invocation of *condor_submit* results in one job cluster. The number of HTCondor jobs within a cluster may be greater than 1.

The configuration variable and the *condor_submit_dag -maxjobs* command-line option are still enforced if these *CATEGORY* and *MAXJOBS* throttles are used.

Please see *Splice Limitations* for a description of the interaction between categories and DAG splices.

7.12 Optimization of Submission Time

condor_dagman works by watching log files for events, such as submission, termination, and going on hold. When a new job is ready to be run, it is submitted to the *condor_schedd*, which needs to acquire a computing resource. Acquisition requires the *condor_schedd* to contact the central manager and get a claim on a machine, and this claim cycle can take many minutes.

Configuration variable avoids the wait for a negotiation cycle. When set to a non zero value, the *condor_schedd* keeps a claim idle, such that the *condor_startd* delays in shifting from the Claimed to the Preempting state (see *Policy Configuration for Execution Points and for Access Points*). Thus, if another job appears that is suitable for the claimed resource, then the *condor_schedd* will submit the job directly to the *condor_startd*, avoiding the wait and overhead of

a negotiation cycle. This results in a speed up of job completion, especially for linear DAGs in pools that have lengthy negotiation cycle times.

By default, `DAGMAN_HOLD_CLAIM_TIME` is 20, causing a claim to remain idle for 20 seconds, during which time a new job can be submitted directly to the already-claimed *condor_startd*. A value of 0 means that claims are not held idle for a running DAG. If a DAG node has no children, the value of `DAGMAN_HOLD_CLAIM_TIME` will be ignored; the `KeepClaimIdle` attribute will not be defined in the job ClassAd of the node job, unless the job requests it using the submit command **keep_claim_idle** .

7.13 Managing Large Numbers of Jobs with DAGMan

Using DAGMan is recommended when submitting large numbers of jobs. The recommendation holds whether the jobs are represented by a DAG due to dependencies, or all the jobs are independent of each other, such as they might be in a parameter sweep. DAGMan offers:

- **Throttling**
  Throttling limits the number of submitted jobs at any point in time.
- **Retry of jobs that fail**
  This is a useful tool when an intermittent error may cause a job to fail or may cause a job to fail to run to completion when attempted at one point in time, but not at another point in time. The conditions under which retry occurs are user-defined. In addition, the administrative support that facilitates the rerunning of only those jobs that fail is automatically generated.
- **Scripts associated with node jobs**
  PRE and POST scripts run on the submit host before and/or after the execution of specified node jobs.

Each of these capabilities is described in detail within this manual section about DAGMan. To make effective use of DAGMan, there is no way around reading the appropriate subsections.

To run DAGMan with large numbers of independent jobs, there are generally two ways of organizing and specifying the files that control the jobs. Both ways presume that programs or scripts will generate needed files, because the file contents are either large and repetitive, or because there are a large number of similar files to be generated representing the large numbers of jobs. The two file types needed are the DAG input file and the submit description file(s) for the HTCondor jobs represented. Each of the two ways is presented separately:

- **A unique submit description file for each of the many jobs.**
  A single DAG input file lists each of the jobs and specifies a distinct submit description file for each job. The DAG input file is simple to generate, as it chooses an identifier for each job and names the submit description file. For example, the simplest DAG input file for a set of 1000 independent jobs, as might be part of a parameter sweep, appears as

```
# file sweep.dag
JOB job0 job0.sub
JOB job1 job1.sub
JOB job2 job2.sub
...
JOB job999 job999.sub
```

  There are 1000 submit description files, with a unique one for each of the job<N> jobs. Assuming that all files associated with this set of jobs are in the same directory, and that files continue the same naming and numbering scheme, the submit description file for `job6.sub` might appear as

```
# file job6.sub
universe = vanilla
executable = /path/to/executable
log = job6.log
input = job6.in
output = job6.out
arguments = "-file job6.out"
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

Submission of the entire set of jobs uses the command line:

```
$ condor_submit_dag sweep.dag
```

A benefit to having unique submit description files for each of the jobs is that they are available if one of the jobs needs to be submitted individually. A drawback to having unique submit description files for each of the jobs is that there are lots of submit description files.

- **Single submit description file.**
  A single HTCondor submit description file might be used for all the many jobs of the parameter sweep. To distinguish the jobs and their associated distinct input and output files, the DAG input file assigns a unique identifier with the *VARS* command.

```
# file sweep.dag
JOB job0 common.sub
VARS job0 runnumber="0"
JOB job1 common.sub
VARS job1 runnumber="1"
JOB job2 common.sub
VARS job2 runnumber="2"
...
JOB job999 common.sub
VARS job999 runnumber="999"
```

The single submit description file for all these jobs utilizes the `runnumber` variable value in its identification of the job's files. This submit description file might appear as

```
# file common.sub
universe = vanilla
executable = /path/to/executable
log = wholeDAG.log
input = job$(runnumber).in
output = job$(runnumber).out
arguments = "-$(runnumber)"
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

The job with `runnumber="8"` expects to find its input file `job8.in` in the single, common directory, and it sends its output to `job8.out`. The single log for all job events of the entire DAG is `wholeDAG.log`. Using

one file for the entire DAG meets the limitation that no macro substitution may be specified for the job log file, and it is likely more efficient as well. This node's executable is invoked with

```
/path/to/executable -8
```

These examples work well with respect to file naming and file location when there are less than several thousand jobs submitted as part of a DAG. The large numbers of files per directory becomes an issue when there are greater than several thousand jobs submitted as part of a DAG. In this case, consider a more hierarchical structure for the files instead of a single directory. Introduce a separate directory for each run. For example, if there were 10,000 jobs, there would be 10,000 directories, one for each of these jobs. The directories are presumed to be generated and populated by programs or scripts that, like the previous examples, utilize a run number. Each of these directories named utilizing the run number will be used for the input, output, and log files for one of the many jobs.

As an example, for this set of 10,000 jobs and directories, assume that there is a run number of 600. The directory will be named `dir600`, and it will hold the 3 files called `in`, `out`, and `log`, representing the input, output, and HTCondor job log files associated with run number 600.

The DAG input file sets a variable representing the run number, as in the previous example:

```
# file biggersweep.dag
JOB job0 bigger.sub
VARS job0 runnumber="0"
JOB job1 bigger.sub
VARS job1 runnumber="1"
JOB job2 bigger.sub
VARS job2 runnumber="2"
...
JOB job9999 bigger.sub
VARS job9999 runnumber="9999"
```

A single HTCondor submit description file may be written. It resides in the same directory as the DAG input file.

```
# file bigger.sub
universe = vanilla
executable = /path/to/executable
log = log
input = in
output = out
arguments = "-$(runnumber)"
initialdir = dir$(runnumber)
request_cpus   = 1
request_memory = 1024M
request_disk   = 10240K

queue
```

One item to care about with this set up is the underlying file system for the pool. The transfer of files (or not) when using **initialdir** differs based upon the job **universe** and whether or not there is a shared file system. See the *condor_submit* manual page for the details on the submit command.

Submission of this set of jobs is no different than the previous examples. With the current working directory the same as the one containing the submit description file, the DAG input file, and the subdirectories:

```
$ condor_submit_dag biggersweep.dag
```

7.14 Custom Variables for Nodes

Jobs may be set up in a way that require a submit time `key=value` macros of information to be used in the jobs submit description dictating various behaviors for how the job can run. This allows a single submit description file for a job to be versatile for many different job runs. However, for a normal job submission (one not automated) the user must pass this extra information at job submit time. To mimic this behavior of passing information at job submit time within a DAGMan workflow, the *VARS* command can be utilized.

7.14.1 Variable Values Associated with Nodes

Macros defined for DAG nodes can be used within the submit description file of the node job. The *VARS* command provides a method for defining a macro. Macros are defined on a per-node basis, using the syntax

```
VARS <JobName | ALL_NODES> [PREPEND | APPEND] macroname="string" [macroname2="string2" ..
↪. ]
```

The macro may be used within the submit description file of the relevant node. A *macroname* may contain alphanumeric characters (a-z, A-Z, and 0-9) and the underscore character. The space character delimits macros, such that there may be more than one macro defined on a single line. Multiple lines defining macros for the same node are permitted.

Correct syntax requires that the *string* must be enclosed in double quotes. To use a double quote mark within a *string*, escape the double quote mark with the backslash character (\). To add the backslash character itself, use two backslashes (\\).

A restriction is that the *macroname* itself cannot begin with the string `queue`, in any combination of upper or lower case letters.

Examples

If the DAG input file contains

```
# File name: diamond.dag

JOB  A  A.submit
JOB  B  B.submit
JOB  C  C.submit
JOB  D  D.submit
VARS A state="Wisconsin"
PARENT A CHILD B C
PARENT B C CHILD D
```

then the submit description file `A.submit` may use the macro state. Consider this submit description file `A.submit`:

```
# file name: A.submit
executable = A.exe
log        = A.log
arguments  = "$(state)"
queue
```

The macro value expands to become a command-line argument in the invocation of the job. The job is invoked with

```
A.exe Wisconsin
```

The use of macros may allow a reduction in the number of distinct submit description files. A separate example shows this intended use of *VARS*. In the case where the submit description file for each node varies only in file naming, macros reduce the number of submit description files to one.

This example references a single submit description file for each of the nodes in the DAG input file, and it uses the *VARS* entry to name files used by each job.

The relevant portion of the DAG input file appears as

```
JOB A theonefile.sub
JOB B theonefile.sub
JOB C theonefile.sub

VARS A filename="A"
VARS B filename="B"
VARS C filename="C"
```

The submit description file appears as

```
# submit description file called:  theonefile.sub
executable   = progX
output       = $(filename)
error        = error.$(filename)
log          = $(filename).log
queue
```

For a DAG such as this one, but with thousands of nodes, the ability to write and maintain a single submit description file together with a single, yet more complex, DAG input file is worthwhile.

7.14.2 Prepend or Append Variables to Node

After *JobName* the word *PREPEND* or *APPEND* can be added to specify how a variable is passed to a node at job submission time. *APPEND* will add the variable after the submit description file is read. Resulting in the passed variable being added as a macro or overwriting any already existing variable values. *PREPEND* will add the variable before the submit description file is read. This allows the variable to be used in submit description file conditionals.

The relevant portion of the DAG input file appears as

```
JOB A theotherfile.sub

VARS A PREPEND var1="A"
VARS A APPEND  var2="B"
```

The submit description file appears as

```
# submit description file called:   theotherfile.sub
executable   = progX

if defined var1
     # This will occur due to PREPEND
     Arguments = "$(var1) was prepended"
else
     # This will occur due to APPEND
     Arguments = "No variables prepended"
endif
```

(continues on next page)

(continued from previous page)

```
var2 = "C"

output        = results-$(var2).out
error         = error.txt
log           = job.log
queue
```

For a DAG such as this one, `Arguments` will become "A was prepended" and the output file will be named `results-B.out`. If instead var1 used *APPEND* and var2 used *PREPEND* then `Arguments` will become "No variables prepended" and the output file will be named `results-C.out`.

If neither *PREPEND* nor *APPEND* is used in the *VARS* line then the variable will either be prepended or appended based on the configuration variable .

7.14.3 Multiple macroname definitions

If a macro name for a specific node in a DAG is defined more than once, as it would be with the partial file contents

```
JOB job1 job1.submit
VARS job1 a="foo"
VARS job1 a="bar"
```

a warning is written to the log, of the format

```
Warning: VAR <macroname> is already defined in job <JobName>
Discovered at file "<DAG input file name>", line <line number>
```

The behavior of DAGMan is such that all definitions for the macro exist, but only the last one defined is used as the variable's value. Using this example, if the `job1.submit` submit description file contains

```
arguments = "$(a)"
```

then the argument will be `bar`.

7.14.4 Special characters within VARS string definitions

The value defined for a macro may contain spaces and tabs. It is also possible to have double quote marks and backslashes within a value. In order to have spaces or tabs within a value specified for a command line argument, use the New Syntax format for the **arguments** submit command, as described in *condor_submit*. Escapes for double quote marks depend on whether the New Syntax or Old Syntax format is used for the **arguments** submit command. Note that in both syntaxes, double quote marks require two levels of escaping: one level is for the parsing of the DAG input file, and the other level is for passing the resulting value through *condor_submit*.

As of HTCondor version 8.3.7, single quotes are permitted within the value specification. For the specification of command line **arguments**, single quotes can be used in three ways:

- in Old Syntax, within a macro's value specification
- in New Syntax, within a macro's value specification
- in New Syntax only, to delimit an argument containing white space

There are examples of all three cases below. In New Syntax, to pass a single quote as part of an argument, escape it with another single quote for *condor_submit* parsing as in the example's NodeA `fourth` macro.

As an example that shows uses of all special characters, here are only the relevant parts of a DAG input file. Note that the NodeA value for the macro `second` contains a tab.

```
VARS NodeA first="Alberto Contador"
VARS NodeA second="\"\"Andy Schleck\"\""
VARS NodeA third="Lance\\ Armstrong"
VARS NodeA fourth="Vincenzo ''The Shark'' Nibali"
VARS NodeA misc="!@#$%^&*()_-=+=[]{}?/"

VARS NodeB first="Lance_Armstrong"
VARS NodeB second="\\\"Andreas_Kloden\\\""
VARS NodeB third="Ivan_Basso"
VARS NodeB fourth="Bernard_'The_Badger'_Hinault"
VARS NodeB misc="!@#$%^&*()_-=+=[]{}?/"

VARS NodeC args="'Nairo Quintana' 'Chris Froome'"
```

Consider an example in which the submit description file for NodeA uses the New Syntax for the **arguments** command:

```
arguments = "'$(first)' '$(second)' '$(third)' '($fourth)' '$(misc)'"
```

The single quotes around each variable reference are only necessary if the variable value may contain spaces or tabs. The resulting values passed to the NodeA executable are:

```
Alberto Contador
"Andy Schleck"
Lance\ Armstrong
Vincenzo 'The Shark' Nibali
!@#$%^&*()_-=+=[]{}?/
```

Consider an example in which the submit description file for NodeB uses the Old Syntax for the **arguments** command:

```
arguments = $(first) $(second) $(third) $(fourth) $(misc)
```

The resulting values passed to the NodeB executable are:

```
Lance_Armstrong
"Andreas_Kloden"
Ivan_Basso
Bernard_'The_Badger'_Hinault
!@#$%^&*()_-=+=[]{}?/
```

Consider an example in which the submit description file for NodeC uses the New Syntax for the **arguments** command:

```
arguments = "$(args)"
```

The resulting values passed to the NodeC executable are:

```
Nairo Quintana
Chris Froome
```

7.14.5 Using special macros within a definition

The $(JOB) and $(RETRY) macros may be used within a definition of the *string* that defines a variable. This usage requires parentheses, such that proper macro substitution may take place when the macro's value is only a portion of the string.

- $(JOB) expands to the node *JobName*. If the *VARS* line appears in a DAG file used as a splice file, then $(JOB) will be the fully scoped name of the node.

  For example, the DAG input file lines

  ```
  JOB  NodeC NodeC.submit
  VARS NodeC nodename="$(JOB)"
  ```

  set `nodename` to `NodeC`, and the DAG input file lines

  ```
  JOB  NodeD NodeD.submit
  VARS NodeD outfilename="$(JOB)-output"
  ```

  set `outfilename` to `NodeD-output`.

- $(RETRY) expands to 0 the first time a node is run; the value is incremented each time the node is retried. For example:

  ```
  VARS NodeE noderetry="$(RETRY)"
  ```

7.14.6 Using VARS to define ClassAd attributes

The *macroname* may also begin with a `My.`, in which case it names a ClassAd attribute. For example, the VARS specification

```
VARS NodeF My.A="\"bob\""
```

results in the the `NodeF` job ClassAd attribute

```
A = "bob"
```

Continuing this example, it allows the HTCondor submit description file for NodeF to use the following line:

```
arguments = "$$([My.A])"
```

Note that while the old behavior of using the + character to signify classad attributes does work, it is not recommended over using `My.`

```
VARS NodeF +A="\"bob\""
```

will also result in

```
A = "bob"
```

7.15 DAG Manager Job Specifications

Some DAG file commands can be used to alter information about the DAG manager job itself such as adding custom classad attributes and setting information in the job environment.

7.15.1 Classad Attributes in the DAG Manager Job

The *SET_JOB_ATTR* keyword within the DAG input file specifies an attribute/value pair to be set in the DAGMan proper job's ClassAd. The syntax for *SET_JOB_ATTR* is

```
SET_JOB_ATTR AttributeName = AttributeValue
```

As an example, if the DAG input file contains:

```
SET_JOB_ATTR TestNumber = 17
```

the ClassAd of the DAGMan job itself will have an attribute `TestNumber` with the value `17`.

The attribute set by the *SET_JOB_ATTR* command is set only in the ClassAd of the DAGMan job itself - it is not propagated to node jobs of the DAG.

Values with spaces can be set by surrounding the string containing a space with single or double quotes. (Note that the quote marks themselves will be part of the value.)

Only a single attribute/value pair can be specified per *SET_JOB_ATTR* command. If the same attribute is specified multiple times in the DAG (or in multiple DAGs run by the same DAGMan instance) the last-specified value is the one that will be utilized. An attribute set in the DAG file can be overridden by specifying

```
-append 'My.<attribute> = <value>'
```

on the *condor_submit_dag* command line.

7.15.2 Environment Variables in the DAG Manager Job

The *ENV* keyword within the DAG input file can be used to specify environment variables to set into the DAGMan jobs environment or get from the environment that the DAGMan job was submitted from. It is important to know that the environment variables in the DAG manager jobs environment effect scripts and node jobs that rely environment variables since scripts and node jobs are submitted from the DAGMan jobs environment. The syntax is:

```
ENV GET VAR-1 VAR-2 ... VAR-N
#  or
ENV SET Key=Value;Key=Value; ...
```

The *GET* keyword takes a list of environment variables names to be added to the DAGMan jobs `getenv` command in the `.condor.sub` file for the DAG.

The *SET* keyword takes a semi-colon delimited list of **key=value** pairs of information to add into DAGMan jobs `environment` command in the `.condor.sub` file for the DAG. These added **key=value** must follow the normal HTCondor job environment rules.

7.16 Configuration Specific to a DAG

All configuration variables and their definitions that relate to DAGMan may be found in *Configuration File Entries for DAGMan*.

Configuration variables for *condor_dagman* can be specified in several ways, as given within the ordered list:

1. In an HTCondor configuration file.
2. With an environment variable. Prepend the string `_CONDOR_` to the configuration variable's name.
3. With a line in the DAG input file using the keyword *CONFIG*, such that there is a configuration file specified that is specific to an instance of *condor_dagman*. The configuration file specification may instead be specified on the *condor_submit_dag* command line using the **-config** option.
4. For some configuration variables, *condor_submit_dag* command line argument specifies a configuration variable. For example, the configuration variable has the corresponding command line argument *-maxjobs*.

For this ordered list, configuration values specified or parsed later in the list override ones specified earlier. For example, a value specified on the *condor_submit_dag* command line overrides corresponding values in any configuration file. And, a value specified in a DAGMan-specific configuration file overrides values specified in a general HTCondor configuration file.

The *CONFIG* command within the DAG input file specifies a configuration file to be used to set configuration variables related to *condor_dagman* when running this DAG. The syntax for *CONFIG* is

```
CONFIG dagman.config
```

then the configuration values in file `dagman.config` will be used for this DAG. If the contents of file `dagman.config` is

```
DAGMAN_MAX_JOBS_IDLE = 10
```

then this configuration is defined for this DAG.

Only a single configuration file can be specified for a given *condor_dagman* run. For example, if one file is specified within a DAG input file, and a different file is specified on the *condor_submit_dag* command line, this is a fatal error at submit time. The same is true if different configuration files are specified in multiple DAG input files and referenced in a single *condor_submit_dag* command.

If multiple DAGs are run in a single *condor_dagman* run, the configuration options specified in the *condor_dagman* configuration file, if any, apply to all DAGs, even if some of the DAGs specify no configuration file.

Configuration variables that are not for *condor_dagman* and not utilized by DaemonCore, yet are specified in a *condor_dagman*-specific configuration file are ignored.

7.17 INCLUDE

The *INCLUDE* command allows the contents of one DAG file to be parsed as if they were physically included in the referencing DAG file. The syntax for *INCLUDE* is

```
INCLUDE FileName
```

For example, if we have two DAG files like this:

```
# File name: foo.dag

JOB  A  A.sub
INCLUDE bar.dag
```

```
# File name: bar.dag

JOB  B  B.sub
JOB  C  C.sub
```

this is equivalent to the single DAG file:

```
JOB  A  A.sub
JOB  B  B.sub
JOB  C  C.sub
```

Note that the included file must be in proper DAG syntax. Also, there are many cases where a valid included DAG file will cause a parse error, such as the included files defining nodes with the same name.

*INCLUDE*s can be nested to any depth (be sure not to create a cycle of includes!).

7.17.1 Example: Using INCLUDE to simplify multiple similar workflows

One use of the *INCLUDE* command is to simplify the DAG files when we have a single workflow that we want to run on a number of data sets. In that case, we can do something like this:

```
# File name: workflow.dag
# Defines the structure of the workflow

JOB Split split.sub
JOB Process00 process.sub
...
JOB Process99 process.sub
JOB Combine combine.sub
PARENT Split CHILD Process00 ... Process99
PARENT Process00 ... Process99 CHILD Combine
```

```
# File name: split.sub

executable = my_split
input = $(dataset).phase1
output = $(dataset).phase2
...
```

```
# File name: data57.vars

VARS Split dataset="data57"
VARS Process00 dataset="data57"
...
VARS Process99 dataset="data57"
VARS Combine dataset="data57"
```

```
# File name: run_dataset57.dag

INCLUDE workflow.dag
INCLUDE data57.vars
```

Then, to run our workflow on dataset 57, we run the following command:

```
$ condor_submit_dag run_dataset57.dag
```

This avoids having to duplicate the *JOB* and *PARENT/CHILD* commands for every dataset - we can just re-use the `workflow.dag` file, in combination with a dataset-specific vars file.

7.18 ALL_NODES Option

In the following commands, a specific node name can be replaced by the option *ALL_NODES*:

- **SCRIPT**
- **PRE_SKIP**
- **RETRY**
- **ABORT-DAG-ON**
- **VARS**
- **PRIORITY**
- **CATEGORY**

This will cause the given command to apply to all nodes (except any FINAL node) in that DAG.

The ALL_NODES never applies to a FINAL node. If the *ALL_NODES* option is used in a DAG that has a FINAL node, the `dagman.out` file will contain messages noting that the FINAL node is skipped when parsing the relevant commands.

The *ALL_NODES* option is case-insensitive.

It is important to note that the *ALL_NODES* option does not apply across splices and sub-DAGs. In other words, an *ALL_NODES* option within a splice or sub-DAG will apply only to nodes within that splice or sub-DAG; also, an *ALL_NODES* option in a parent DAG will PRIORITY DAG (again, except any FINAL node).

As of version 8.5.8, the *ALL_NODES* option cannot be used when multiple DAG files are specified on the *condor_submit_dag* command line. Hopefully this limitation will be fixed in a future release.

When multiple commands (whether using the *ALL_NODES* option or not) set a given property of a DAG node, the last relevant command overrides earlier commands, as shown in the following examples:

For example, in this DAG:

```
JOB A node.sub
VARS A name="A"
VARS ALL_NODES name="X"
```

the value of *name* for node A will be "X".

In this DAG:

```
JOB A node.sub
VARS A name="A"
VARS ALL_NODES name="X"
VARS A name="foo"
```

the value of *name* for node A will be "foo".

Here is an example DAG using the *ALL_NODES* option:

```
# File: all_ex.dag
JOB A node.sub
JOB B node.sub
JOB C node.sub

SCRIPT PRE ALL_NODES my_script $JOB

VARS ALL_NODES name="$(JOB)"

# This overrides the above VARS command for node B.
VARS B name="nodeB"

RETRY all_nodes 3
```

7.19 DAGMan and Accounting Groups

As of version 8.5.6, *condor_dagman* propagates **accounting_group** and **accounting_group_user** values specified for *condor_dagman* itself to all jobs within the DAG (including sub-DAGs).

The **accounting_group** and **accounting_group_user** values can be specified using the **-append** flag to *condor_submit_dag*, for example:

```
$ condor_submit_dag -append accounting_group=group_physics -append \
  accounting_group_user=albert relativity.dag
```

See *Group Accounting* for a discussion of group accounting and *Accounting Groups with Hierarchical Group Quotas* for a discussion of accounting groups with hierarchical group quotas.

As of version 10.0.0, any explicitly set accounting group information within a DAGMan nodes job description will take precedence over the accounting information propagated down through DAGMan. This allows for easy setting of accounting information for all DAG jobs while giving a way for specific jobs to run with different accounting information.

7.20 Special Node Types

While most DAGMan nodes are the standard JOB type that run a job and possibly a PRE or POST script, special nodes can be specified in the DAG submit description to help manage the DAG and its resources in various ways.

7.20.1 FINAL Node

A FINAL node is a single and special node that is always run at the end of the DAG, even if previous nodes in the DAG have failed. A FINAL node can be used for tasks such as cleaning up intermediate files and checking the output of previous nodes. The *FINAL* command in the DAG input file specifies a node job to be run at the end of the DAG.

The syntax used for the *FINAL* command is

```
FINAL JobName SubmitDescriptionFileName [DIR directory] [NOOP]
```

The FINAL node within the DAG is identified by *JobName*, and the HTCondor job is described by the contents of the HTCondor submit description file given by *SubmitDescriptionFileName*.

The keywords *DIR* and *NOOP* are as detailed in *JOB* command documentation. If both *DIR* and *NOOP* are used, they must appear in the order shown within the syntax specification.

There may only be one FINAL node in a DAG. A parse error will be logged by the *condor_dagman* job in the `dagman.out` file, if more than one FINAL node is specified.

The FINAL node is virtually always run. It is run if the *condor_dagman* job is removed with *condor_rm*. The only case in which a FINAL node is not run is if the configuration variable is set to `True`, and a cycle is detected at start up time. If is set to `False` and a cycle is detected during the course of the run, the FINAL node will be run.

The success or failure of the FINAL node determines the success or failure of the entire DAG, overriding the status of all previous nodes. This includes any status specified by any ABORT-DAG-ON specification that has taken effect. If some nodes of a DAG fail, but the FINAL node succeeds, the DAG will be considered successful. Therefore, it is important to be careful about setting the exit status of the FINAL node.

The `$DAG_STATUS` and `$FAILED_COUNT` macros can be used both as PRE and POST script arguments, and in node job submit description files. As an example of this, here are the partial contents of the DAG input file,

```
FINAL final_node final_node.sub
SCRIPT PRE final_node final_pre.pl $DAG_STATUS $FAILED_COUNT
```

and here are the partial contents of the submit description file, `final_node.sub`

```
arguments = "$(DAG_STATUS) $(FAILED_COUNT)"
```

If there is a FINAL node specified for a DAG, it will be run at the end of the workflow. If this FINAL node must not do anything in certain cases, use the `$DAG_STATUS` and `$FAILED_COUNT` macros to take appropriate actions. Here is an example of that behavior. It uses a PRE script that aborts if the DAG has been removed with *condor_rm*, which, in turn, causes the FINAL node to be considered failed without actually submitting the HTCondor job specified for the node. Partial contents of the DAG input file:

```
FINAL final_node final_node.sub
SCRIPT PRE final_node final_pre.pl $DAG_STATUS
```

and partial contents of the Perl PRE script, `final_pre.pl`:

```
#!/usr/bin/env perl

if ($ARGV[0] eq 4) {
    exit(1);
}
```

There are restrictions on the use of a FINAL node. The DONE option is not allowed for a FINAL node. And, a FINAL node may not be referenced in any of the following specifications:

- PARENT, CHILD
- RETRY
- ABORT-DAG-ON
- PRIORITY
- CATEGORY

As of HTCondor version 8.3.7, DAGMan allows at most two submit attempts of a FINAL node, if the DAG has been removed from the queue with *condor_rm*.

7.20.2 PROVISIONER Node

A PROVISIONER node is a single and special node that is always run at the beginning of a DAG. It can be used to provision resources (ie. Amazon EC2 instances, in-memory database servers) that can then be used by the remainder of the nodes in the workflow.

The syntax used for the *PROVISIONER* command is

```
PROVISIONER JobName SubmitDescriptionFileName
```

When a PROVISIONER is defined in a DAG, it gets run at the beginning of the DAG, and no other nodes are run until the PROVISIONER has advertised that it is ready. It does this by setting the `ProvisionerState` attribute in its job classad to the enumerated value `ProvisionerState::PROVISIONING_COMPLETE` (currently: 2). Once DAGMan sees that it is ready, it will start running other nodes in the DAG as usual. At this point the PROVISIONER job continues to run, typically sleeping and waiting while other nodes in the DAG use its resources.

A PROVISIONER runs for a set amount of time defined in its job. It does not get terminated automatically at the end of a DAG workflow. The expectation is that it needs to explicitly deprovision any resources, such as expensive cloud computing instances that should not be allowed to run indefinitely.

7.20.3 SERVICE Node

A **SERVICE** node is a special type of node that is always run at the beginning of a DAG. These are typically used to run tasks that need to run alongside a DAGMan workflow (ie. progress monitoring) without any direct dependencies to the other nodes in the workflow.

The syntax used for the *SERVICE* command is

```
SERVICE ServiceName SubmitDescriptionFileName
```

When a SERVICE is defined in a DAG, it gets started at the beginning of the workflow. There is no guarantee that it will start running before any of the other nodes, although running it directly from the access point using `universe = local` or `universe = scheduler` will almost always make this go first.

A SERVICE node runs on a **best-effort basis**. If this node fails to submit correctly, this will not register as an error and the DAG workflow will continue normally.

If a DAGMan workflow finishes while there are SERVICE nodes still running, it will shut these down and then exit the workflow successfully.

7.21 Visualizing DAGs

It can be helpful to see a picture of a DAG. DAGMan can assist you in visualizing a DAG by creating the input files used by the AT&T Research Labs *graphviz* package. *dot* is a program within this package, available from http://www.graphviz.org/, and it is used to draw pictures of DAGs.

DAGMan produces one or more dot files as the result of an extra line in a DAG input file. The line appears as

```
DOT dag.dot
```

This creates a file called `dag.dot` which contains a specification of the DAG before any jobs within the DAG are submitted to HTCondor. The `dag.dot` file is used to create a visualization of the DAG by using this file as input to *dot*. This example creates a Postscript file, with a visualization of the DAG:

```
$ dot -Tps dag.dot -o dag.ps
```

Within the DAG input file, the DOT command can take several optional parameters:

- **UPDATE** This will update the dot file every time a significant update happens.
- **DONT-UPDATE** Creates a single dot file, when the DAGMan begins executing. This is the default if the parameter **UPDATE** is not used.
- **OVERWRITE** Overwrites the dot file each time it is created. This is the default, unless **DONT-OVERWRITE** is specified.
- **DONT-OVERWRITE** Used to create multiple dot files, instead of overwriting the single one specified. To create file names, DAGMan uses the name of the file concatenated with a period and an integer. For example, the DAG input file line

  ```
  DOT dag.dot DONT-OVERWRITE
  ```

  causes files `dag.dot.0`, `dag.dot.1`, `dag.dot.2`, etc. to be created. This option is most useful when combined with the **UPDATE** option to visualize the history of the DAG after it has finished executing.
- **INCLUDE** *path-to-filename* Includes the contents of a file given by `path-to-filename` in the file produced by the **DOT** command. The include file contents are always placed after the line of the form label=. This may be useful if further editing of the created files would be necessary, perhaps because you are automatically visualizing the DAG as it progresses.

If conflicting parameters are used in a DOT command, the last one listed is used.

7.22 Capturing the Status of Nodes in a File

DAGMan can capture the status of the overall DAG and all DAG nodes in a node status file, such that the user or a script can monitor this status. This file is periodically rewritten while the DAG runs. To enable this feature, the DAG input file must contain a line with the *NODE_STATUS_FILE* command.

The syntax for a *NODE_STATUS_FILE* command is

```
NODE_STATUS_FILE statusFileName [minimumUpdateTime] [ALWAYS-UPDATE]
```

The status file is written on the machine on which the DAG is submitted; its location is given by *statusFileName*, and it may be a full path and file name.

The optional *minimumUpdateTime* specifies the minimum number of seconds that must elapse between updates to the node status file. This setting exists to avoid having DAGMan spend too much time writing the node status file for very large DAGs. If no value is specified, this value defaults to 60 seconds (as of version 8.5.8; previously, it defaulted to 0). The node status file can be updated at most once per no matter how small the *minimumUpdateTime* value. Also, the node status file will be updated when the DAG finishes, whether successfully or not, even if *minimumUpdateTime* seconds have not elapsed since the last update.

Normally, the node status file is only updated if the status of some nodes has changed since the last time the file was written. However, the optional *ALWAYS-UPDATE* keyword specifies that the node status file should be updated every time the minimum update time (and), has passed, even if no nodes have changed status since the last time the file was updated. The file will change slightly, because timestamps will be updated. For performance reasons, large DAGs with approximately 10,000 or more nodes are poor candidates for using the *ALWAYS-UPDATE* option.

As an example, if the DAG input file contains the line

```
NODE_STATUS_FILE my.dag.status 30
```

the file `my.dag.status` will be rewritten at intervals of 30 seconds or more.

This node status file is overwritten each time it is updated. Therefore, it only holds information about the current status of each node; it does not provide a history of the node status.

Changed in version 8.1.6: HTCondor version 8.1.6 changes the format of the node status file.

The node status file is a collection of ClassAds in New ClassAd format. There is one ClassAd for the overall status of the DAG, one ClassAd for the status of each node, and one ClassAd with the time at which the node status file was completed as well as the time of the next update.

Here is an example portion of a node status file:

```
[
  Type = "DagStatus";
  DagFiles = {
    "job_dagman_node_status.dag"
  };
  Timestamp = 1399674138;
  DagStatus = 3;
  NodesTotal = 12;
  NodesDone = 11;
  NodesPre = 0;
  NodesQueued = 1;
  NodesPost = 0;
  NodesReady = 0;
```

(continues on next page)

(continued from previous page)

```
  NodesUnready = 0;
  NodesFailed = 0;
  JobProcsHeld = 0;
  JobProcsIdle = 1;
]
[
  Type = "NodeStatus";
  Node = "A";
  NodeStatus = 5;
  StatusDetails = "";
  RetryCount = 0;
  JobProcsQueued = 0;
  JobProcsHeld = 0;
]
...
[
  Type = "NodeStatus";
  Node = "C";
  NodeStatus = 3;
  StatusDetails = "idle";
  RetryCount = 0;
  JobProcsQueued = 1;
  JobProcsHeld = 0;
]
[
  Type = "StatusEnd";
  EndTime = 1399674138;
  NextUpdate = 1399674141;
]
```

Possible `DagStatus` and `NodeStatus` attribute values are:

- 0 (STATUS_NOT_READY): At least one parent has not yet finished or the node is a FINAL node.
- 1 (STATUS_READY): All parents have finished, but the node is not yet running.
- 2 (STATUS_PRERUN): The node's PRE script is running.
- 3 (STATUS_SUBMITTED): The node's HTCondor job(s) are in the queue.
- 4 (STATUS_POSTRUN): The node's POST script is running.
- 5 (STATUS_DONE): The node has completed successfully.
- 6 (STATUS_ERROR): The node has failed.
- 7 (STATUS_FUTILE): The node will never run because ancestor node failed.

An *ancestor* is a node that a another node depends on either directly or indirectly through a chain of **PARENT/CHILD** relationships. For example, the **DAG** shown below would result in node **G**'s *ancestors* to be nodes **A**, **B**, **D**, and **F** because the **PARENT** to **CHILD** relationships appear as `A & B -> D -> F -> G`

```
Example DAG Visualized
      A     B
      └─────
      C─────D
          E──F
```

(continues on next page)

(continued from previous page)

```
          |
          G
```

A *NODE_STATUS_FILE* command inside any splice is ignored. If multiple DAG files are specified on the *condor_submit_dag* command line, and more than one specifies a node status file, the first specification takes precedence.

7.23 Machine-Readable Event History

DAGMan can produce a machine-readable history of events. The `jobstate.log` file is designed for use by the Pegasus Workflow Management System, which operates as a layer on top of DAGMan. Pegasus uses the `jobstate.log` file to monitor the state of a workflow. The `jobstate.log` file can used by any automated tool for the monitoring of workflows.

DAGMan produces this file when the command *JOBSTATE_LOG* is in the DAG input file. The syntax for *JOBSTATE_LOG* is

JOBSTATE_LOG *JobstateLogFileName*

No more than one `jobstate.log` file can be created by a single instance of *condor_dagman*. If more than one `jobstate.log` file is specified, the first file name specified will take effect, and a warning will be printed in the `dagman.out` file when subsequent *JOBSTATE_LOG* specifications are parsed. Multiple specifications may exist in the same DAG file, within splices, or within multiple, independent DAGs run with a single *condor_dagman* instance.

The `jobstate.log` file can be considered a filtered version of the `dagman.out` file, in a machine-readable format. It contains the actual node job events that from *condor_dagman*, plus some additional meta-events.

The `jobstate.log` file is different from the node status file, in that the `jobstate.log` file is appended to, rather than being overwritten as the DAG runs. Therefore, it contains a history of the DAG, rather than a snapshot of the current state of the DAG.

There are 5 line types in the `jobstate.log` file. Each line begins with a Unix timestamp in the form of seconds since the Epoch. Fields within each line are separated by a single space character.

- **DAGMan start**: This line identifies the *condor_dagman* job. The formatting of the line is

  ```
  timestamp INTERNAL \*** DAGMAN_STARTED dagmanCondorID \***
  ```

  The *dagmanCondorID* field is the *condor_dagman* job's `ClusterId` attribute, a period, and the `ProcId` attribute.

- **DAGMan exit**: This line identifies the completion of the *condor_dagman* job. The formatting of the line is

  ```
  timestamp INTERNAL \*** DAGMAN_FINISHED exitCode \***
  ```

  The *exitCode* field is value the *condor_dagman* job returns upon exit.

- **Recovery started**: If the *condor_dagman* job goes into recovery mode, this meta-event is printed. During recovery mode, events will only be printed in the file if they were not already printed before recovery mode started. The formatting of the line is

  ```
  timestamp INTERNAL \*** RECOVERY_STARTED \***
  ```

- **Recovery finished or Recovery failure**: At the end of recovery mode, either a RECOVERY_FINISHED or RECOVERY_FAILURE meta-event will be printed, as appropriate. The formatting of the line is

```
timestamp INTERNAL \*** RECOVERY_FINISHED \***
```

or

```
timestamp INTERNAL \*** RECOVERY_FAILURE \***
```

- **Normal**: This line is used for all other event and meta-event types. The formatting of the line is

```
timestamp JobName eventName condorID jobTag - sequenceNumber
```

The *JobName* is the name given to the node job as defined in the DAG input file with the command *JOB*. It identifies the node within the DAG.

The *eventName* is one of the many defined event or meta-events given in the lists below.

The *condorID* field is the job's `ClusterId` attribute, a period, and the `ProcId` attribute. There is no *condorID* assigned yet for some meta-events, such as PRE_SCRIPT_STARTED. For these, the dash character ('-') is printed.

The *jobTag* field is defined for the Pegasus workflow manager. Its usage is generalized to be useful to other workflow managers. Pegasus-managed jobs add a line of the following form to their HTCondor submit description file:

```
+pegasus_site = "local"
```

This defines the string `local` as the *jobTag* field.

Generalized usage adds a set of 2 commands to the HTCondor submit description file to define a string as the *jobTag* field:

```
+job_tag_name = "+job_tag_value"
+job_tag_value = "viz"
```

This defines the string `viz` as the *jobTag* field. Without any of these added lines within the HTCondor submit description file, the dash character ('-') is printed for the *jobTag* field.

The *sequenceNumber* is a monotonically-increasing number that starts at one. It is associated with each attempt at running a node. If a node is retried, it gets a new sequence number; a submit failure does not result in a new sequence number. When a Rescue DAG is run, the sequence numbers pick up from where they left off within the previous attempt at running the DAG. Note that this only applies if the Rescue DAG is run automatically or with the *-dorescuefrom* command-line option.

Here is an example of a very simple Pegasus `jobstate.log` file, assuming the example *jobTag* field of `local`:

```
1292620511 INTERNAL *** DAGMAN_STARTED 4972.0 ***
1292620523 NodeA PRE_SCRIPT_STARTED - local - 1
1292620523 NodeA PRE_SCRIPT_SUCCESS - local - 1
1292620525 NodeA SUBMIT 4973.0 local - 1
1292620525 NodeA EXECUTE 4973.0 local - 1
1292620526 NodeA JOB_TERMINATED 4973.0 local - 1
1292620526 NodeA JOB_SUCCESS 0 local - 1
1292620526 NodeA POST_SCRIPT_STARTED 4973.0 local - 1
1292620531 NodeA POST_SCRIPT_TERMINATED 4973.0 local - 1
1292620531 NodeA POST_SCRIPT_SUCCESS 4973.0 local - 1
1292620535 INTERNAL *** DAGMAN_FINISHED 0 ***
```

7.24 Workflow Metrics

For every DAG, a metrics file is created. This metrics file is named <dag_file_name>.metrics, where <dag_file_name> is the name of the DAG input file. In a workflow with nested DAGs, each nested DAG will create its own metrics file.

Here is an example metrics output file:

```
{
    "client":"condor_dagman",
    "version":"8.1.0",
    "planner":"/lfs1/devel/Pegasus/pegasus/bin/pegasus-plan",
    "planner_version":"4.3.0cvs",
    "type":"metrics",
    "wf_uuid":"htcondor-test-job_dagman_metrics-A-subdag",
    "root_wf_uuid":"htcondor-test-job_dagman_metrics-A",
    "start_time":1375313459.603,
    "end_time":1375313491.498,
    "duration":31.895,
    "exitcode":1,
    "dagman_id":"26",
    "parent_dagman_id":"11",
    "rescue_dag_number":0,
    "jobs":4,
    "jobs_failed":1,
    "jobs_succeeded":3,
    "dag_jobs":0,
    "dag_jobs_failed":0,
    "dag_jobs_succeeded":0,
    "total_jobs":4,
    "total_jobs_run":4,
    "total_job_time":0.000,
    "dag_status":2
}
```

Here is an explanation of each of the items in the file:

- `client`: the name of the client workflow software; in the example, it is `"condor_dagman"`
- `version`: the version of the client workflow software
- `planner`: the workflow planner, as read from the `braindump.txt` file
- `planner_version`: the planner software version, as read from the `braindump.txt` file
- `type`: the type of data, `"metrics"`
- `wf_uuid`: the workflow ID, generated by *pegasus-plan*, as read from the `braindump.txt` file
- `root_wf_uuid`: the root workflow ID, which is relevant for nested workflows. It is generated by *pegasus-plan*, as read from the `braindump.txt` file.
- `start_time`: the start time of the client, in epoch seconds, with millisecond precision
- `end_time`: the end time of the client, in epoch seconds, with millisecond precision
- `duration`: the duration of the client, in seconds, with millisecond precision

- `exitcode`: the *condor_dagman* exit code
- `dagman_id`: the value of the `ClusterId` attribute of the *condor_dagman* instance
- `parent_dagman_id`: the value of the `ClusterId` attribute of the parent *condor_dagman* instance of this DAG; empty if this DAG is not a SUBDAG
- `rescue_dag_number`: the number of the Rescue DAG being run, or 0 if not running a Rescue DAG
- `jobs`: the number of nodes in the DAG input file, not including SUBDAG nodes
- `jobs_failed`: the number of failed nodes in the workflow, not including SUBDAG nodes
- `jobs_succeeded`: the number of successful nodes in the workflow, not including SUBDAG nodes; this includes jobs that succeeded after retries
- `dag_jobs`: the number of SUBDAG nodes in the DAG input file
- `dag_jobs_failed`: the number of SUBDAG nodes that failed
- `dag_jobs_succeeded`: the number of SUBDAG nodes that succeeded
- `total_jobs`: the total number of jobs in the DAG input file
- `total_jobs_run`: the total number of nodes executed in a DAG. It should be equal to `jobs_succeeded + jobs_failed + dag_jobs_succeeded + dag_jobs_failed`
- `total_job_time`: the sum of the time between the first execute event and the terminated event for all jobs that are not SUBDAGs
- `dag_status`: the final status of the DAG, with values
  - `0`: OK
  - `1`: error; an error condition different than those listed here
  - `2`: one or more nodes in the DAG have failed
  - `3`: the DAG has been aborted by an ABORT-DAG-ON specification
  - `4`: removed; the DAG has been removed by *condor_rm*
  - `5`: a cycle was found in the DAG
  - `6`: the DAG has been halted; see the *Suspending a Running DAG* section. for an explanation of halting a DAG

  Note that any `dag_status` other than 0 corresponds to a non-zero exit code.

The `braindump.txt` file is generated by *pegasus-plan*; the name of the `braindump.txt` file is specified with the `PEGASUS_BRAINDUMP_FILE` environment variable. If not specified, the file name defaults to `braindump.txt`, and it is placed in the current directory.

Note that the `total_job_time` value is always zero, because the calculation of that value has not yet been implemented.

CHAPTER

EIGHT

PYTHON BINDINGS

The HTCondor Python bindings expose a Pythonic interface to the HTCondor client libraries. They utilize the same C++ libraries as HTCondor itself, meaning they have nearly the same behavior as the command line tools.

Installing the Bindings
Instructions on installing the HTCondor Python bindings.

HTCondor Python Bindings Tutorials
Learn how to use the HTCondor Python bindings.

classad API Reference
Documentation for `classad`.

htcondor API Reference
Documentation for `htcondor`.

htcondor.htchirp API Reference
Documentation for `htcondor.htchirp`.

htcondor.dags API Reference
Documentation for `htcondor.dags`.

htcondor.personal API Reference
Documentation for `htcondor.personal`.

8.1 Installing the Bindings

The HTCondor Python bindings are available from a variety of sources, depending on what platform you are on and what tool you want to use to do the installation.

8.1.1 Linux System Packages

Availability: RHEL; CentOS; Debian; Ubuntu

The bindings are available as a package in various Linux system package repositories. The packages will automatically be installed if you install HTCondor itself from our repositories. This method will let you use the Python bindings in your system Python installation.

8.1.2 Windows Installer

Availability: Windows

The bindings are packaged in the Windows installer. Download the `.msi` for the version of your choice from the table here and run it. After installation, the bindings packages will be in `lib\python` in your install directory (e.g., `C:\condor\lib\python`). Add this directory to your `PYTHONPATH` environment variable to use the bindings.

8.1.3 PyPI

Availability: Linux

The bindings are available on PyPI. To install from PyPI using `pip`, run

```
python -m pip install htcondor
```

8.1.4 Conda

Availability: Linux

The bindings are available on conda-forge. To install using `conda`, run

```
conda install -c conda-forge python-htcondor
```

8.2 HTCondor Python Bindings Tutorials

These tutorials are also available as a series of runnable Jupyter notebooks via Binder:

If Binder is not working for some reason, you can also try running them using the instructions in the GitHub repository.

The HTCondor Python bindings provide a powerful mechanism to interact with HTCondor from a Python program. They utilize the same C++ libraries as HTCondor itself, meaning they have nearly the same behavior as the command line tools.

In these tutorials you will learn the basics of the Python bindings and how to use them. They are broken down into a few major sections:

- **Introductory Topics**, quick overviews of the major features of the bindings.
- **Advanced Topics**, in-depth examinations of the nooks and crannies of the system.

8.2.1 Introductory Tutorials

These tutorials cover the basics of the Python bindings and how to use them through a quick overview of the major components.

1. *Submitting and Managing Jobs* - How to submit and manage HTCondor jobs from Python.
2. *ClassAds Introduction* - The essentials of the ClassAd language.
3. *HTCondor Introduction* - How to interact with the individual HTCondor daemons.

8.2.2 Advanced Tutorials

The advanced tutorials are in-depth looks at specific pieces of the Python bindings. Each is meant to be stand-alone and should only require knowledge from the introductory tutorials.

1. *Advanced Job Submission and Management* - More details on submitting and managing jobs from Python.
2. *Advanced Schedd Interaction* - Performing transactions in the schedd and querying history.
3. *Interacting with Daemons* - Generic commands that work with any HTCondor daemon.
4. *Scalable Job Tracking* - Techniques for keeping close track of many jobs without overloading the schedd.
5. *DAG Creation and Submission* - Using `htcondor.dags` to create and submit a DAG.
6. *Personal Pools* - Using `htcondor.personal` to create and manage a "personal" HTCondor pool.

Submitting and Managing Jobs

Launch this tutorial in a Jupyter Notebook on Binder:

What is HTCondor?

An HTCondor pool provides a way for you (as a user) to submit units of work, called **jobs**, to be executed on a distributed network of computing resources. HTCondor provides tools to monitor your jobs as they run, and make certain kinds of changes to them after submission, which we call "managing" jobs.

In this tutorial, we will learn how to submit and manage jobs *from Python*. We will see how to submit jobs with various toy executables, how to ask HTCondor for information about them, and how to tell HTCondor to do things with them. All of these things are possible from the command line as well, using tools like `condor_submit`, `condor_qedit`, and `condor_hold`. However, working from Python instead of the command line gives us access to the full power of Python to do things like generate jobs programmatically based on user input, pass information consistently from submission to management, or even expose an HTCondor pool to a web application.

We start by importing the HTCondor Python bindings modules, which provide the functions we will need to talk to HTCondor.

[1]:
```
import htcondor  # for submitting jobs, querying HTCondor daemons, etc.
import classad   # for interacting with ClassAds, HTCondor's internal data format
```

Submitting a Simple Job

To submit a job, we must first describe it. A submit description is held in a `Submit` object. `Submit` objects consist of key-value pairs, and generally behave like Python dictionaries. If you're familiar with HTCondor's submit file syntax, you should think of each line in the submit file as a single key-value pair in the `Submit` object.

Let's start by writing a `Submit` object that describes a job that executes the `hostname` command on an execute node, which prints out the "name" of the node. Since `hostname` prints its results to standard output (stdout), we will capture stdout and bring it back to the submit machine so we can see the name.

[2]:
```
hostname_job = htcondor.Submit({
    "executable": "/bin/hostname",  # the program to run on the execute node
    "output": "hostname.out",       # anything the job prints to standard output will␣
↪end up in this file
    "error": "hostname.err",        # anything the job prints to standard error will end␣
↪up in this file
    "log": "hostname.log",          # this file will contain a record of what happened␣
↪to the job
    "request_cpus": "1",            # how many CPU cores we want
    "request_memory": "128MB",      # how much memory we want
    "request_disk": "128MB",        # how much disk space we want
})

print(hostname_job)
```

```
executable = /bin/hostname
output = hostname.out
error = hostname.err
log = hostname.log
request_cpus = 1
request_memory = 128MB
request_disk = 128MB
```

The available descriptors are documented in the `condor_submit` manual page <https://htcondor.readthedocs.io/en/latest/man-pages/condor_submit.html>`__. The keys of the Python dictionary you pass to `htcondor.Submit` should be the same as for the submit descriptors, and the values should be **strings containing exactly what would go on the right-hand side**.

Note that we gave the `Submit` object several relative filepaths. These paths are relative to the directory containing this Jupyter notebook (or, more generally, the current working directory). When we run the job, you should see those files appear in the file browser on the left as HTCondor creates them.

Now that we have a job description, let's submit a job. The `htcondor.Schedd.submit` method returns a `SubmitResult` object that contains information about the job, such as its `ClusterId`.

[3]:
```
schedd = htcondor.Schedd()                  # get the Python representation of the␣
↪scheduler
submit_result = schedd.submit(hostname_job)  # submit the job
print(submit_result.cluster())               # print the job's ClusterId
```

```
13
```

The job's `ClusterId` uniquely identifies this submission. Later in this module, we will use it to ask the HTCondor scheduler for information about our jobs.

For now, our job will hopefully have finished running. You should be able to see the files in the file browser on the left. Try opening one of them and seeing what's inside.

We can also look at the output from inside Python:

[4]:
```
import os
import time

output_path = "hostname.out"

# this is a crude way to wait for the job to finish
# see the Advanced tutorial "Scalable Job Tracking" for better methods!
while not os.path.exists(output_path):
    print("Output file doesn't exist yet; sleeping for one second")
    time.sleep(1)

with open(output_path, mode = "r") as f:
    print(f.read())
```

```
2ca25178243f
```

If you got some text, it worked!

If the file never shows up, it means your job didn't run. You might try looking at the `log` or `error` files specified in the submit description to see if there is any useful information in them about why the job failed.

Submitting Multiple Jobs

By default, each `submit` will submit a single job. A more common use case is to submit many jobs at once, often sharing some base submit description. Let's write a new submit description which runs `sleep`.

When we have multiple **jobs** in a single **cluster**, each job will be identified not just by its **ClusterId** but also by a **ProcID**. We can use the ProcID to separate the output and error files for each individual job. Anything that looks like `$(...)` in a submit description is a **macro**, a placeholder which will be "expanded" later by HTCondor into a real value for that particular job. The ProcID expands to a series of incrementing integers, starting at 0. So the first job in a cluster will have ProcID 0, the next will have ProcID 1, etc.

[5]:
```
sleep_job = htcondor.Submit({
    "executable": "/bin/sleep",
    "arguments": "10s",                # sleep for 10 seconds
    "output": "sleep-$(ProcId).out",   # output and error for each job, using the
↪$(ProcId) macro
    "error": "sleep-$(ProcId).err",
    "log": "sleep.log",                # we still send all of the HTCondor logs for every␣
↪job to the same file (not split up!)
    "request_cpus": "1",
    "request_memory": "128MB",
    "request_disk": "128MB",
})

print(sleep_job)
```

```
executable = /bin/sleep
arguments = 10s
```

(continues on next page)

(continued from previous page)

```
output = sleep-$(ProcId).out
error = sleep-$(ProcId).err
log = sleep.log
request_cpus = 1
request_memory = 128MB
request_disk = 128MB
```

We will submit 10 of these jobs. All we need to change from our previous `submit` call is to add the `count` keyword argument.

[6]:
```
schedd = htcondor.Schedd()
submit_result = schedd.submit(sleep_job, count=10)  # submit 10 jobs

print(submit_result.cluster())
```

```
14
```

Now that we have a bunch of jobs in flight, we might want to check how they're doing. We can ask the HTCondor scheduler about jobs by using its `query` method. We give it a **constraint**, which tells it which jobs to look for, and a **projection**, which tells it what information to return.

[7]:
```
schedd.query(
    constraint=f"ClusterId == {submit_result.cluster()}",
    projection=["ClusterId", "ProcId", "Out"],
)
```

[7]:
```
[[ ClusterId = 14; ProcId = 0; Out = "sleep-0.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 1; Out = "sleep-1.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 2; Out = "sleep-2.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 3; Out = "sleep-3.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 4; Out = "sleep-4.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 5; Out = "sleep-5.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 6; Out = "sleep-6.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 7; Out = "sleep-7.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 8; Out = "sleep-8.out"; ServerTime = 1631798183 ],
 [ ClusterId = 14; ProcId = 9; Out = "sleep-9.out"; ServerTime = 1631798183 ]]
```

There are a few things to notice here: - Depending on how long it took you to run the cell, you may only get a few of your 10 jobs in the query. Jobs that have finished **leave the queue**, and will no longer show up in queries. To see those jobs, you must use the `history` method instead, which behaves like `query`, but **only** looks at jobs that have left the queue. - The results may not have come back in ProcID-sorted order. If you want to guarantee the order of the results, you must do so yourself. - Attributes are often renamed between the submit description and the actual job description in the queue. See the manual for a description of the job attribute names. - The objects returned by the query are instances of `ClassAd`. ClassAds are the common data exchange format used by HTCondor. In Python, they mostly behave like dictionaries.

Using Itemdata to Vary Over Parameters

By varying some part of the submit description using the ProcID, we can change how each individual job behaves. Perhaps it will use a different input file, or a different argument. However, we often want more flexibility than that. Perhaps our input files are named after different cities, or by timestamp, or some other naming scheme that already exists.

To use such information in the submit description, we need to use **itemdata**. Itemdata lets us pass arbitrary extra information when we queue, which we can reference with macros inside the submit description. This lets use the full power of Python to generate the submit descriptions for our jobs.

Let's mock this situation out by generating some files with randomly-chosen names. We'll also switch to using `pathlib.Path`, Python's more modern file path manipulation library.

[8]:
```
from pathlib import Path
import random
import string
import shutil

def random_string(length):
    """Produce a random lowercase ASCII string with the given length."""
    return "".join(random.choices(string.ascii_lowercase, k = length))

# make a directory to hold the input files, clearing away any existing directory
input_dir = Path.cwd() / "inputs"
shutil.rmtree(input_dir, ignore_errors = True)
input_dir.mkdir()

# make 5 input files
for idx in range(5):
    rs = random_string(5)
    input_file = input_dir / "{}.txt".format(rs)
    input_file.write_text("Hello from job {}".format(rs))
```

Now we'll get a list of all the files we just created in the input directory. This is precisely the kind of situation where Python affords us a great deal of flexibility over a submit file: we can use Python instead of the HTCondor submit language to generate and inspect the information we're going to put into the submit description.

[9]:
```
input_files = list(input_dir.glob("*.txt"))

for path in input_files:
    print(path)
```

```
/home/jovyan/tutorials/inputs/juvsl.txt
/home/jovyan/tutorials/inputs/lyitt.txt
/home/jovyan/tutorials/inputs/pnzjh.txt
/home/jovyan/tutorials/inputs/qyeet.txt
/home/jovyan/tutorials/inputs/uhmiu.txt
```

Now we'll make our submit description. Our goal is just to print out the text held in each file, which we can do using `cat`.

We will tell HTCondor to transfer the input file to the execute location by including it in `transfer_input_files`. We also need to call `cat` on the right file via `arguments`. Keep in mind that HTCondor will move the files in `transfer_input_files` directly to the scratch directory on the execute machine, so instead of the full path, we just need the file's "name", the last component of its path. `pathlib` will make it easy to extract this information.

[10]:
```
cat_job = htcondor.Submit({
    "executable": "/bin/cat",
    "arguments": "$(input_file_name)",          # we will pass in the value for this␣
↪macro via itemdata
    "transfer_input_files": "$(input_file)",    # we also need HTCondor to move the file␣
↪to the execute node
    "should_transfer_files": "yes",             # force HTCondor to transfer files even␣
↪though we're running entirely inside a container (and it normally wouldn't need to)
    "output": "cat-$(ProcId).out",
    "error": "cat-$(ProcId).err",
    "log": "cat.log",
    "request_cpus": "1",
    "request_memory": "128MB",
    "request_disk": "128MB",
})

print(cat_job)
```

```
executable = /bin/cat
arguments = $(input_file_name)
transfer_input_files = $(input_file)
should_transfer_files = yes
output = cat-$(ProcId).out
error = cat-$(ProcId).err
log = cat.log
request_cpus = 1
request_memory = 128MB
request_disk = 128MB
```

The itemdata should be passed as a list of dictionaries, where the keys are the macro names to replace in the submit description. In our case, the keys are `input_file` and `input_file_name`, so should have a list of 10 dictionaries, each with two entries. HTCondor expects the input file list to be a comma-separated list of POSIX-style paths, so we explicitly convert our `Path` to a POSIX string.

[11]:
```
itemdata = [{"input_file": path.as_posix(), "input_file_name": path.name} for path in␣
↪input_files]

for item in itemdata:
    print(item)
```

```
{'input_file': '/home/jovyan/tutorials/inputs/juvsl.txt', 'input_file_name': 'juvsl.txt'}
{'input_file': '/home/jovyan/tutorials/inputs/lyitt.txt', 'input_file_name': 'lyitt.txt'}
{'input_file': '/home/jovyan/tutorials/inputs/pnzjh.txt', 'input_file_name': 'pnzjh.txt'}
{'input_file': '/home/jovyan/tutorials/inputs/qyeet.txt', 'input_file_name': 'qyeet.txt'}
{'input_file': '/home/jovyan/tutorials/inputs/uhmiu.txt', 'input_file_name': 'uhmiu.txt'}
```

Now we'll submit the jobs, adding the `itemdata` parameter to the `submit` call:

[12]:
```
schedd = htcondor.Schedd()
submit_result = schedd.submit(cat_job, itemdata = iter(itemdata))  # submit one job for␣
↪each item in the itemdata

print(submit_result.cluster())
```

```
15
```

Let's do a query to make sure we got the itemdata right (these jobs run fast, so you might need to re-run the jobs if your first run has already left the queue):

[13]:
```
schedd.query(
    constraint=f"ClusterId == {submit_result.cluster()}",
    projection=["ClusterId", "ProcId", "Out", "Args", "TransferInput"],
)
```

[13]:
```
[[ Args = "juvsl.txt"; ClusterId = 15; ProcId = 0; Out = "cat-0.out"; TransferInput = "/
↪home/jovyan/tutorials/inputs/juvsl.txt"; ServerTime = 1631798183 ],
 [ Args = "lyitt.txt"; ClusterId = 15; ProcId = 1; Out = "cat-1.out"; TransferInput = "/
↪home/jovyan/tutorials/inputs/lyitt.txt"; ServerTime = 1631798183 ],
 [ Args = "pnzjh.txt"; ClusterId = 15; ProcId = 2; Out = "cat-2.out"; TransferInput = "/
↪home/jovyan/tutorials/inputs/pnzjh.txt"; ServerTime = 1631798183 ],
 [ Args = "qyeet.txt"; ClusterId = 15; ProcId = 3; Out = "cat-3.out"; TransferInput = "/
↪home/jovyan/tutorials/inputs/qyeet.txt"; ServerTime = 1631798183 ],
 [ Args = "uhmiu.txt"; ClusterId = 15; ProcId = 4; Out = "cat-4.out"; TransferInput = "/
↪home/jovyan/tutorials/inputs/uhmiu.txt"; ServerTime = 1631798183 ]]
```

And let's take a look at all the output:

[14]:
```
# again, this is very crude - see the advanced tutorials!
while not len(list(Path.cwd().glob("cat-*.out"))) == len(itemdata):
    print("Not all output files exist yet; sleeping for one second")
    time.sleep(1)

for output_file in Path.cwd().glob("cat-*.out"):
    print(output_file, "->", output_file.read_text())
```

```
/home/jovyan/tutorials/cat-0.out -> Hello from job ilmzj
/home/jovyan/tutorials/cat-1.out -> Hello from job lddhl
/home/jovyan/tutorials/cat-2.out -> Hello from job nsxcj
/home/jovyan/tutorials/cat-3.out -> Hello from job rycnn
/home/jovyan/tutorials/cat-4.out -> Hello from job veamy
```

Managing Jobs

Once a job is in queue, the scheduler will try its best to execute it to completion. There are several cases where you may want to interrupt the normal flow of jobs. Perhaps the results are no longer needed; perhaps the job needs to be edited to correct a submission error. These actions fall under the purview of **job management**.

There are two `Schedd` methods dedicated to job management:

- `edit()`: Change an attribute for a set of jobs.
- `act()`: Change the state of a job (remove it from the queue, hold it, suspend it, etc.).

The `act` method takes an argument from the `JobAction` enum. Commonly-used values include:

- `Hold`: put a job on hold, vacating a running job if necessary. A job will stay in the hold state until told otherwise.
- `Release`: Release a job from the hold state, returning it to Idle.
- `Remove`: Remove a job from the queue. If it is running, it will stop running. This requires the execute node to acknowledge it has successfully vacated the job, so `Remove` may not be instantaneous.

- `Vacate`: Cause a running job to be killed on the remote resource and return to the Idle state. With `Vacate`, jobs may be given significant time to cleanly shut down.

To play with this, let's bring back our sleep submit description, but increase the sleep time significantly so that we have time to interact with the jobs.

[15]:
```
long_sleep_job = htcondor.Submit({
    "executable": "/bin/sleep",
    "arguments": "10m",                 # sleep for 10 minutes
    "output": "sleep-$(ProcId).out",
    "error": "sleep-$(ProcId).err",
    "log": "sleep.log",
    "request_cpus": "1",
    "request_memory": "128MB",
    "request_disk": "128MB",
})

print(long_sleep_job)
```

```
executable = /bin/sleep
arguments = 10m
output = sleep-$(ProcId).out
error = sleep-$(ProcId).err
log = sleep.log
request_cpus = 1
request_memory = 128MB
request_disk = 128MB
```

[16]:
```
schedd = htcondor.Schedd()
submit_result = schedd.submit(long_sleep_job, count=5)
```

As an experiment, let's set an arbitrary attribute on the jobs and check that it worked. When we're really working, we could do things like change the amount of memory a job has requested by editing its `RequestMemory` attribute. The job attributes that are built-in to HTCondor are described here, but your site may specify additional, custom attributes as well.

[17]:
```
# sets attribute foo to the string "bar" for all of our jobs
# note the nested quotes around bar! The outer "" make it a Python string; the inner ""␣
↪make it a ClassAd string.
schedd.edit(f"ClusterId == {submit_result.cluster()}", "foo", "\"bar\"")

# do a query to check the value of attribute foo
schedd.query(
    constraint=f"ClusterId == {submit_result.cluster()}",
    projection=["ClusterId", "ProcId", "JobStatus", "foo"],
)
```

[17]:
```
[[ ClusterId = 16; ProcId = 0; foo = "bar"; JobStatus = 1; ServerTime = 1631798184 ],
 [ ClusterId = 16; ProcId = 1; foo = "bar"; JobStatus = 1; ServerTime = 1631798184 ],
 [ ClusterId = 16; ProcId = 2; foo = "bar"; JobStatus = 1; ServerTime = 1631798184 ],
 [ ClusterId = 16; ProcId = 3; foo = "bar"; JobStatus = 1; ServerTime = 1631798184 ],
 [ ClusterId = 16; ProcId = 4; foo = "bar"; JobStatus = 1; ServerTime = 1631798184 ]]
```

Although the job status appears to be an attribute, we cannot `edit` it directly. As mentioned above, we must instead `act` on the job. Let's hold the first two jobs so that they stop running, but leave the others going.

[18]:
```
# hold the first two jobs
schedd.act(htcondor.JobAction.Hold, f"ClusterId == {submit_result.
↪cluster()} && ProcID <= 1")

# check the status of the jobs
ads = schedd.query(
    constraint=f"ClusterId == {submit_result.cluster()}",
    projection=["ClusterId", "ProcId", "JobStatus"],
)

for ad in ads:
    # the ClassAd objects returned by the query act like dictionaries, so we can extract␣
↪individual values out of them using []
    print(f"ProcID = {ad['ProcID']} has JobStatus = {ad['JobStatus']}")
```

```
ProcID = 0 has JobStatus = 5
ProcID = 1 has JobStatus = 5
ProcID = 2 has JobStatus = 1
ProcID = 3 has JobStatus = 1
ProcID = 4 has JobStatus = 1
```

The various job statuses are represented by numbers. `1` means `Idle`, `2` means `Running`, and `5` means `Held`. If you see `JobStatus = 5` above for `ProcID = 0` and `ProcID = 1`, then we succeeded!

The opposite of `JobAction.Hold` is `JobAction.Release`. Let's release those jobs and let them go back to `Idle`.

[19]:
```
schedd.act(htcondor.JobAction.Release, f"ClusterId == {submit_result.cluster()}")

ads = schedd.query(
    constraint=f"ClusterId == {submit_result.cluster()}",
    projection=["ClusterId", "ProcId", "JobStatus"],
)

for ad in ads:
    # the ClassAd objects returned by the query act like dictionaries, so we can extract␣
↪individual values out of them using []
    print(f"ProcID = {ad['ProcID']} has JobStatus = {ad['JobStatus']}")
```

```
ProcID = 0 has JobStatus = 1
ProcID = 1 has JobStatus = 1
ProcID = 2 has JobStatus = 1
ProcID = 3 has JobStatus = 1
ProcID = 4 has JobStatus = 1
```

Note that we simply released all the jobs in the cluster. Releasing a job that is not held doesn't do anything, so we don't have to be extremely careful.

Finally, let's clean up after ourselves:

[20]:
```
schedd.act(htcondor.JobAction.Remove, f"ClusterId == {submit_result.cluster()}")
```

[20]:
```
[ TotalJobAds = 0; TotalPermissionDenied = 0; TotalAlreadyDone = 0; TotalNotFound = 0;␣
↪TotalSuccess = 5; TotalChangedAds = 1; TotalBadStatus = 0; TotalError = 0 ]
```

Exercises

Now let's practice what we've learned.

- In each exercise, you will be given a piece of code and a test that does not yet pass.
- The exercises are vaguely in order of increasing difficulty.
- Modify the code, or add new code to it, to pass the test. Do whatever it takes!
- You can run the test by running the block it is in.
- Feel free to look at the test for clues as to how to modify the code.
- Many of the exercises can be solved either by using Python to generate inputs, or by using advanced features of the ClassAd language. Either way is valid!
- Don't modify the test. That's cheating!

Exercise 1: Incrementing Sleeps

Submit five jobs which sleep for 5, 6, 7, 8, and 9 seconds, respectively.

[21]:
```
# MODIFY OR ADD TO THIS BLOCK...

incrementing_sleep = htcondor.Submit({
    "executable": "/bin/sleep",
    "arguments": "1",
    "output": "ex1-$(ProcId).out",
    "error": "ex1-$(ProcId).err",
    "log": "ex1.log",
    "request_cpus": "1",
    "request_memory": "128MB",
    "request_disk": "128MB",
})

schedd = htcondor.Schedd()
submit_result = schedd.submit(incrementing_sleep)
```

[22]:
```
# ... TO MAKE THIS TEST PASS

expected = [str(i) for i in range(5, 10)]
print("Expected ", expected)

ads = schedd.query(f"ClusterId == {submit_result.cluster()}", projection = ["Args"])
arguments = sorted(ad["Args"] for ad in ads)
print("Got      ", arguments)

assert arguments == expected, "Arguments were not what we expected!"
print("The test passed. Good job!")
```

```
Expected  ['5', '6', '7', '8', '9']
Got       ['1']
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
/tmp/ipykernel_454/3067880786.py in <module>
      8 print("Got      ", arguments)
      9
---> 10 assert arguments == expected, "Arguments were not what we expected!"
     11 print("The test passed. Good job!")

AssertionError: Arguments were not what we expected!
```

Exercise 2: Echo to Target

Run a job that makes the text `Echo to Target` appear in a file named `ex3.txt`.

[23]:
```
# MODIFY OR ADD TO THIS BLOCK...

echo = htcondor.Submit({
    "request_cpus": "1",
    "request_memory": "128MB",
    "request_disk": "128MB",
})

schedd = htcondor.Schedd()
submit_result = schedd.submit(echo)
```

```
---------------------------------------------------------------------------
HTCondorInternalError                     Traceback (most recent call last)
/tmp/ipykernel_454/2917236442.py in <module>
      8
      9 schedd = htcondor.Schedd()
---> 10 submit_result = schedd.submit(echo)

/opt/conda/lib/python3.9/site-packages/htcondor/_lock.py in wrapper(*args, **kwargs)
     67             acquired = LOCK.acquire()
     68
---> 69             rv = func(*args, **kwargs)
     70
     71             # if the function returned a context manager,

HTCondorInternalError: No 'executable' parameter was provided
```

[24]:
```
# ... TO MAKE THIS TEST PASS

does_file_exist = os.path.exists("ex3.txt")
assert does_file_exist, "ex3.txt does not exist!"

expected = "Echo to Target"
print("Expected ", expected)

contents = open("ex3.txt", mode = "r").read().strip()
print("Got      ", contents)

assert expected in contents, "Contents were not what we expected!"
```

(continues on next page)

(continued from previous page)

```
print("The test passed. Good job!")
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_454/1707749984.py in <module>
      2
      3 does_file_exist = os.path.exists("ex3.txt")
----> 4 assert does_file_exist, "ex3.txt does not exist!"
      5
      6 expected = "Echo to Target"

AssertionError: ex3.txt does not exist!
```

Exercise 3: Holding Odds

Hold all of the odd-numbered jobs in this large cluster.

- Note that the test block **removes all of the jobs you own** when it runs, to prevent these long-running jobs from corrupting other tests!

[25]:
```
# MODIFY OR ADD TO THIS BLOCK...

long_sleep = htcondor.Submit({
    "executable": "/bin/sleep",
    "arguments": "10m",
    "output": "ex2-$(ProcId).out",
    "error": "ex2-$(ProcId).err",
    "log": "ex2.log",
    "request_cpus": "1",
    "request_memory": "128MB",
    "request_disk": "128MB",
})

schedd = htcondor.Schedd()
submit_result = schedd.submit(long_sleep, count=100)
```

[26]:
```
# ... TO MAKE THIS TEST PASS

import getpass

try:
    ads = schedd.query(f"ClusterId == {submit_result.cluster()}", projection = ["ProcID",
↪ "JobStatus"])
    proc_to_status = {int(ad["ProcID"]): ad["JobStatus"] for ad in sorted(ads, key =␣
↪lambda ad: ad["ProcID"])}

    for proc, status in proc_to_status.items():
        print("Proc {} has status {}".format(proc, status))

    assert len(proc_to_status) == 100,␣
```

(continues on next page)

(continued from previous page)

```
↪"Wrong number of jobs (perhaps you need to resubmit them?)."
    assert all(status == 5 for proc, status in proc_to_status.items() if proc % 2 != 0),␣
↪"Not all odd jobs were held."
    assert all(status != 5 for proc, status in proc_to_status.items() if proc % 2 == 0),␣
↪"An even job was held."

    print("The test passed. Good job!")
finally:
    schedd.act(htcondor.JobAction.Remove, f'Owner=="{getpass.getuser()}"')
```

```
Proc 0 has status 1
Proc 1 has status 1
Proc 2 has status 1
Proc 3 has status 1
Proc 4 has status 1
Proc 5 has status 1
Proc 6 has status 1
Proc 7 has status 1
Proc 8 has status 1
Proc 9 has status 1
Proc 10 has status 1
Proc 11 has status 1
Proc 12 has status 1
Proc 13 has status 1
Proc 14 has status 1
Proc 15 has status 1
Proc 16 has status 1
Proc 17 has status 1
Proc 18 has status 1
Proc 19 has status 1
Proc 20 has status 1
Proc 21 has status 1
Proc 22 has status 1
Proc 23 has status 1
Proc 24 has status 1
Proc 25 has status 1
Proc 26 has status 1
Proc 27 has status 1
Proc 28 has status 1
Proc 29 has status 1
Proc 30 has status 1
Proc 31 has status 1
Proc 32 has status 1
Proc 33 has status 1
Proc 34 has status 1
Proc 35 has status 1
Proc 36 has status 1
Proc 37 has status 1
Proc 38 has status 1
Proc 39 has status 1
Proc 40 has status 1
Proc 41 has status 1
```

(continues on next page)

(continued from previous page)

```
Proc 42 has status 1
Proc 43 has status 1
Proc 44 has status 1
Proc 45 has status 1
Proc 46 has status 1
Proc 47 has status 1
Proc 48 has status 1
Proc 49 has status 1
Proc 50 has status 1
Proc 51 has status 1
Proc 52 has status 1
Proc 53 has status 1
Proc 54 has status 1
Proc 55 has status 1
Proc 56 has status 1
Proc 57 has status 1
Proc 58 has status 1
Proc 59 has status 1
Proc 60 has status 1
Proc 61 has status 1
Proc 62 has status 1
Proc 63 has status 1
Proc 64 has status 1
Proc 65 has status 1
Proc 66 has status 1
Proc 67 has status 1
Proc 68 has status 1
Proc 69 has status 1
Proc 70 has status 1
Proc 71 has status 1
Proc 72 has status 1
Proc 73 has status 1
Proc 74 has status 1
Proc 75 has status 1
Proc 76 has status 1
Proc 77 has status 1
Proc 78 has status 1
Proc 79 has status 1
Proc 80 has status 1
Proc 81 has status 1
Proc 82 has status 1
Proc 83 has status 1
Proc 84 has status 1
Proc 85 has status 1
Proc 86 has status 1
Proc 87 has status 1
Proc 88 has status 1
Proc 89 has status 1
Proc 90 has status 1
Proc 91 has status 1
Proc 92 has status 1
Proc 93 has status 1
```

(continues on next page)

(continued from previous page)

```
Proc 94 has status 1
Proc 95 has status 1
Proc 96 has status 1
Proc 97 has status 1
Proc 98 has status 1
Proc 99 has status 1
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_454/4042351238.py in <module>
     11
     12     assert len(proc_to_status) == 100, "Wrong number of jobs (perhaps you need␣
↪to resubmit them?)."
---> 13     assert all(status == 5 for proc, status in proc_to_status.items() if proc %␣
↪2 != 0), "Not all odd jobs were held."
     14     assert all(status != 5 for proc, status in proc_to_status.items() if proc %␣
↪2 == 0), "An even job was held."
     15

AssertionError: Not all odd jobs were held.
```

ClassAds Introduction

Launch this tutorial in a Jupyter Notebook on Binder:

In this tutorial, we will learn the basics of the ClassAd language, the policy and data exchange language that underpins all of HTCondor. ClassAds are fundamental in the HTCondor ecosystem, so understanding them will be good preparation for future tutorials.

The Python implementation of the ClassAd language is in the `classad` module:

[1]:
```
import classad
```

Expressions

The ClassAd language is built around *values* and *expressions*. If you know Python, both concepts are familiar. Examples of familiar values include: - Integers (`1`, `2`, `3`), - Floating point numbers (`3.145`, `-1e-6`) - Booleans (`true` and `false`).

Examples of expressions are: - Attribute references: `foo` - Boolean expressions: `a && b` - Arithmetic expressions: `123 + c` - Function calls: `ifThenElse(foo == 123, 3.14, 5.2)`

Expressions can be evaluated to values. Unlike many programming languages, expressions are lazily-evaluated: they are kept in memory as expressions until a value is explicitly requested. ClassAds holding expressions to be evaluated later are how many internal parts of HTCondor, like job requirements, are expressed.

Expressions are represented in Python with `ExprTree` objects. The desired ClassAd expression is passed as a string to the constructor:

[2]:
```
arith_expr = classad.ExprTree("1 + 4")
print(f"ClassAd arithemetic expression: {arith_expr} (of type {type(arith_expr)})")
```

```
ClassAd arithemetic expression: 1 + 4 (of type <class 'classad.classad.ExprTree'>)
```

Expressions can be evaluated on-demand:

[3]:
```
print(arith_expr.eval())
```

```
5
```

Here's an expression that includes a ClassAd function:

[4]:
```
function_expr = classad.ExprTree("ifThenElse(4 > 6, 123, 456)")
print(f"Function expression: {function_expr}")

value = function_expr.eval()
print(f"Corresponding value: {value} (of type {type(value)})")
```

```
Function expression: ifThenElse(4 > 6,123,456)
Corresponding value: 456 (of type <class 'int'>)
```

Notice that, when possible, we convert ClassAd values to Python values. Hence, the result of evaluating the expression above is the Python `int` `456`.

There are two important values in the ClassAd language that have no direct equivalent in Python: `Undefined` and `Error`.

`Undefined` occurs when a reference occurs to an attribute that is not defined; it is analogous to a `NameError` exception in Python (but there is no concept of an exception in ClassAds). For example, evaluating an unset attribute produces `Undefined`:

[5]:
```
print(classad.ExprTree("foo").eval())
```

```
Undefined
```

`Error` occurs primarily when an expression combines two different types or when a function call occurs with the incorrect arguments. Note that even in this case, no Python exception is raised!

[6]:
```
print(classad.ExprTree('5 + "bar"').eval())
print(classad.ExprTree('ifThenElse(1, 2, 3, 4, 5)').eval())
```

```
Error
Error
```

ClassAds

The concept that makes the ClassAd language special is, of course, the *ClassAd*!

The ClassAd is analogous to a Python or JSON dictionary. *Unlike* a dictionary, which is a set of unique key-value pairs, the ClassAd object is a set of key-*expression* pairs. The expressions in the ad can contain attribute references to other keys in the ad, which will be followed when evaluated.

There are two common ways to represent ClassAds in text. The "new ClassAd" format:

```
[
  a = 1;
  b = "foo";
  c = b
]
```

And the "old ClassAd" format:

```
a = 1
b = "foo"
c = b
```

Despite the "new" and "old" monikers, "new" is over a decade old. HTCondor command line tools utilize the "old" representation. The Python bindings default to "new".

A `ClassAd` object may be initialized via a string in either of the above representation. As a ClassAd is so similar to a Python dictionary, they may also be constructed from a dictionary.

Let's construct some ClassAds!

[7]:
```
ad1 = classad.ClassAd("""
[
  a = 1;
  b = "foo";
  c = b;
  d = a + 4;
]""")
print(ad1)
```

```
[
    a = 1;
    b = "foo";
    c = b;
    d = a + 4
]
```

We can construct the same ClassAd from a dictionary:

[8]:
```
ad_from_dict = classad.ClassAd(
{
    "a": 1,
    "b": "foo",
    "c": classad.ExprTree("b"),
    "d": classad.ExprTree("a + 4"),
})
print(ad_from_dict)
```

```
[
    d = a + 4;
    c = b;
    b = "foo";
    a = 1
]
```

ClassAds are quite similar to dictionaries; in Python, the `ClassAd` object behaves similarly to a dictionary and has similar convenience methods:

[9]:
```
print(ad1["a"])
print(ad1["not_here"])
```

```
1
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/tmp/ipykernel_116/3690994919.py in <module>
      1 print(ad1["a"])
----> 2 print(ad1["not_here"])

KeyError: 'not_here'
```

[10]:
```
print(ad1.get("not_here", 5))
```

```
5
```

[11]:
```
ad1.update({"e": 8, "f": True})
print(ad1)
```

```
[
    f = true;
    e = 8;
    a = 1;
    b = "foo";
    c = b;
    d = a + 4
]
```

Remember our example of an `Undefined` attribute above? We now can evaluate references within the context of the ad:

[12]:
```
print(ad1.eval("d"))
```

```
5
```

Note that an expression is still not evaluated until requested, even if it is invalid:

[13]:
```
ad1["g"] = classad.ExprTree("b + 5")
print(ad1["g"])
print(type(ad1["g"]))
print(ad1.eval("g"))
```

```
b + 5
<class 'classad.classad.ExprTree'>
Error
```

Onto HTCondor!

ClassAds and expressions are core concepts in interacting with HTCondor. Internally, machines and jobs are represented as ClassAds; expressions are used to filter objects and to define policy.

There's much more to learn in ClassAds! For now, you have enough background to continue to the next tutorial - *HTCondor Introduction*.

HTCondor Introduction

Launch this tutorial in a Jupyter Notebook on Binder:

Let's start interacting with the HTCondor daemons!

We'll cover the basics of two daemons, the *Collector* and the *Schedd*:

- The **Collector** maintains an inventory of all the pieces of the HTCondor pool. For example, each machine that can run jobs will advertise a ClassAd describing its resources and state. In this module, we'll learn the basics of querying the collector for information and displaying results.
- The **Schedd** maintains a queue of jobs and is responsible for managing their execution. We'll learn the basics of querying the schedd.

There are several other daemons - particularly, the *Startd* and the *Negotiator* - that the Python bindings can interact with. We'll cover those in the advanced modules.

If you are running these tutorials in the provided Docker container or on Binder, a local HTCondor pool has been started in the background for you to interact with.

To get start, let's import the `htcondor` modules.

[1]:
```
import htcondor
import classad
```

Collector

We'll start with the *Collector*, which gathers descriptions of the states of all the daemons in your HTCondor pool. The collector provides both **service discovery** and **monitoring** for these daemons.

Let's try to find the Schedd information for your HTCondor pool. First, we'll create a `Collector` object, then use the `locate` method:

[2]:
```
coll = htcondor.Collector()  # create the object representing the collector
schedd_ad = coll.locate(htcondor.DaemonTypes.Schedd) # locate the default schedd

print(schedd_ad)
```

```
    [
        CondorPlatform = "$CondorPlatform: X86_64-CentOS_5.11 $";
        CondorVersion = "$CondorVersion: 9.1.3 Aug 19 2021 BuildID: UW_Python_Wheel_
↪Build $";
        Machine = "abae0fbbde81";
        MyType = "Scheduler";
        Name = "jovyan@abae0fbbde81";
        MyAddress = "<172.17.0.2:9618?addrs=172.17.0.2-9618&alias=abae0fbbde81&noUDP&
↪sock=schedd_19_eccb>"
    ]
```

The `locate` method takes a type of daemon and (optionally) a name, returning a ClassAd that describes how to contact the daemon.

A few interesting points about the above example: - Because we didn't provide the collector with a constructor, we used the default collector in the container's configuration file. If we wanted to instead query a non-default collector, we could have done `htcondor.Collector("collector.example.com")`. - We used the `DaemonTypes` enumeration to pick the kind of daemon to return. - If there were multiple schedds in the pool, the `locate` query would have failed. In such

a case, we need to provide an explicit name to the method. E.g., `coll.locate(htcondor.DaemonTypes.Schedd, "schedd.example.com")`. - The `MyAddress` field in the ad is the actual address information. You may be surprised that this is not simply a `hostname:port`; to help manage addressing in the today's complicated Internet (full of NATs, private networks, and firewalls), a more flexible structure was needed. HTCondor developers sometimes refer to this as the *sinful string*; here, *sinful* is a play on a Unix data structure, not a moral judgement.

The `locate` method often returns only enough data to contact a remote daemon. Typically, a ClassAd records significantly more attributes. For example, if we wanted to query for a few specific attributes, we would use the `query` method instead:

[3]:
```
coll.query(htcondor.AdTypes.Schedd, projection=["Name", "MyAddress",␣
↪"DaemonCoreDutyCycle"])
```

[3]:
```
[[ DaemonCoreDutyCycle = 1.486565213627500E-02; Name = "jovyan@abae0fbbde81"; MyAddress␣
↪= "<172.17.0.2:9618?addrs=172.17.0.2-9618&alias=abae0fbbde81&noUDP&sock=schedd_19_eccb>
↪" ]]
```

Here, `query` takes an `AdType` (slightly more generic than the `DaemonTypes`, as many kinds of ads are in the collector) and several optional arguments, then returns a list of ClassAds.

We used the `projection` keyword argument; this indicates what attributes you want returned. The collector may automatically insert additional attributes (here, only `MyType`); if an ad is missing a requested attribute, it is simply not set in the returned ClassAd object. If no projection is specified, then all attributes are returned.

WARNING: when possible, utilize the projection to limit the data returned. Some ads may have hundreds of attributes, making returning the entire ad an expensive operation.

The projection filters the returned *keys*; to filter out unwanted *ads*, utilize the `constraint` option. Let's do the same query again, but specify our hostname explicitly:

[4]:
```
import socket # We'll use this to automatically fill in our hostname

name = classad.quote(f"jovyan@{socket.getfqdn()}")
coll.query(
    htcondor.AdTypes.Schedd,
    constraint=f"Name =?= {name}",
    projection=["Name", "MyAddress", "DaemonCoreDutyCycle"],
)
```

[4]:
```
[[ DaemonCoreDutyCycle = 1.486565213627500E-02; Name = "jovyan@abae0fbbde81"; MyAddress␣
↪= "<172.17.0.2:9618?addrs=172.17.0.2-9618&alias=abae0fbbde81&noUDP&sock=schedd_19_eccb>
↪" ]]
```

Notes: - `constraint` accepts either an `ExprTree` or `string` object; the latter is automatically parsed as an expression. - We used the `classad.quote` function to properly quote the hostname string. In this example, we're relatively certain the hostname won't contain quotes. However, it is good practice to use the `quote` function to avoid possible SQL-injection-type attacks. Consider what would happen if the host's FQDN contained spaces and doublequotes, such as `foo.example.com" || true`!

Schedd

Let's try our hand at querying the `schedd`!

First, we'll need a schedd object. You may either create one out of the ad returned by `locate` above or use the default in the configuration file:

[5]:
```
schedd = htcondor.Schedd(schedd_ad)
print(schedd)
```

```
<htcondor.htcondor.Schedd object at 0x7f36ee8158b0>
```

Unfortunately, as there are no jobs in our personal HTCondor pool, querying the `schedd` will be boring. Let's submit a few jobs (**note** the API used below will be covered by the next module; it's OK if you don't understand it now):

[6]:
```
sub = htcondor.Submit(
    executable = "/bin/sleep",
    arguments = "5m",
)
schedd.submit(sub, count=10)
```

[6]:
```
<htcondor.htcondor.SubmitResult at 0x7f36ec0aab30>
```

We should now have 10 jobs in queue, each of which should take 5 minutes to complete.

Let's query for the jobs, paying attention to the jobs' ID and status:

[7]:
```
for job in schedd.xquery(projection=['ClusterId', 'ProcId', 'JobStatus']):
    print(repr(job))
```

```
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 3; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 4; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 5; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 6; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 7; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 8; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 9; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 2; ProcId = 0; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 1; ClusterId = 12 ]
[ ServerTime = 1631798120; JobStatus = 1; ProcId = 2; ClusterId = 12 ]
```

The `JobStatus` is an integer; the integers map into the following states: - `1`: Idle (`I`) - `2`: Running (`R`) - `3`: Removed (`X`) - `4`: Completed (`C`) - `5`: Held (`H`) - `6`: Transferring Output - `7`: Suspended

Depending on how quickly you executed the above cell, you might see all jobs idle (`JobStatus = 1`) or some jobs running (`JobStatus = 2`) above.

As with the Collector's `query` method, we can also filter out jobs using `xquery`:

[8]:
```
for ad in schedd.xquery(constraint = 'ProcId >= 5', projection=['ProcId']):
    print(ad.get('ProcId'))
```

```
5
6
7
8
9
```

Astute readers may notice that the `Schedd` object has both `xquery` and `query` methods. The difference between them is primarily how memory is managed: - `query` returns a *list* of ClassAds, meaning all objects are held in memory at once. This utilizes more memory, but the results are immediately available. - `xquery` returns an *iterator* that produces ClassAds. This only requires one ClassAd to be in memory at once.

Finally, let's clean up after ourselves (this will remove all of the jobs you own from the queue).

[9]:
```
import getpass

schedd.act(htcondor.JobAction.Remove, f'Owner == "{getpass.getuser()}"')
```

[9]:
```
[ TotalJobAds = 0; TotalPermissionDenied = 0; TotalAlreadyDone = 0; TotalNotFound = 0;␣
↪TotalSuccess = 10; TotalChangedAds = 1; TotalBadStatus = 0; TotalError = 0 ]
```

On Job Submission

Congratulations! You can now perform simple queries against the collector for worker and submit hosts, as well as simple job queries against the submit host!

It is now time to move on to *advanced job submission and management*.

Advanced Job Submission and Management

Launch this tutorial in a Jupyter Notebook on Binder:

The two most common HTCondor command line tools are `condor_q` and `condor_submit`. In the previous module, we learned about the `xquery()` method that corresponds to `condor_q`. Here, we will learn the Python binding equivalent of `condor_submit` in greater detail.

We start by importing the relevant modules:

[1]:
```
import htcondor
```

Submitting Jobs

We will submit jobs utilizing the dedicated `Submit` object.

`Submit` objects consist of key-value pairs. Unlike ClassAds, the values do not have an inherent type (such as strings, integers, or booleans); they are evaluated with macro expansion at submit time. Where reasonable, they behave like Python dictionaries:

[2]:
```
sub = htcondor.Submit({"foo": "1", "bar": "2", "baz": "$(foo)"})
print(sub)
```

```
foo = 1
bar = 2
baz = $(foo)
```

[3]:
```
sub["qux"] = 3
print(sub)
```

```
foo = 1
bar = 2
baz = $(foo)
qux = 3
```

[4]:
```
print(sub.expand("baz"))
```

```
1
```

The available attributes and their semantics are documented in the `condor_submit` manual, sowe won't repeat them here. A minimal realistic submit object may look like the following:

[5]:
```
sub = htcondor.Submit({
    "executable": "/bin/sleep",
    "arguments": "5m"
})
```

To go from a submit object to job in a schedd, one must use the `submit` method of a `htcondor.Schedd`:

[6]:
```
schedd = htcondor.Schedd()          # create a schedd object connected to the local␣
↪schedd
submit_result = schedd.submit(sub)  # queue one job
print(submit_result.cluster())      # print the job's ClusterId
```

```
1
```

By default, each invocation of `submit` will submit a single job. A more common use case is to submit many jobs at once - often identical. Suppose we don't want to submit a single "sleep" job, but 10; instead of writing a `for`-loop around the `submit` method, we can use the `count` argument:

[7]:
```
submit_result = schedd.submit(sub, count=10)

print(submit_result.cluster())
```

```
2
```

We can now query for those jobs in the queue:

[8]:
```
schedd.query(
    constraint='ClusterId =?= {}'.format(submit_result.cluster()),
    projection=["ClusterId", "ProcId", "JobStatus", "EnteredCurrentStatus"],
)
```

[8]:
```
[[ ClusterId = 2; ProcId = 0; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 1; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 2; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 3; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 4; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 5; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
```

(continues on next page)

(continued from previous page)

```
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 6; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 7; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 8; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ],
 [ ClusterId = 2; ProcId = 9; EnteredCurrentStatus = 1631798050; JobStatus = 1;␣
↪ServerTime = 1631798050 ]]
```

It's not entirely useful to submit many identical jobs – but rather each one needs to vary slightly based on its ID (the "process ID") within the job cluster. For this, the `Submit` object in Python behaves similarly to submit files: references within the submit command are evaluated as macros at submit time.

For example, suppose we want the argument to `sleep` to vary based on the process ID:

[9]:
```
sub = htcondor.Submit({"executable": "/bin/sleep", "arguments": "$(Process)s"})
```

Here, the `$(Process)` string will be substituted with the process ID at submit time.

[10]:
```
submit_result = schedd.submit(sub, count=10)

print(submit_result.cluster())

schedd.query(
    constraint='ClusterId=?={}'.format(submit_result.cluster()),
    projection=["ClusterId", "ProcId", "JobStatus", "Args"],
)
```

```
3
```

[10]:
```
[[ Args = "0s"; ClusterId = 3; ProcId = 0; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "1s"; ClusterId = 3; ProcId = 1; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "2s"; ClusterId = 3; ProcId = 2; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "3s"; ClusterId = 3; ProcId = 3; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "4s"; ClusterId = 3; ProcId = 4; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "5s"; ClusterId = 3; ProcId = 5; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "6s"; ClusterId = 3; ProcId = 6; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "7s"; ClusterId = 3; ProcId = 7; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "8s"; ClusterId = 3; ProcId = 8; JobStatus = 1; ServerTime = 1631798050 ],
 [ Args = "9s"; ClusterId = 3; ProcId = 9; JobStatus = 1; ServerTime = 1631798050 ]]
```

The macro evaluation behavior (and the various usable tricks and techniques) are identical between the python bindings and the `condor_submit` executable.

Managing Jobs

Once a job is in queue, the schedd will try its best to execute it to completion. There are several cases where a user may want to interrupt the normal flow of jobs. Perhaps the results are no longer needed; perhaps the job needs to be edited to correct a submission error. These actions fall under the purview of *job management*.

There are two `Schedd` methods dedicated to job management:

- `edit()`: Change an attribute for a set of jobs to a given expression. If invoked within a transaction, multiple calls to `edit` are visible atomically.
  - The set of jobs to change can be given as a ClassAd expression. If no jobs match the filter, *then an exception is thrown*.
- `act()`: Change the state of a job to a given state (remove, hold, suspend, etc).

Both methods take a *job specification*: either a ClassAd expression (such as `Owner =?= "janedoe"`) or a list of job IDs (such as `["1.1", "2.2", "2.3"]`). The `act` method takes an argument from the `JobAction` enum. The commonly-used values are:

- `Hold`: put a job on hold, vacating a running job if necessary. A job will stay in the hold state until explicitly acted upon by the admin or owner.
- `Release`: Release a job from the hold state, returning it to Idle.
- `Remove`: Remove a job from the Schedd's queue, cleaning it up first on the remote host (if running). This requires the remote host to acknowledge it has successfully vacated the job, meaning `Remove` may not be instantaneous.
- `Vacate`: Cause a running job to be killed on the remote resource and return to idle state. With `Vacate`, jobs may be given significant time to cleanly shut down.

Here's an example of job management in action:

[11]:
```
submit_result = schedd.submit(sub, count=5)  # queues 5 copies of this job
schedd.edit([f"{submit_result.cluster()}.{idx}" for idx in range(2)], "foo", '"bar"')  #␣
↪sets attribute foo to the string "bar" for the first two jobs

for ad in schedd.xquery(
    constraint=f"ClusterId == {submit_result.cluster()}",
    projection=["ProcId", "JobStatus", "foo"],
):
    print(repr(ad))
```

```
[ ServerTime = 1631798050; ProcId = 0; JobStatus = 1; foo = "bar" ]
[ ServerTime = 1631798050; ProcId = 1; JobStatus = 1; foo = "bar" ]
[ ServerTime = 1631798050; ProcId = 2; JobStatus = 1 ]
[ ServerTime = 1631798050; ProcId = 3; JobStatus = 1 ]
[ ServerTime = 1631798050; ProcId = 4; JobStatus = 1 ]
```

[12]:
```
schedd.act(htcondor.JobAction.Hold, f"ClusterId == {submit_result.
↪cluster()} && ProcId >= 2")

for ad in schedd.xquery(
    constraint=f"ClusterId == {submit_result.cluster()}",
    projection=["ProcId", "JobStatus", "foo"],
):
    print(repr(ad))
```

```
[ ServerTime = 1631798050; ProcId = 0; JobStatus = 1; foo = "bar" ]
[ ServerTime = 1631798050; ProcId = 1; JobStatus = 1; foo = "bar" ]
[ ServerTime = 1631798051; ProcId = 2; JobStatus = 5 ]
[ ServerTime = 1631798051; ProcId = 3; JobStatus = 5 ]
[ ServerTime = 1631798051; ProcId = 4; JobStatus = 5 ]
```

Finally, let's clean up after ourselves (this will remove all of the jobs you own from the queue).

[13]:
```
import getpass

schedd.act(htcondor.JobAction.Remove, f'Owner == "{getpass.getuser()}"')
```

[13]:
```
[ TotalJobAds = 26; TotalPermissionDenied = 0; TotalAlreadyDone = 0; TotalNotFound = 0;␣
↪TotalSuccess = 26; TotalChangedAds = 1; TotalBadStatus = 0; TotalError = 0 ]
```

That's It!

You've made it through the very basics of the Python bindings. While there are many other features the Python module has to offer, we have covered enough to replace the command line tools of `condor_q`, `condor_submit`, `condor_status`, `condor_rm` and others.

Advanced Schedd Interaction

Launch this tutorial in a Jupyter Notebook on Binder:

The introductory tutorial only scratches the surface of what the Python bindings can do with the `condor_schedd`; this module focuses on covering a wider range of functionality:

- Job and history querying.
- Advanced job submission.
- Python-based negotiation with the Schedd.

As usual, we start by importing the relevant modules:

[1]:
```
import htcondor
import classad
```

Job and History Querying

In *HTCondor Introduction*, we covered the `Schedd.xquery` method and its two most important keywords:

- `requirements`: Filters the jobs the schedd should return.
- `projection`: Filters the attributes returned for each job.

For those familiar with SQL queries, `requirements` performs the equivalent as the `WHERE` clause while `projection` performs the equivalent of the column listing in `SELECT`.

There are two other keywords worth mentioning:

- `limit`: Limits the number of returned ads; equivalent to SQL's `LIMIT`.

- `opts`: Additional flags to send to the schedd to alter query behavior. The only flag currently defined is `QueryOpts.AutoCluster`; this groups the returned results by the current set of "auto-cluster" attributes used by the pool. It's analogous to `GROUP BY` in SQL, except the columns used for grouping are controlled by the schedd.

To illustrate these additional keywords, let's first submit a few jobs:

[2]:
```
schedd = htcondor.Schedd()
sub = htcondor.Submit({
    "executable": "/bin/sleep",
    "arguments": "5m",
    "hold": "True",
})
submit_result = schedd.submit(sub, count=10)
print(submit_result.cluster())
```

```
5
```

Note: In this example, we used the `hold` submit command to indicate that the jobs should start out in the `condor_schedd` in the *Hold* state; this is used simply to prevent the jobs from running to completion while you are running the tutorial.

We now have 10 jobs running under `cluster_id`; they should all be identical:

[3]:
```
print(len(schedd.query(projection=["ProcID"], constraint=f"ClusterId=={submit_result.
↪cluster()}")))
```

```
10
```

The `sum(1 for _ in ...)` syntax is a simple way to count the number of items produced by an iterator without buffering all the objects in memory.

Querying many Schedds

On larger pools, it's common to write Python scripts that interact with not one but many schedds. For example, if you want to implement a "global query" (equivalent to `condor_q -g`; concatenates all jobs in all schedds), it might be tempting to write code like this:

[4]:
```
jobs = []
for schedd_ad in htcondor.Collector().locateAll(htcondor.DaemonTypes.Schedd):
    schedd = htcondor.Schedd(schedd_ad)
    jobs += schedd.xquery()
print(len(jobs))
```

```
10
```

This is sub-optimal for two reasons:

- `xquery` is not given any projection, meaning it will pull all attributes for all jobs - much more data than is needed for simply counting jobs.
- The querying across all schedds is serialized: we may wait for painfully long on one or two "bad apples."

We can instead begin the query for all schedds simultaneously, then read the responses as they are sent back. First, we start all the queries without reading responses:

[5]:
```
queries = []
coll_query = htcondor.Collector().locateAll(htcondor.DaemonTypes.Schedd)
for schedd_ad in coll_query:
    schedd_obj = htcondor.Schedd(schedd_ad)
    queries.append(schedd_obj.xquery())
```

The iterators will yield the matching jobs; to return the autoclusters instead of jobs, use the `AutoCluster` option (`schedd_obj.xquery(opts=htcondor.QueryOpts.AutoCluster)`). One auto-cluster ad is returned for each set of jobs that have identical values for all significant attributes. A sample auto-cluster looks like:

```
[
 RequestDisk = DiskUsage;
 Rank = 0.0;
 FileSystemDomain = "hcc-briantest7.unl.edu";
 MemoryUsage = ( ( ResidentSetSize + 1023 ) / 1024 );
 ImageSize = 1000;
 JobUniverse = 5;
 DiskUsage = 1000;
 JobCount = 1;
 Requirements = ( TARGET.Arch == "X86_64" ) && ( TARGET.OpSys == "LINUX" ) && ( TARGET.
↪Disk >= RequestDisk ) && ( TARGET.Memory >= RequestMemory ) && ( ( TARGET.
↪HasFileTransfer ) || ( TARGET.FileSystemDomain == MY.FileSystemDomain ) );
 RequestMemory = ifthenelse(MemoryUsage isnt undefined,MemoryUsage,( ImageSize + 1023 ) /
↪ 1024);
 ResidentSetSize = 0;
 ServerTime = 1483758177;
 AutoClusterId = 2
]
```

We use the `poll` function, which will return when a query has available results:

[6]:
```
job_counts = {}
for query in htcondor.poll(queries):
    schedd_name = query.tag()
    job_counts.setdefault(schedd_name, 0)
    count = len(query.nextAdsNonBlocking())
    job_counts[schedd_name] += count
    print("Got {} results from {}.".format(count, schedd_name))
print(job_counts)
```

```
Got 10 results from jovyan@abae0fbbde81.
{'jovyan@abae0fbbde81': 10}
```

The `QueryIterator.tag` method is used to identify which query is returned; the tag defaults to the Schedd's name but can be manually set through the `tag` keyword argument to `Schedd.xquery`.

History Queries

After a job has finished in the Schedd, it moves from the queue to the history file. The history can be queried (locally or remotely) with the `Schedd.history` method:

[7]:
```
schedd = htcondor.Schedd()
for ad in schedd.history(
    constraint='true',
    projection=['ProcId', 'ClusterId', 'JobStatus'],
    match=2,  # limit to 2 returned results
):
    print(ad)
```

```
[
    JobStatus = 3;
    ProcId = 0;
    ClusterId = 1
]

[
    JobStatus = 3;
    ProcId = 9;
    ClusterId = 3
]
```

[ ]:
```
```

Interacting With Daemons

Launch this tutorial in a Jupyter Notebook on Binder:

In this module, we'll look at how the HTCondor Python bindings can be used to interact with running daemons.

As usual, we start by importing the relevant modules:

[1]:
```
import htcondor
```

Configuration

The HTCondor configuration is exposed to Python in two ways:

- The local process's configuration is available in the module-level `param` object.
- A remote daemon's configuration may be queried using a `RemoteParam`

The `param` object emulates a Python dictionary:

[2]:
```
print(htcondor.param["SCHEDD_LOG"])   # prints the schedd's current log file
print(htcondor.param.get("TOOL_LOG")) # print None, since TOOL_LOG isn't set by default
```

```
/home/jovyan/.condor/local/log/SchedLog
None
```

[3]:
```
htcondor.param["TOOL_LOG"] = "/tmp/log" # sets TOOL_LOG to /tmp/log
print(htcondor.param["TOOL_LOG"])       # prints /tmp/log, as set above
```

```
/tmp/log
```

Note that assignments to `param` will persist only in memory; if we use `reload_config` to re-read the configuration files from disk, our change to `TOOL_LOG` disappears:

[4]:
```
print(htcondor.param.get("TOOL_LOG"))
htcondor.reload_config()
print(htcondor.param.get("TOOL_LOG"))
```

```
/tmp/log
None
```

In HTCondor, a configuration *prefix* may indicate that a setting is specific to that daemon. By default, the Python binding's prefix is `TOOL`. If you would like to use the configuration of a different daemon, utilize the `set_subsystem` function:

[5]:
```
htcondor.param["TEST_FOO"] = "foo"          # sets the default value of TEST_FOO to foo
htcondor.param["SCHEDD.TEST_FOO"] = "bar"  # the schedd has a special setting for TEST_
↪FOO
```

[6]:
```
print(htcondor.param['TEST_FOO'])        # default access; should be 'foo'
```

```
foo
```

[7]:
```
htcondor.set_subsystem('SCHEDD')         # changes the running process to identify as a␣
↪schedd and sets subsytem to be trusted with root privileges.
print(htcondor.param['TEST_FOO'])        # since we now identify as a schedd, should use␣
↪the special setting of 'bar'
```

```
bar
```

Between `param`, `reload_config`, and `set_subsystem`, we can explore the configuration of the local host.

Remote Configuration

What happens if we want to test the configuration of a remote daemon? For that, we can use the `RemoteParam` class.

The object is first initialized from the output of the `Collector.locate` method:

[8]:
```
master_ad = htcondor.Collector().locate(htcondor.DaemonTypes.Master)
print(master_ad['MyAddress'])
master_param = htcondor.RemoteParam(master_ad)
```

```
<172.17.0.2:9618?addrs=172.17.0.2-9618&alias=abae0fbbde81&noUDP&sock=master_19_eccb>
```

Once we have the `master_param` object, we can treat it like a local dictionary to access the remote daemon's configuration.

NOTE that the `htcondor.param` object attempts to infer type information for configuration values from the compile-time metadata while the `RemoteParam` object does not:

[9]:
```
print(repr(master_param['UPDATE_INTERVAL']))      # returns a string
print(repr(htcondor.param['UPDATE_INTERVAL']))    # returns an integer
```

```
'5'
5
```

In fact, we can even *set* the daemon's configuration using the `RemoteParam` object... if we have permission. By default, this is disabled for security reasons:

[10]:
```
master_param['UPDATE_INTERVAL'] = '500'
```

```
---------------------------------------------------------------------------
HTCondorReplyError                        Traceback (most recent call last)
/tmp/ipykernel_252/743935840.py in <module>
----> 1 master_param['UPDATE_INTERVAL'] = '500'

/opt/conda/lib/python3.9/site-packages/htcondor/_lock.py in wrapper(*args, **kwargs)
     67             acquired = LOCK.acquire()
     68
---> 69             rv = func(*args, **kwargs)
     70
     71             # if the function returned a context manager,

HTCondorReplyError: Failed to set remote daemon parameter.
```

Logging Subsystem

The logging subsystem is available to the Python bindings; this is often useful for debugging network connection issues between the client and server.

NOTE Jupyter notebooks discard output from library code; hence, you will not see the results of `enable_debug` below.

[11]:
```
htcondor.set_subsystem("TOOL")
htcondor.param['TOOL_DEBUG'] = 'D_FULLDEBUG'
htcondor.param['TOOL_LOG'] = '/tmp/log'
htcondor.enable_log()    # Send logs to the log file (/tmp/foo)
htcondor.enable_debug()  # Send logs to stderr; this is ignored by the web notebook.
print(open("/tmp/log").read())  # Print the log's contents.
```

Sending Daemon Commands

An administrator can send administrative commands directly to the remote daemon. This is useful if you'd like a certain daemon restarted, drained, or reconfigured.

Because we have a personal HTCondor instance, we are the administrator - and we can test this out!

To send a command, use the top-level `send_command` function, provide a daemon location, and provide a specific command from the `DaemonCommands` enumeration. For example, we can *reconfigure*:

[12]:
```
print(master_ad['MyAddress'])

htcondor.send_command(master_ad, htcondor.DaemonCommands.Reconfig)
```

```
<172.17.0.2:9618?addrs=172.17.0.2-9618&alias=abae0fbbde81&noUDP&sock=master_19_eccb>
```

```
09/16/21 13:15:27 SharedPortClient: sent connection request to <172.17.0.2:9618> for␣
↪shared port id master_19_eccb
```

[13]:
```
import time

time.sleep(1)

log_lines = open(htcondor.param['MASTER_LOG']).readlines()
print(log_lines[-4:])
```

```
['09/16/21 13:15:27 Sent SIGHUP to NEGOTIATOR (pid 23)\n', '09/16/21 13:15:27 Sent␣
↪SIGHUP to SCHEDD (pid 24)\n', '09/16/21 13:15:27 Sent SIGHUP to SHARED_PORT (pid 21)\n
↪', '09/16/21 13:15:27 Sent SIGHUP to STARTD (pid 27)\n']
```

We can also instruct the master to shut down a specific daemon:

[14]:
```
htcondor.send_command(master_ad, htcondor.DaemonCommands.DaemonOff, "SCHEDD")

time.sleep(1)

log_lines = open(htcondor.param['MASTER_LOG']).readlines()
print(log_lines[-1])
```

```
09/16/21 13:15:28 SharedPortClient: sent connection request to <172.17.0.2:9618> for␣
↪shared port id master_19_eccb
09/16/21 13:15:28 Can't open directory "/etc/condor/passwords.d" as PRIV_ROOT, errno: 13␣
↪(Permission denied)
09/16/21 13:15:28 Can't open directory "/etc/condor/passwords.d" as PRIV_ROOT, errno: 13␣
↪(Permission denied)
```

```
09/16/21 13:15:28 The SCHEDD (pid 24) exited with status 0
```

Or even turn off the whole HTCondor instance:

[15]:
```
htcondor.send_command(master_ad, htcondor.DaemonCommands.OffFast)

time.sleep(10)

log_lines = open(htcondor.param['MASTER_LOG']).readlines()
print(log_lines[-1])
```

```
09/16/21 13:15:29 SharedPortClient: sent connection request to <172.17.0.2:9618> for␣
↪shared port id master_19_eccb
```

```
09/16/21 13:15:30 **** condor_master (condor_MASTER) pid 19 EXITING WITH STATUS 0
```

Let's turn HTCondor back on for future tutorials:

[16]:
```
import os
os.system("condor_master")
time.sleep(10)  # give condor a few seconds to get started
```

Scalable Job Tracking

Launch this tutorial in a Jupyter Notebook on Binder:

The Python bindings provide two scalable mechanisms for tracking jobs:

- **Poll-based tracking**: The Schedd can be periodically polled through the use of `Schedd.xquery` to get job status information.
- **Event-based tracking**: Using the job's *user log*, Python can see all job events and keep an in-memory representation of the job status.

Both poll- and event-based tracking have their strengths and weaknesses; the intrepid user can even combine both methodologies to have extremely reliable, low-latency job status tracking.

In this module, we outline the important design considerations behind each approach and walk through examples.

Poll-based Tracking

Poll-based tracking involves periodically querying the schedd(s) for jobs of interest. We have covered the technical aspects of querying the Schedd in prior tutorials. Beside the technical means of polling, important aspects to consider are *how often* the poll should be performed and *how much* data should be retrieved.

Note: When `Schedd.xquery` is used, the query will cause the schedd to fork up to `SCHEDD_QUERY_WORKERS` simultaneous workers. Beyond that point, queries will be handled in a non-blocking manner inside the main `condor_schedd` process. Thus, the memory used by many concurrent queries can be reduced by decreasing `SCHEDD_QUERY_WORKERS`.

A job tracking system should not query the Schedd more than once a minute. Aim to minimize the data returned from the query through the use of the projection; minimize the number of jobs returned by using a query constraint. Better yet, use the `AutoCluster` flag to have `Schedd.xquery` return a list of job summaries instead of individual jobs.

Advantages:

- A single entity can poll all `condor_schedd` instances in a pool; using `htcondor.poll`, multiple Schedds can be queried simultaneously.
- The tracking is resilient to bugs or crashes. All tracked state is replaced at the next polling cycle.

Disadvantages:

- The amount of work to do is a function of the number of jobs in the schedd; may scale poorly once more than 100,000 simultaneous jobs are tracked.
- Each job state transition is not seen; only snapshots of the queue in time.
- If a job disappears from the Schedd, it may be difficult to determine why (Did it finish? Was it removed?)
- Only useful for tracking jobs at the minute-level granularity.

Event-based Tracking

Each job in the Schedd can specify the `UserLog` attribute; the Schedd will atomically append a machine-parseable event to the specified file for every state transition the job goes through. By keeping track of the events in the logs, we can build an in-memory representation of the job queue state.

Advantages:

- No interaction with the `condor_schedd` process is needed to read the event logs; the job tracking effectively places no burden on the Schedd.

- In most cases, the Schedd writes to the log synchronously after the event occurs. Hence, the latency of receiving an update can be sub-second.
- The job tracking scales as a function of the event rate, not the total number of jobs.
- Each job state is seen, even after the job has left the queue.

Disadvantages:

- Only the local `condor_schedd` can be tracked; there is no mechanism to receive the event log remotely.
- Log files must be processed from the beginning, with no rotations or truncations possible. Large files can take a large amount of CPU time to process.
- If every job writes to a separate log file, the job tracking software may have to keep an enormous number of open file descriptors. If every job writes to the same log file, the log file may grow to many gigabytes.
- If the job tracking software misses an event (or an unknown bug causes the `condor_schedd` to fail to write the event), then the job tracker may believe a job incorrectly is stuck in the wrong state.

At a technical level, event tracking is implemented with the htcondor.JobEventLog class.

```
>>> jel = htcondor.JobEventLog("/tmp/job_one.log")
>>> for event in jel.events(stop_after=0):
...     print event
```

The return value of `JobEventLog.events()` is an iterator over htcondor.JobEvent objects. The example above does not block.

DAG Creation and Submission

Launch this tutorial in a Jupyter Notebook on Binder:

In this tutorial, we will learn how to use `htcondor.dags` to create and submit an HTCondor DAGMan workflow. Our goal will be to create an image of the Mandelbrot set. This is a perfect problem for high-throughput computing because each point in the image can be calculated completely independently of any other point, so we are free to divide the image creation up into patches, each created by a single HTCondor job. DAGMan will enter the picture to coordinate stitching the image patches we create back into a single image.

Making a Mandelbrot set image locally

We'll use `goatbrot` (https://github.com/beejjorgensen/goatbrot) to make the image. `goatbrot` can be run from the command line, and takes a series of options to specify which part of the Mandelbrot set to draw, as well as the properties of the image itself.

`goatbrot` options: - `-i 1000` The number of iterations. - `-c 0,0` The center point of the image region. - `-w 3` The width of the image region. - `-s 1000,1000` The pixel dimensions of the image. - `-o test.ppm` The name of the output file to generate.

We can run a shell command from Jupyter by prefixing it with a `!`:

[1]:
```
! ./goatbrot -i 10 -c 0,0 -w 3 -s 500,500 -o test.ppm
! convert test.ppm test.png
```

```
Complex image:
            Center: 0 + 0i
             Width: 3
            Height: 3
```

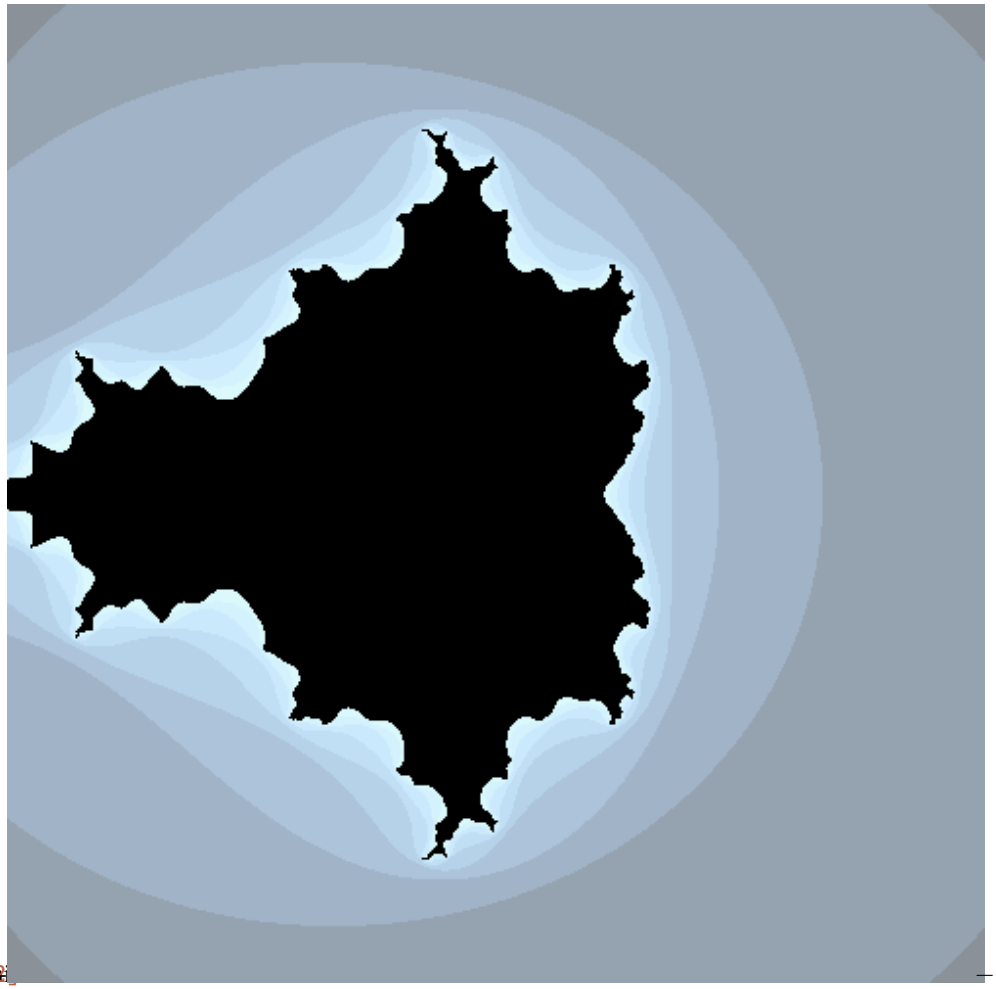
(continues on next page)

(continued from previous page)

```
        Upper Left: -1.5 + 1.5i
       Lower Right: 1.5 + -1.5i

Output image:
          Filename: test.ppm
     Width, Height: 500, 500
             Theme: beej
       Antialiased: no

Mandelbrot:
    Max Iterations: 10
        Continuous: no

Goatbrot:
    Multithreading: not supported in this build

Completed: 100.0%
```

Let's take a look at the test image. It won't be very good, because we didn't run for very many iterations. We'll use HTCondor to produce a better image!

[2]:
```
from IPython.display import Image

Image('test.png')
```



nbsphinx-code-borderwidth

What is the workflow?

We can parallelize this calculation by drawing rectangular sub-regions of the full region ("tiles") we want and stitching them together into a single image using `montage`. Let's draw this out as a graph, showing how data (image patches) will flow through the system. (Don't worry about this code, unless you want to know how to make dot diagrams in Python!)

[3]:
```
from graphviz import Digraph
import itertools

num_tiles_per_side = 2

dot = Digraph()

dot.node('montage')
for x, y in itertools.product(range(num_tiles_per_side), repeat = 2):
    n = f'tile_{x}-{y}'
    dot.node(n)
    dot.edge(n, 'montage')

dot
```

nbsphinx-code-borderwhite

Since we can chop the image up however we'd like, we have as many tiles per side as we'd like (try changing `num_tiles_per_side` above). The "shape" of the DAG is the same: there is a "layer" of `goatbrot` jobs that calculate tiles, which all feed into `montage`. Now that we know the structure of the problem, we can start describing it to HTCondor.

Describing `goatbrot` as an HTCondor job

We describe a job using a `Submit` object. It corresponds to the submit *file* used by the command line tools. It mostly behaves like a standard Python dictionary, where the keys and values correspond to submit descriptors.

[4]:
```
import htcondor

tile_description = htcondor.Submit(
    executable = 'goatbrot',  # the program we want to run
    arguments = '-i 10000 -c $(x),$(y) -w $(w) -s 500,500 -o tile_$(tile_x)-$(tile_y).ppm',  # the arguments to pass to the executable
    log = 'mandelbrot.log',  # the HTCondor job event log
    output = 'goatbrot.out.$(tile_x)_$(tile_y)',  # stdout from the job goes here
    error = 'goatbrot.err.$(tile_x)_$(tile_y)',   # stderr from the job goes here
    request_cpus = '1',        # resource requests; we don't need much per job for this problem
    request_memory = '128MB',
    request_disk = '1GB',
)

print(tile_description)
```

```
executable = goatbrot
arguments = -i 10000 -c $(x),$(y) -w $(w) -s 500,500 -o tile_$(tile_x)-$(tile_y).ppm
log = mandelbrot.log
output = goatbrot.out.$(tile_x)_$(tile_y)
error = goatbrot.err.$(tile_x)_$(tile_y)
request_cpus = 1
request_memory = 128MB
request_disk = 1GB
```

Notice the heavy use of macros like `$(x)` to specify the tile. Those aren't built-in submit macros; instead, we will plan on passing their values in through **vars**. Vars will let us customize each individual job in the tile layer by filling in those macros individually. Each job will recieve a dictionary of macro values; our next goal is to make a list of those dictionaries.

We will do this using a function that takes the number of tiles per side as an argument. As mentioned above, the **structure** of the DAG is the same no matter how "wide" the tile layer is. This is why we define a function to produce the tile vars instead of just calculating them once: we can vary the width of the DAG by passing different arguments to `make_tile_vars`. More customizations could be applied to make different images (for example, you could make it possible to set the center point of the image).

[5]:
```
def make_tile_vars(num_tiles_per_side, width = 3):
    width_per_tile = width / num_tiles_per_side
```

(continues on next page)

(continued from previous page)

```
    centers = [
        width_per_tile * (n + 0.5 - (num_tiles_per_side / 2))
        for n in range(num_tiles_per_side)
    ]

    vars = []
    for (tile_y, y), (tile_x, x) in itertools.product(enumerate(centers), repeat = 2):
        var = dict(
            w = width_per_tile,
            x = x,
            y = -y,  # image coordinates vs. Cartesian coordinates
            tile_x = str(tile_x).rjust(5, '0'),
            tile_y = str(tile_y).rjust(5, '0'),
        )

        vars.append(var)

    return vars
```

[6]:
```
tile_vars = make_tile_vars(2)
for var in tile_vars:
    print(var)
```

```
{'w': 1.5, 'x': -0.75, 'y': 0.75, 'tile_x': '00000', 'tile_y': '00000'}
{'w': 1.5, 'x': 0.75, 'y': 0.75, 'tile_x': '00001', 'tile_y': '00000'}
{'w': 1.5, 'x': -0.75, 'y': -0.75, 'tile_x': '00000', 'tile_y': '00001'}
{'w': 1.5, 'x': 0.75, 'y': -0.75, 'tile_x': '00001', 'tile_y': '00001'}
```

If we want to increase the number of tiles per side, we just pass in a larger number. Because the `tile_description` is **parameterized** in terms of these variables, it will work the same way no matter what we pass in as `vars`.

[7]:
```
tile_vars = make_tile_vars(4)
for var in tile_vars:
    print(var)
```

```
{'w': 0.75, 'x': -1.125, 'y': 1.125, 'tile_x': '00000', 'tile_y': '00000'}
{'w': 0.75, 'x': -0.375, 'y': 1.125, 'tile_x': '00001', 'tile_y': '00000'}
{'w': 0.75, 'x': 0.375, 'y': 1.125, 'tile_x': '00002', 'tile_y': '00000'}
{'w': 0.75, 'x': 1.125, 'y': 1.125, 'tile_x': '00003', 'tile_y': '00000'}
{'w': 0.75, 'x': -1.125, 'y': 0.375, 'tile_x': '00000', 'tile_y': '00001'}
{'w': 0.75, 'x': -0.375, 'y': 0.375, 'tile_x': '00001', 'tile_y': '00001'}
{'w': 0.75, 'x': 0.375, 'y': 0.375, 'tile_x': '00002', 'tile_y': '00001'}
{'w': 0.75, 'x': 1.125, 'y': 0.375, 'tile_x': '00003', 'tile_y': '00001'}
{'w': 0.75, 'x': -1.125, 'y': -0.375, 'tile_x': '00000', 'tile_y': '00002'}
{'w': 0.75, 'x': -0.375, 'y': -0.375, 'tile_x': '00001', 'tile_y': '00002'}
{'w': 0.75, 'x': 0.375, 'y': -0.375, 'tile_x': '00002', 'tile_y': '00002'}
{'w': 0.75, 'x': 1.125, 'y': -0.375, 'tile_x': '00003', 'tile_y': '00002'}
{'w': 0.75, 'x': -1.125, 'y': -1.125, 'tile_x': '00000', 'tile_y': '00003'}
{'w': 0.75, 'x': -0.375, 'y': -1.125, 'tile_x': '00001', 'tile_y': '00003'}
{'w': 0.75, 'x': 0.375, 'y': -1.125, 'tile_x': '00002', 'tile_y': '00003'}
{'w': 0.75, 'x': 1.125, 'y': -1.125, 'tile_x': '00003', 'tile_y': '00003'}
```

Describing montage as an HTCondor job

Now we can write the `montage` job description. The problem is that the arguments and input files depend on how many tiles we have, which we don't know ahead-of-time. We'll take the brute-force approach of just writing a function that takes the tile `vars` we made in the previous section and using them to build the `montage` job description.

Not that some of the work of building up the submit description is done in Python. This is a major advantage of communicating with HTCondor via Python: you can do the hard work in Python instead of in submit language!

One area for possible improvement here is to remove the duplication of the format of the input file names, which is repeated here from when it was first used in the `goatbrot` submit object. When building a larger, more complicated workflow, it is important to reduce duplication of information to make it easier to modify the workflow in the future.

[8]:
```
def make_montage_description(tile_vars):
    num_tiles_per_side = int(len(tile_vars) ** .5)

    input_files = [f'tile_{d["tile_x"]}-{d["tile_y"]}.ppm' for d in tile_vars]

    return htcondor.Submit(
        executable = '/usr/bin/montage',
        arguments = f'{" ".join(input_files)} -mode Concatenate -tile {num_tiles_per_
↪side}x{num_tiles_per_side} mandelbrot.png',
        transfer_input_files = ', '.join(input_files),
        log = 'mandelbrot.log',
        output = 'montage.out',
        error = 'montage.err',
        request_cpus = '1',
        request_memory = '128MB',
        request_disk = '1GB',
    )
```

[9]:
```
montage_description = make_montage_description(make_tile_vars(2))

print(montage_description)
```

```
executable = /usr/bin/montage
arguments = tile_00000-00000.ppm tile_00001-00000.ppm tile_00000-00001.ppm tile_00001-
↪00001.ppm -mode Concatenate -tile 2x2 mandelbrot.png
transfer_input_files = tile_00000-00000.ppm, tile_00001-00000.ppm, tile_00000-00001.ppm,␣
↪tile_00001-00001.ppm
log = mandelbrot.log
output = montage.out
error = montage.err
request_cpus = 1
request_memory = 128MB
request_disk = 1GB
```

Describing the DAG using `htcondor.dags`

Now that we have the job descriptions, all we have to do is use `htcondor.dags` to tell DAGMan about the dependencies between them. `htcondor.dags` is a subpackage of the HTCondor Python bindings that lets you write DAG descriptions using a higher-level language than raw DAG description file syntax. Incidentally, it also lets you use Python to drive the creation process, increasing your flexibility.

Important Concept: the code from `dag = dags.DAG()` onwards only defines the **topology** (or **structure**) of the DAG. The `tile` layer can be flexibly grown or shrunk by adjusting the `tile_vars` without changing the topology, and this can be clearly expressed in the code. The `tile_vars` are driving the creation of the DAG. Try changing `num_tiles_per_side` to some other value!

[10]:
```
from htcondor import dags

num_tiles_per_side = 2

# create the tile vars early, since we need to pass them to multiple places later
tile_vars = make_tile_vars(num_tiles_per_side)

dag = dags.DAG()

# create the tile layer, passing in the submit description for a tile job and the tile␣
↪vars
tile_layer = dag.layer(
    name = 'tile',
    submit_description = tile_description,
    vars = tile_vars,
)

# create the montage "layer" (it only has one job in it, so no need for vars)
# note that the submit description is created "on the fly"!
montage_layer = tile_layer.child_layer(
    name = 'montage',
    submit_description = make_montage_description(tile_vars),
)
```

We can get a textual description of the DAG structure by calling the `describe` method:

[11]:
```
print(dag.describe())
```

```
Type     Name    # Nodes  # Children  Parents
Layer    tile       4          1
Layer  montage      1          0       tile[ManyToMany]
```

Write the DAG to disk

We still need to write the DAG to disk to get DAGMan to work with it. We also need to move some files around so that the jobs know where to find them.

[12]:
```
from pathlib import Path
import shutil

dag_dir = (Path.cwd() / 'mandelbrot-dag').absolute()
```

(continues on next page)

(continued from previous page)

```
# blow away any old files
shutil.rmtree(dag_dir, ignore_errors = True)

# make the magic happen!
dag_file = dags.write_dag(dag, dag_dir)

# the submit files are expecting goatbrot to be next to them, so copy it into the dag␣
↪directory
shutil.copy2('goatbrot', dag_dir)

print(f'DAG directory: {dag_dir}')
print(f'DAG description file: {dag_file}')
```

```
DAG directory: /home/jovyan/tutorials/mandelbrot-dag
DAG description file: /home/jovyan/tutorials/mandelbrot-dag/dagfile.dag
```

Submit the DAG via the Python bindings

Now that we have written out the DAG description file, we can submit it for execution using the standard Python bindings submit mechanism. The `Submit` class has a static method which can read a DAG description and generate a corresponding `Submit` object:

[13]:
```
dag_submit = htcondor.Submit.from_dag(str(dag_file), {'force': 1})

print(dag_submit)
```

```
universe = scheduler
executable = /usr/bin/condor_dagman
getenv = True
output = /home/jovyan/tutorials/mandelbrot-dag/dagfile.dag.lib.out
error = /home/jovyan/tutorials/mandelbrot-dag/dagfile.dag.lib.err
log = /home/jovyan/tutorials/mandelbrot-dag/dagfile.dag.dagman.log
remove_kill_sig = SIGUSR1
MY.OtherJobRemoveRequirements = "DAGManJobId =?= $(cluster)"
on_exit_remove = (ExitSignal =?= 11 || (ExitCode =!= UNDEFINED && ExitCode >=0 &&␣
↪ExitCode <= 2))
arguments = "-p 0 -f -l . -Lockfile /home/jovyan/tutorials/mandelbrot-dag/dagfile.dag.
↪lock -AutoRescue 1 -DoRescueFrom 0 -Dag /home/jovyan/tutorials/mandelbrot-dag/dagfile.
↪dag -Suppress_notification -CsdVersion $CondorVersion:' '9.1.3' 'Aug' '19' '2021'
↪'BuildID:' 'UW_Python_Wheel_Build' '$ -Dagman /usr/bin/condor_dagman"
environment = _CONDOR_MAX_DAGMAN_LOG=0;_CONDOR_DAGMAN_LOG=/home/jovyan/tutorials/
↪mandelbrot-dag/dagfile.dag.dagman.out
```

Now we can enter the DAG directory and submit the DAGMan job, which will execute the graph:

[14]:
```
import os
os.chdir(dag_dir)

schedd = htcondor.Schedd()
with schedd.transaction() as txn:
```

(continues on next page)

(continued from previous page)

```
    cluster_id = dag_submit.queue(txn)

print(f"DAGMan job cluster is {cluster_id}")

os.chdir('..')
```

```
DAGMan job cluster is 6
```

Let's wait for the DAGMan job to complete by reading it's event log:

[15]:
```
dag_job_log = f"{dag_file}.dagman.log"
print(f"DAG job log file is {dag_job_log}")
```

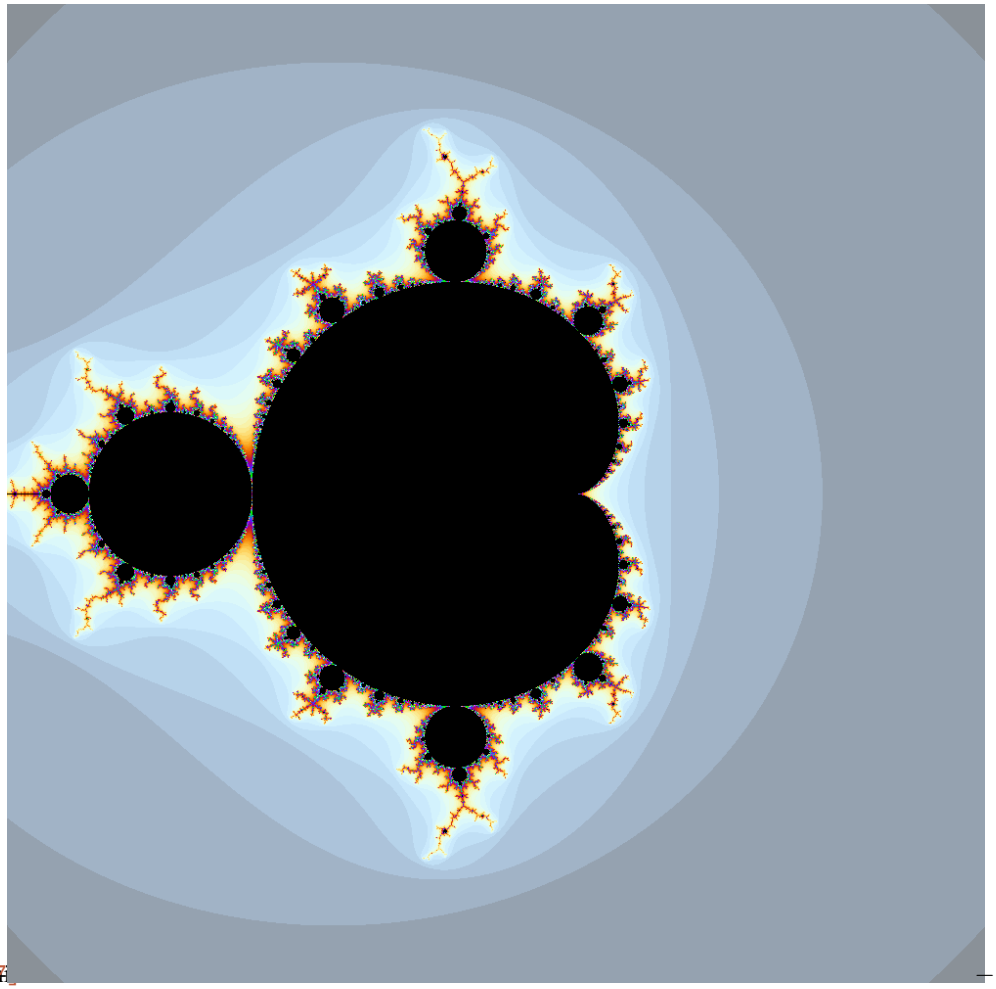
```
DAG job log file is /home/jovyan/tutorials/mandelbrot-dag/dagfile.dag.dagman.log
```

[16]:
```
# read events from the log, waiting forever for the next event
dagman_job_events = htcondor.JobEventLog(str(dag_job_log)).events(None)

# this event stream only contains the events for the DAGMan job itself, not the jobs it␣
↪submits
for event in dagman_job_events:
    print(event)

    # stop waiting when we see the terminate event
    if event.type is htcondor.JobEventType.JOB_TERMINATED and event.cluster == cluster_
↪id:
        break
```

```
000 (006.000.000) 2021-09-16 13:14:29 Job submitted from host: <172.17.0.2:9618?
↪addrs=172.17.0.2-9618&alias=abae0fbbde81&noUDP&sock=schedd_19_eccb>

001 (006.000.000) 2021-09-16 13:14:32 Job executing on host: <172.17.0.2:9618?addrs=172.
↪17.0.2-9618&alias=abae0fbbde81&noUDP&sock=schedd_19_eccb>

005 (006.000.000) 2021-09-16 13:15:10 Job terminated.
        (1) Normal termination (return value 0)
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
        0  -  Run Bytes Sent By Job
        0  -  Run Bytes Received By Job
        0  -  Total Bytes Sent By Job
        0  -  Total Bytes Received By Job
```

Let's look at the final image!

[17]:
```
Image(dag_dir / "mandelbrot.png")
```



nbsphinx-code-border

```
[ ]:
```

Personal Pools

Launch this tutorial in a Jupyter Notebook on Binder:

A Personal HTCondor Pool is an HTCondor Pool that has a single owner, who is: - The pool's administrator. - The only submitter who is allowed to submit jobs to the pool. - The owner of all resources managed by the pool.

The HTCondor Python bindings provide a submodule, `htcondor.personal`, which allows you to manage personal pools from Python. Personal pools are useful for: - Utilizing local computational resources (i.e., all of the cores on a lab server). - Created an isolated testing/development environment for HTCondor workflows. - Serving as an entrypoint to other computational resources, like annexes or flocked pools (not yet implemented).

We can start a personal pool by instantiating a `PersonalPool`. This object represents the personal pool and lets us manage its "lifecycle": start up and shut down. We can also use the `PersonalPool` to interact with the HTCondor pool once it has been started up.

Each Personal Pool must have a unique "local directory", corresponding to the HTCondor configuration parameter `LOCAL_DIR`. For this tutorial, we'll put it in the current working directory so that it's easy to find.

> Advanced users can configure the personal pool using the `PersonalPool` constructor. See the documentation for details on the available options.

[1]:
```
import htcondor
from htcondor.personal import PersonalPool
from pathlib import Path
```

[2]:
```
pool = PersonalPool(local_dir = Path.cwd() / "personal-condor")
pool
```

[2]:
```
PersonalPool(local_dir=./personal-condor, state=INITIALIZED)
```

To tell the personal pool to start running, call the `start()` method:

[3]:
```
pool.start()
```

[3]:
```
PersonalPool(local_dir=./personal-condor, state=READY)
```

`start()` doesn't return until the personal pool is `READY`, which means that it can accept commands (e.g., job submission).

`Schedd` and `Collector` objects for the personal pool are available as properties on the `PersonalPool`:

[4]:
```
pool.schedd
```

[4]:
```
<htcondor.htcondor.Schedd at 0x7f2c08111ea0>
```

[5]:
```
pool.collector
```

[5]:
```
<htcondor.htcondor.Collector at 0x7f2c08197400>
```

For example, we can submit jobs using `pool.schedd`:

[6]:
```
sub = htcondor.Submit(
    executable = "/bin/sleep",
    arguments = "$(ProcID)s",
)

schedd = pool.schedd
submit_result = schedd.submit(sub, count=10)

print(f"ClusterID is {submit_result.cluster()}")
```

```
ClusterID is 2
```

And we can query for the state of those jobs:

[7]:
```
for ad in pool.schedd.query(
    constraint = f"ClusterID == {submit_result.cluster()}",
    projection = ["ClusterID", "ProcID", "JobStatus"]
):
    print(repr(ad))
```

```
[ ClusterID = 2; ProcID = 0; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 1; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 2; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 3; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 4; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 5; JobStatus = 1; ServerTime = 1631798161 ]
```

(continues on next page)

(continued from previous page)

```
[ ClusterID = 2; ProcID = 6; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 7; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 8; JobStatus = 1; ServerTime = 1631798161 ]
[ ClusterID = 2; ProcID = 9; JobStatus = 1; ServerTime = 1631798161 ]
```

We can use the collector to query the state of pool:

[8]:
```
# get 3 random ads from the daemons in the pool
for ad in pool.collector.query()[:3]:
    print(ad)
```

```
    [
        UpdateSequenceNumber = 1;
        TargetType = "none";
        AuthenticationMethod = "FAMILY";
        Name = "jovyan@abae0fbbde81";
        AccountingGroup = "<none>";
        WeightedUnchargedTime = 0.0;
        DaemonStartTime = 1631798156;
        WeightedResourcesUsed = 2.000000000000000E+00;
        LastHeardFrom = 1631798160;
        Priority = 5.000000000000000E+02;
        LastUpdate = 1631798160;
        SubmitterLimit = 2.000000000000000E+00;
        MyType = "Accounting";
        PriorityFactor = 1.000000000000000E+03;
        IsAccountingGroup = false;
        Ceiling = -1;
        ResourcesUsed = 1;
        DaemonLastReconfigTime = 1631798156;
        AuthenticatedIdentity = "condor@family";
        NegotiatorName = "jovyan@abae0fbbde81";
        UnchargedTime = 0;
        SubmitterShare = 1.000000000000000E+00
    ]

    [
        AuthenticatedIdentity = "condor@family";
        EffectiveQuota = 0.0;
        GroupSortKey = 0.0;
        ResourcesUsed = 1;
        PriorityFactor = 1.000000000000000E+03;
        NegotiatorName = "jovyan@abae0fbbde81";
        Name = "<none>";
        AccumulatedUsage = 0.0;
        ConfigQuota = 0.0;
        LastHeardFrom = 1631798160;
        SubtreeQuota = 0.0;
        DaemonStartTime = 1631798156;
        LastUsageTime = 0;
        SurplusPolicy = "byquota";
        TargetType = "none";
```

(continues on next page)

(continued from previous page)

```
        AuthenticationMethod = "FAMILY";
        LastUpdate = 1631798160;
        WeightedAccumulatedUsage = 0.0;
        Priority = 5.000000000000000E+02;
        MyType = "Accounting";
        IsAccountingGroup = true;
        BeginUsageTime = 0;
        AccountingGroup = "<none>";
        UpdateSequenceNumber = 3;
        DaemonLastReconfigTime = 1631798156;
        WeightedResourcesUsed = 2.000000000000000E+00;
        Requested = 0.0
    ]

    [
        CCBReconnects = 0;
        MachineAdsPeak = 0;
        DetectedCpus = 2;
        UpdatesInitial_Accouting = 1;
        CurrentJobsRunningLinda = 0;
        StatsLifetime = 1;
        MaxJobsRunningAll = 0;
        CondorPlatform = "$CondorPlatform: X86_64-Ubuntu_20.04 $";
        MaxJobsRunningJava = 0;
        MaxJobsRunningGrid = 0;
        MaxJobsRunningPVMD = 0;
        RecentUpdatesLostMax = 0;
        UpdatesLost = 0;
        RecentUpdatesLostRatio = 0.0;
        MonitorSelfRegisteredSocketCount = 2;
        UpdatesTotal_Collector = 1;
        MonitorSelfTime = 1631798156;
        RecentUpdatesTotal_Collector = 1;
        CondorAdmin = "root@abae0fbbde81";
        MaxJobsRunningLinda = 0;
        CurrentJobsRunningPVM = 0;
        UpdatesLost_Collector = 0;
        CCBRequests = 0;
        CurrentJobsRunningPipe = 0;
        RecentUpdatesLost_Negotiator = 0;
        RecentUpdatesTotal = 3;
        RecentCCBRequestsFailed = 0;
        MaxJobsRunningVM = 0;
        CCBEndpointsConnected = 0;
        UpdatesLost_Accouting = 0;
        CurrentJobsRunningScheduler = 0;
        CurrentJobsRunningVanilla = 0;
        IdleJobs = 0;
        RecentUpdatesInitial_Accouting = 1;
        PendingQueriesPeak = 0;
        RecentUpdatesLost_Accouting = 0;
        ActiveQueryWorkersPeak = 2;
```

(continues on next page)

(continued from previous page)

```
        MonitorSelfAge = 1;
        MonitorSelfCPUUsage = 1.800000000000000E+01;
        PendingQueries = 0;
        ActiveQueryWorkers = 0;
        DetectedMemory = 1988;
        CurrentJobsRunningMPI = 0;
        UpdateInterval = 21600;
        CurrentJobsRunningPVMD = 0;
        DroppedQueries = 0;
        RecentCCBRequestsSucceeded = 0;
        CCBEndpointsConnectedPeak = 0;
        StatsLastUpdateTime = 1631798157;
        CondorVersion = "$CondorVersion: 8.9.11 Dec 29 2020 BuildID: Debian-8.9.11-1.2␣
↪PackageID: 8.9.11-1.2 Debian-8.9.11-1.2 $";
        MaxJobsRunningPipe = 0;
        CurrentJobsRunningParallel = 0;
        CCBEndpointsRegisteredPeak = 0;
        UpdatesInitial_Collector = 1;
        RecentDaemonCoreDutyCycle = 3.488135394901704E-02;
        SubmitterAdsPeak = 0;
        RecentUpdatesTotal_Accouting = 1;
        DaemonCoreDutyCycle = 3.488135394901704E-02;
        UpdatesTotal_Accouting = 1;
        MaxJobsRunningParallel = 0;
        UpdatesTotal = 3;
        RecentStatsLifetime = 1;
        MonitorSelfSecuritySessions = 2;
        CCBEndpointsRegistered = 0;
        LastHeardFrom = 1631798157;
        ForkQueriesFromCOLLECTOR = 2;
        HostsTotal = 0;
        CurrentJobsRunningJava = 0;
        RecentUpdatesTotal_Negotiator = 1;
        RecentForkQueriesFromCOLLECTOR = 2;
        CurrentJobsRunningAll = 0;
        RecentCCBRequestsNotFound = 0;
        Name = "My Pool - 127.0.0.1@abae0fbbde81";
        HostsOwner = 0;
        TargetType = "";
        CCBRequestsNotFound = 0;
        CurrentJobsRunningStandard = 0;
        SubmitterAds = 0;
        UpdatesLost_Negotiator = 0;
        MonitorSelfResidentSetSize = 11084;
        CCBRequestsSucceeded = 0;
        RecentUpdatesLost_Collector = 0;
        RecentUpdatesInitial_Collector = 1;
        RecentUpdatesLost = 0;
        RecentCCBRequests = 0;
        UpdatesTotal_Negotiator = 1;
        UpdatesInitial_Negotiator = 1;
        RecentDroppedQueries = 0;
```

(continues on next page)

(continued from previous page)

```
        CurrentJobsRunningUnknown = 0;
        RecentUpdatesInitial_Negotiator = 1;
        HostsUnclaimed = 0;
        MachineAds = 0;
        RecentCCBReconnects = 0;
        UpdatesLostMax = 0;
        CollectorIpAddr = "<172.17.0.2:46143?addrs=172.17.0.2-46143&alias=abae0fbbde81&
↪noUDP&sock=collector>";
        UpdatesInitial = 3;
        HostsClaimed = 0;
        MaxJobsRunningLocal = 0;
        AddressV1 = "{[ p=\"primary\"; a=\"172.17.0.2\"; port=46143; n=\"Internet\";␣
↪alias=\"abae0fbbde81\"; spid=\"collector\"; noUDP=true; ], [ p=\"IPv4\"; a=\"172.17.0.
↪2\"; port=46143; n=\"Internet\"; alias=\"abae0fbbde81\"; spid=\"collector\";␣
↪noUDP=true; ]}";
        MaxJobsRunningUnknown = 0;
        MyAddress = "<172.17.0.2:46143?addrs=172.17.0.2-46143&alias=abae0fbbde81&noUDP&
↪sock=collector>";
        Machine = "abae0fbbde81";
        CurrentJobsRunningGrid = 0;
        RunningJobs = 0;
        MyType = "Collector";
        MaxJobsRunningMPI = 0;
        MaxJobsRunningScheduler = 0;
        MyCurrentTime = 1631798156;
        RecentUpdatesInitial = 3;
        UpdatesLostRatio = 0.0;
        MaxJobsRunningVanilla = 0;
        CurrentJobsRunningLocal = 0;
        CCBRequestsFailed = 0;
        CurrentJobsRunningVM = 0;
        MaxJobsRunningStandard = 0;
        MonitorSelfImageSize = 16224;
        MaxJobsRunningPVM = 0
    ]
```

When you're done using the personal pool, you can `stop()` it:

```
[9]: pool.stop()
```

```
[9]: PersonalPool(local_dir=./personal-condor, state=STOPPED)
```

`stop()`, like `start()` will not return until the personal pool has actually stopped running. The personal pool will also automatically be stopped if the `PersonalPool` object is garbage-collected, or when the Python interpreter stops running.

> To prevent the pool from being automatically stopped in these situations, call the `detach()` method. The corresponding `attach()` method can be used to "re-connect" to a detached personal pool.

When working with a personal pool in a script, you may want to use it as a context manager. This pool will automatically start and stop at the beginning and end of the context:

```
[10]: with PersonalPool(local_dir = Path.cwd() / "another-personal-condor") as pool:  # note:␣
↪no need to call start()
```

(continues on next page)

(continued from previous page)

```
    print(pool.get_config_val("LOCAL_DIR"))
```

```
/home/jovyan/tutorials/another-personal-condor
```

```
[ ]:
```

8.3 classad API Reference

This page is an exhaustive reference of the API exposed by the *classad* module. It is not meant to be a tutorial for new users but rather a helpful guide for those who already understand the basic usage of the module.

8.3.1 ClassAd Representation

ClassAds are individually represented by the *ClassAd* class. Their attribute are key-value pairs, as in a standard Python dictionary. The keys are strings, and the values may be either Python primitives corresponding to ClassAd data types (string, bool, etc.) or *ExprTree* objects, which correspond to un-evaluated ClassAd expressions.

class classad.**ClassAd**(*input*)

The *ClassAd* object is the Python representation of a ClassAd. Where possible, *ClassAd* attempts to mimic a Python `dict`. When attributes are referenced, they are converted to Python values if possible; otherwise, they are represented by a *ExprTree* object.

New *ClassAd* objects can be initialized via a string (which is parsed as an ad) or a dictionary-like object containing attribute-value pairs.

The *ClassAd* object is iterable (returning the attributes) and implements the dictionary protocol. The `items`, `keys`, `values`, `get`, `setdefault`, and `update` methods have the same semantics as a dictionary.

Note: Where possible, we recommend using the dedicated parsing functions (*parseOne()*, *parseNext()*, or *parseAds()*) instead of using the constructor.

Parameters
input (*str or dict*) – A string or dictionary which will be interpreted as a classad.

eval(*attr*) → object :

Evaluate an attribute to a Python object. The result will *not* be an *ExprTree* but rather an built-in type such as a string, integer, boolean, etc.

Parameters
attr (*str*) – Attribute to evaluate.

Returns
The Python object corresponding to the evaluated ClassAd attribute

Raises
ValueError – if unable to evaluate the object.

lookup(*attr*) → ExprTree :

Look up the *ExprTree* object associated with attribute.

No attempt will be made to convert to a Python object.

Parameters
attr (*str*) – Attribute to evaluate.

Returns
The *ExprTree* object referenced by `attr`.

printOld() → str :

Serialize the ClassAd in the old ClassAd format.

Returns
The 'old ClassAd' representation of the ad.

Return type
str

printJson(*arg1*) → str :

Serialize the ClassAd as a string in JSON format.

Returns
The JSON representation of the ad.

Return type
str

flatten(*expr*) → object :

Given ExprTree object expression, perform a partial evaluation. All the attributes in expression and defined in this ad are evaluated and expanded. Any constant expressions, such as `1 + 2`, are evaluated; undefined attributes are not evaluated.

Parameters
expr (*ExprTree*) – The expression to evaluate in the context of this ad.

Returns
The partially-evaluated expression.

Return type
ExprTree

matches(*ad*) → bool :

Lookup the `Requirements` attribute of given `ad` return `True` if the `Requirements` evaluate to `True` in our context.

Parameters
ad (*ClassAd*) – ClassAd whose `Requirements` we will evaluate.

Returns
`True` if we satisfy `ad`'s requirements; `False` otherwise.

Return type
bool

symmetricMatch(*ad*) → bool :

Check for two-way matching between given ad and ourselves.

Equivalent to `self.matches(ad) and ad.matches(self)`.

Parameters
ad (*ClassAd*) – ClassAd to check for matching.

Returns
`True` if both ads' requirements are satisfied.

Return type
bool

externalRefs(*expr*) → list :

Returns a Python list of external references found in `expr`.

An external reference is any attribute in the expression which *is not* defined by the ClassAd object.

Parameters
expr (*ExprTree*) – Expression to examine.

Returns
A list of external attribute references.

Return type
list[str]

internalRefs(*expr*) → list :

Returns a Python list of internal references found in `expr`.

An internal reference is any attribute in the expression which *is* defined by the ClassAd object.

Parameters
expr (*ExprTree*) – Expression to examine.

Returns
A list of internal attribute references.

Return type
list[str]

__eq__(*arg1*, *arg2*) → bool :

One ClassAd is equivalent to another if they have the same number of attributes, and each attribute is the *sameAs()* the other.

__ne__(*arg1*, *arg2*) → bool :

The opposite of *__eq__()*.

class classad.**ExprTree**(*expr*)

The *ExprTree* class represents an expression in the ClassAd language.

The *ExprTree* constructor takes an ExprTree, or a string, which it will attempt to parse into a ClassAd expression. `str(expr)` will turn the `ExprTree` back into its string representation. `int`, `float`, and `bool` behave similarly, evaluating as necessary.

As with typical ClassAd semantics, lazy-evaluation is used. So, the expression `'foo' + 1` does not produce an error until it is evaluated with a call to `bool()` or the *ExprTree.eval()* method.

Note: The Python operators for *ExprTree* have been overloaded so, if `e1` and `e2` are *ExprTree* objects, then `e1 + e2` is also an *ExprTree* object. However, Python short-circuit evaluation semantics for `e1 && e2` cause `e1` to be evaluated. In order to get the 'logical and' of the two expressions *without* evaluating, use `e1.and_(e2)`. Similarly, `e1.or_(e2)` results in the 'logical or'.

---

and_(*expr*) → ExprTree :

Return a new expression, formed by `self && expr`.

Parameters
expr (*ExprTree*) – Right-hand-side expression to 'and'

Returns
A new expression, defined to be `self && expr`.

Return type
ExprTree

or_(*expr*) → ExprTree :

Return a new expression, formed by `self || expr`.

Parameters
expr (*ExprTree*) – Right-hand-side expression to 'or'

Returns
A new expression, defined to be `self || expr`.

Return type
ExprTree

is_(*expr*) → ExprTree :

Logical comparison using the 'meta-equals' operator.

Parameters
expr (*ExprTree*) – Right-hand-side expression to =?= operator.

Returns
A new expression, formed by `self =?= expr`.

Return type
ExprTree

isnt_(*expr*) → ExprTree :

Logical comparison using the 'meta-not-equals' operator.

Parameters
expr (*ExprTree*) – Right-hand-side expression to =!= operator.

Returns
A new expression, formed by `self =!= expr`.

Return type
ExprTree

sameAs(*expr*) → bool :

Returns `True` if given *ExprTree* is same as this one.

Parameters
expr (*ExprTree*) – Expression to compare against.

Returns
`True` if and only if `expr` is equivalent to this object.

Return type
bool

eval(*scope*) → object :

Evaluate the expression and return as a ClassAd value, typically a Python object.

> **Warning:** If `scope` is passed and is not the *ClassAd* this *ExprTree* might belong to, this method is not thread-safe.

Parameters
scope (*ClassAd*) – Optionally, the *ClassAd* context in which to evaluate. Unnecessary if the *ExprTree* comes from its own *ClassAd*, in which case it will be evaluated in the scope of that ad, or if the *ExprTree* can be evaluated without a context.

If passed, `scope` must be a *classad.ClassAd*.

Returns
The evaluated expression as a Python object.

simplify(*scope*, *target*) → ExprTree :
Evaluate the expression and return as a *ExprTree*.

Warning: If `scope` is passed and is not the *ClassAd* this *ExprTree* might belong to, this method is not thread-safe.

Warning: It is erroneous for `scope` to be a temporary; the lifetime of the returned object may depend on the lifetime of the scope object.

Parameters
- **scope** (*ClassAd*) – Optionally, the *ClassAd* context in which to evaluate. Unnecessary if the *ExprTree* comes from its own *ClassAd*, in which case it will be evaluated in the scope of that ad, or if the *ExprTree* can be evaluated without a context.

  If passed, `scope` must be a *classad.ClassAd*.
- **target** (*ClassAd*) – Optionally, the *ClassAd* TARGET ad.

  If passed, `target` must be a `classAd.ClassAd`.

Returns
The evaluated expression as an *ExprTree*.

class classad.**Value**
An enumeration of the two special ClassAd values `Undefined` and `Error`.

The values of the enumeration are:

Undefined

Error

8.3.2 Parsing and Creating ClassAds

classad provides a variety of utility functions that can help you construct ClassAd expressions and parse string representations of ClassAds.

classad.**parseAds**(*input*, *parser=classad.classad.Parser.Auto*) → object :
Parse the input as a series of ClassAds.

Parameters
- **input** (*str or file*) – Serialized ClassAd input; may be a file-like object.
- **parser** (*Parser*) – Controls behavior of the ClassAd parser.

Returns
An iterator that produces *ClassAd*.

classad.**parseNext**(*input*, *parser=classad.classad.Parser.Auto*) → object :

Parse the next ClassAd in the input string. Advances the **input** to point after the consumed ClassAd.

Parameters

- **input** (*str or file*) – Serialized ClassAd input; may be a file-like object.
- **parser** (*Parser*) – Controls behavior of the ClassAd parser.

Returns
An iterator that produces *ClassAd*.

classad.**parseOne**(*input*, *parser=classad.classad.Parser.Auto*) → ClassAd :

Parse the entire input into a single *ClassAd* object.

In the presence of multiple ClassAds or blank lines in the input, continue to merge ClassAds together until the entire input is consumed.

Parameters

- **input** (*str or file*) – Serialized ClassAd input; may be a file-like object.
- **parser** (*Parser*) – Controls behavior of the ClassAd parser.

Returns
Corresponding *ClassAd* object.

Return type
ClassAd

classad.**quote**(*input*) → str :

Converts the Python string into a ClassAd string literal; this handles all the quoting rules for the ClassAd language. For example:

```
>>> classad.quote('hello'world')
''hello\\'world''
```

This allows one to safely handle user-provided strings to build expressions. For example:

```
>>> classad.ExprTree('Foo =?= %s' % classad.quote('hello'world'))
Foo is 'hello\'world'
```

Parameters
input (*str*) – Input string to quote.

Returns
The corresponding string literal as a Python string.

Return type
str

classad.**unquote**(*input*) → str :

Converts a ClassAd string literal, formatted as a string, back into a Python string. This handles all the quoting rules for the ClassAd language.

Parameters
input (*str*) – Input string to unquote.

Returns
The corresponding Python string for a string literal.

Return type
str

classad.**Attribute**(*name*) → ExprTree :

Given an attribute name, construct an *ExprTree* object which is a reference to that attribute.

Note: This may be used to build ClassAd expressions easily from python. For example, the ClassAd expression `foo == 1` can be constructed by the Python code `Attribute('foo') == 1`.

Parameters
name (*str*) – Name of attribute to reference.

Returns
Corresponding expression consisting of an attribute reference.

Return type
ExprTree

classad.**Function**()

Given function name `name`, and zero-or-more arguments, construct an *ExprTree* which is a function call expression. The function is not evaluated.

For example, the ClassAd expression `strcat('hello ', 'world')` can be constructed by the Python expression `Function('strcat', 'hello ', 'world')`.

Returns
Corresponding expression consisting of a function call.

Return type
ExprTree

classad.**Literal**(*obj*) → ExprTree :

Convert a given Python object to a ClassAd literal.

Python strings, floats, integers, and booleans have equivalent literals in the ClassAd language.

Parameters
obj – Python object to convert to an expression.

Returns
Corresponding expression consising of a literal.

Return type
ExprTree

classad.**lastError**() → str :

Return the string representation of the last error to occur in the ClassAd library.

As the ClassAd language has no concept of an exception, this is the only mechanism to receive detailed error messages from functions.

classad.**register**(*function*, *name=None*) → None :

Given the Python function, register it as a function in the ClassAd language. This allows the invocation of the Python function from within a ClassAd evaluation context.

Parameters

- **function** – A callable object to register with the ClassAd runtime.
- **name** (*str*) – Provides an alternate name for the function within the ClassAd library. The default, None, indicates to use the built-in function name.

classad.**registerLibrary**(*arg1*) → None :

Given a file system path, attempt to load it as a shared library of ClassAd functions. See the upstream documentation for configuration variable CLASSAD_USER_LIBS for more information about loadable libraries for ClassAd functions.

Parameters
path (*str*) – The library to load.

8.3.3 Parser Control

The behavior of *parseAds()*, *parseNext()*, and *parseOne()* can be controlled by giving them different values of the *Parser* enumeration.

class classad.**Parser**

An enumeration that controls the behavior of the ClassAd parser. The values of the enumeration are…

Auto

The parser should automatically determine the ClassAd representation.

Old

The parser should only accept the old ClassAd format.

New

The parser should only accept the new ClassAd format.

8.3.4 Utility Functions

classad.**version**() → str :

Return the version of the linked ClassAd library.

8.3.5 Exceptions

For backwards-compatibility, the exceptions in this module inherit from the built-in exceptions raised in earlier (pre-v8.9.9) versions.

class classad.**ClassAdException**

Never raised. The parent class of all exceptions raised by this module.

class classad.**ClassAdEnumError**

Raised when a value must be in an enumeration, but isn't.

class classad.**ClassAdEvaluationError**

Raised when the ClassAd library fails to evaluate an expression.

class classad.**ClassAdInternalError**

Raised when the ClassAd library encounters an internal error.

class classad.**ClassAdOSError**

Raised instead of OSError for backwards compatibility.

class classad.**ClassAdParseError**

Raised when the ClassAd library fails to parse a (putative) ClassAd.

class classad.**ClassAdTypeError**

Raised instead of `TypeError` for backwards compatibility.

class classad.**ClassAdValueError**

Raised instead of `ValueError` for backwards compatibility.

8.3.6 Deprecated Functions

The functions in this section are deprecated; new code should not use them and existing code should be rewritten to use their replacements.

classad.**parse**(*input*) → ClassAd :

> **Warning:** This function is deprecated.

Parse input, in the new ClassAd format, into a *ClassAd* object.

Parameters
input (*str or file*) – A string-like object or a file pointer.

Returns
Corresponding *ClassAd* object.

Return type
ClassAd

classad.**parseOld**(*input*) → ClassAd :

> **Warning:** This function is deprecated.

Parse input, in the old ClassAd format, into a *ClassAd* object.

Parameters
input (*str or file*) – A string-like object or a file pointer.

Returns
Corresponding *ClassAd* object.

Return type
ClassAd

8.4 htcondor API Reference

This page is an exhaustive reference of the API exposed by the *htcondor* module. It is not meant to be a tutorial for new users but rather a helpful guide for those who already understand the basic usage of the module.

8.4.1 Interacting with Collectors

class htcondor.**Collector**(*pool*)

Client object for a remote *condor_collector*. The *Collector* can be used to:

- Locate a daemon.
- Query the *condor_collector* for one or more specific ClassAds.
- Advertise a new ad to the *condor_collector*.

Parameters
pool (*str or list[str]*) – A `host:port` pair specified for the remote collector (or a list of pairs for HA setups). If omitted, the value of configuration parameter `COLLECTOR_HOST` is used.

locate(*daemon_type*, *name*) → object :

Query the *condor_collector* for a particular daemon.

Parameters
- **daemon_type** (*DaemonTypes*) – The type of daemon to locate.
- **name** (*str*) – The name of daemon to locate. If not specified, it searches for the local daemon.

Returns
a minimal ClassAd of the requested daemon, sufficient only to contact the daemon; typically, this limits to the `MyAddress` attribute.

Return type
ClassAd

locateAll(*daemon_type*) → object :

Query the condor_collector daemon for all ClassAds of a particular type. Returns a list of matching ClassAds.

Parameters
daemon_type (*DaemonTypes*) – The type of daemon to locate.

Returns
Matching ClassAds

Return type
list[*ClassAd*]

query(*ad_type=htcondor.htcondor.AdTypes.Any*, *constraint=''*, *projection=[]*, *statistics=''*) → object :

Query the contents of a condor_collector daemon. Returns a list of ClassAds that match the constraint parameter.

Parameters
- **ad_type** (*AdTypes*) – The type of ClassAd to return. If not specified, the type will be ANY_AD.
- **constraint** (str or *ExprTree*) – A constraint for the collector query; only ads matching this constraint are returned. If not specified, all matching ads of the given type are returned.
- **projection** (*list[str]*) – A list of attributes to use for the projection. Only these attributes, plus a few server-managed, are returned in each *ClassAd*.
- **statistics** (*list[str]*) – Statistics attributes to include, if they exist for the specified daemon.

Returns
A list of matching ads.

Return type
list[*ClassAd*]

directQuery(*daemon_type*, *name=''*, *projection=[]*, *statistics=''*) → object :

Query the specified daemon directly for a ClassAd, instead of using the ClassAd from the *condor_collector* daemon. Requires the client library to first locate the daemon in the collector, then querying the remote daemon.

Parameters

- **daemon_type** (*DaemonTypes*) – Specifies the type of the remote daemon to query.
- **name** (*str*) – Specifies the daemon's name. If not specified, the local daemon is used.
- **projection** (*list[str]*) – is a list of attributes requested, to obtain only a subset of the attributes from the daemon's *ClassAd*.
- **statistics** (*str*) – Statistics attributes to include, if they exist for the specified daemon.

Returns
The ad of the specified daemon.

Return type
ClassAd

advertise(*ad_list*, *command='UPDATE_AD_GENERIC'*, *use_tcp=True*) → None :

Advertise a list of ClassAds into the condor_collector.

Parameters

- **ad_list** (list[ClassAds]) – ClassAds to advertise.
- **command** (*str*) – An advertise command for the remote *condor_collector*. It defaults to UPDATE_AD_GENERIC. Other commands, such as UPDATE_STARTD_AD, may require different authorization levels with the remote daemon.
- **use_tcp** (*bool*) – When set to True, updates are sent via TCP. Defaults to True.

class htcondor.**DaemonTypes**

An enumeration of different types of daemons available to HTCondor.

The values of the enumeration are:

None

Any
Any type of daemon; useful when specifying queries where all matching daemons should be returned.

Master
Ads representing the *condor_master*.

Schedd
Ads representing the *condor_schedd*.

Startd
Ads representing the resources on a worker node.

Collector
Ads representing the *condor_collector*.

Negotiator
Ads representing the *condor_negotiator*.

HAD
Ads representing the high-availability daemons (*condor_had*).

Generic
All other ads that are not categorized as above.

Credd

class htcondor.**AdTypes**
A list of different types of ads that may be kept in the *condor_collector*.

The values of the enumeration are:

None

Any
Type representing any matching ad. Useful for queries that match everything in the collector.

Generic
Generic ads, associated with no particular daemon.

Startd
Startd ads, produced by the *condor_startd* daemon. Represents the available slots managed by the startd.

StartdPrivate
The "private" ads, containing the claim IDs associated with a particular slot. These require additional authorization to read as the claim ID may be used to run jobs on the slot.

Schedd
Schedd ads, produced by the *condor_schedd* daemon.

Master
Master ads, produced by the *condor_master* daemon.

Collector
Ads from the *condor_collector* daemon.

Negotiator
Negotiator ads, produced by the *condor_negotiator* daemon.

Submitter
Ads describing the submitters with available jobs to run; produced by the *condor_schedd* and read by the *condor_negotiator* to determine which users need a new negotiation cycle.

Grid
Ads associated with the grid universe.

HAD
Ads produced by the *condor_had*.

License
License ads. These do not appear to be used by any modern HTCondor daemon.

Credd

Defrag

Accounting

8.4.2 Interacting with Schedulers

class htcondor.**Schedd**(*location_ad*)

Client object for a *condor_schedd*.

Parameters

location_ad (*ClassAd* or DaemonLocation) – A *ClassAd* describing the location of the remote *condor_schedd* daemon, as returned by the *Collector.locate()* method, or a tuple of type DaemonLocation as returned by Schedd.location(). If the parameter is omitted, the local *condor_schedd* daemon is used.

transaction()

> **Warning:** Schedd.transaction() was deprecated in version 10.7.0 and will be removed in a future release. Use Schedd.submit() instead.

transaction((Schedd)self [, (TransactionFlags)flags=0 [, (bool)continue_txn=False]]) -> Transaction :

This method is DEPRECATED. Use *Schedd.submit()* instead.

Start a transaction with the *condor_schedd*.

Starting a new transaction while one is ongoing is an error unless the continue_txn flag is set.

param flags
Flags controlling the behavior of the transaction, defaulting to 0.

type flags
TransactionFlags

param bool continue_txn
Set to True if you would like this transaction to extend any pre-existing transaction; defaults to False. If this is not set, starting a transaction inside a pre-existing transaction will cause an exception to be thrown.

return
A *Transaction* object.

query(*constraint='true'*, *projection=[]*, *callback=None*, *limit=-1*, *opts=htcondor.htcondor.QueryOpts.Default*) → object :

Query the *condor_schedd* daemon for job ads.

> **Warning:** This returns a *list* of *ClassAd* objects, meaning all results must be held in memory simultaneously. This may be memory-intensive for queries that return many and/or large jobs ads. If you are retrieving many large ads, consider using *xquery()* instead to reduce memory requirements.

Parameters

- **constraint** (str or *ExprTree*) – A query constraint. Only jobs matching this constraint will be returned. Defaults to 'true', which means all jobs will be returned.
- **projection** (*list[str]*) – Attributes that will be returned for each job in the query. At least the attributes in this list will be returned, but additional ones may be returned as well. An empty list (the default) returns all attributes.
- **callback** – A callable object; if provided, it will be invoked for each ClassAd. The return value (if not None) will be added to the returned list instead of the ad.

- **limit** (*int*) – The maximum number of ads to return; the default (-1) is to return all ads.
- **opts** (*QueryOpts*.) – Additional flags for the query; these may affect the behavior of the *condor_schedd*.

Returns
ClassAds representing the matching jobs.

Return type
list[*ClassAd*]

xquery()

Warning: Schedd.xquery() was deprecated in version 10.7.0 and will be removed in a future release.

xquery((Schedd)self [, (object)constraint='true' [, (list)projection=[] [, (int)limit=-1 [, (QueryOpts)opts=htcondor.htcondor.QueryOpts.Default [, (object)name=None]]]]]) -> QueryIterator :

Warning: This function is deprecated.

Query the *condor_schedd* daemon for job ads.

Warning: This returns an *iterator* of *ClassAd* objects, which means you may not need to hold all of the ads returned by the query in memory simultaneously. However, this method holds a connection open to the schedd, and a fork of the schedd will remain active, until you finish iterating. If you are **not** retrieving many large ads, consider using *query()* instead to reduce load on the schedd.

param constraint
A query constraint. Only jobs matching this constraint will be returned. Defaults to `'true'`, which means all jobs will be returned.

type constraint
str or *ExprTree*

param projection
Attributes that will be returned for each job in the query. At least the attributes in this list will be returned, but additional ones may be returned as well. An empty list (the default) returns all attributes.

type projection
list[str]

param int limit
A limit on the number of matches to return. The default (-1) indicates all matching jobs should be returned.

param opts
Additional flags for the query, from *QueryOpts*.

type opts
QueryOpts

param str name
A tag name for the returned query iterator. This string will always be returned from

the *QueryIterator.tag()* method of the returned iterator. The default value is the *condor_schedd*'s name. This tag is useful to identify different queries when using the *poll()* function.

return
An iterator for the matching job ads

rtype
QueryIterator

act(*action*, *job_spec*, *reason=None*) → object :

Change status of job(s) in the *condor_schedd* daemon. The return value is a ClassAd object describing the number of jobs changed.

This will throw an exception if no jobs are matched by the constraint.

Parameters

- **action** (*JobAction*) – The action to perform; must be of the enum JobAction.
- **job_spec** (*list[str] or str*) – The job specification. It can either be a list of job IDs, or an ExprTree or string specifying a constraint. Only jobs matching this description will be acted upon.
- **reason** (*str*) – The reason for the action. If omitted, the reason will be "Python-initiated action".

edit(*job_spec*, *attr*, *value*, *flags=0*) → EditResult :

Edit one or more jobs in the queue.

This will throw an exception if no jobs are matched by the `job_spec` constraint.

Parameters

- **job_spec** (*list[str] or str*) – The job specification. It can either be a list of job IDs or a string specifying a constraint. Only jobs matching this description will be acted upon.
- **attr** (*str*) – The name of the attribute to edit.
- **value** (str or *ExprTree*) – The new value of the attribute. It should be a string, which will be converted to a ClassAd expression, or an ExprTree object. Be mindful of quoting issues; to set the value to the string `foo`, one would set the value to `''foo''`
- **flags** (*TransactionFlags*) – Flags controlling the behavior of the transaction, defaulting to 0.

Returns
An EditResult containing the number of jobs that were edited.

Return type
`EditResult`

history(*constraint*, *projection*, *match=-1*, *since=None*) → HistoryIterator :

Fetch history records from the *condor_schedd* daemon.

Parameters

- **constraint** (str or *ExprTree*) – A query constraint. Only jobs matching this constraint will be returned. `None` will return all jobs.
- **projection** (*list[str]*) – Attributes that will be returned for each job in the query. At least the attributes in this list will be returned, but additional ones may be returned as well. An empty list returns all attributes.

- **match** (*int*) – A limit on the number of jobs to include; the default (`-1`) indicates to return all matching jobs. The schedd may return fewer than `match` jobs because of its setting of `HISTORY_HELPER_MAX_HISTORY` (default 10,000).
- **since** (int, str, or *ExprTree*) – A cluster ID, job ID, or expression. If a cluster ID (passed as an *int*) or job ID (passed a *str* in the format `{clusterID}.{procID}`), only jobs recorded in the history file after (and not including) the matching ID will be returned. If an expression (passed as a *str* or *ExprTree*), jobs will be returned, most-recently-recorded first, until the expression becomes true; the job making the expression become true will not be returned. Thus, `1038` and `clusterID == 1038` return the same set of jobs.

Returns
All matching ads in the Schedd history, with attributes according to the `projection` keyword.

Return type
HistoryIterator

jobEpochHistory(*constraint*, *projection*, *match=-1*, *since=None*) → HistoryIterator :

Fetch per job run instance (epoch) history records from the *condor_schedd* daemon.

Parameters

- **constraint** (str or *ExprTree*) – A query constraint. Only jobs matching this constraint will be returned. `None` will return all jobs.
- **projection** (*list[str]*) – Attributes that will be returned for each job in the query. At least the attributes in this list will be returned, but additional ones may be returned as well. An empty list returns all attributes.
- **match** (*int*) – A limit on the number of jobs to include; the default (`-1`) indicates to return all matching jobs. The schedd may return fewer than `match` jobs because of its setting of `HISTORY_HELPER_MAX_HISTORY` (default 10,000).
- **since** (int, str, or *ExprTree*) – A cluster ID, job ID, or expression. If a cluster ID (passed as an *int*) or job ID (passed a *str* in the format `{clusterID}.{procID}`), only jobs recorded in the history file after (and not including) the matching ID will be returned. If an expression (passed as a *str* or *ExprTree*), jobs will be returned, most-recently-recorded first, until the expression becomes true; the job making the expression become true will not be returned. Thus, `1038` and `clusterID == 1038` return the same set of jobs.

Returns
All matching ads in the Schedd history, with attributes according to the `projection` keyword.

Return type
HistoryIterator

submit(*description*, *count=1*, *spool=False*, *ad_results=None*, *itemdata=None*) → object :

Submit one or more jobs to the *condor_schedd* daemon.

This method requires the invoker to provide a *Submit* object that describes the jobs to submit. The return value will be a *SubmitResult* that contains the cluster ID and ClassAd of the submitted jobs.

For backward compatibility, this method will also accept a *ClassAd* that describes a single job to submit, but use of this form of is DEPRECATED. If the deprecated form is used the return value will be the cluster ID, and ad_results will optionally be the actual job ClassAds that were submitted.

Parameters

- **description** (*Submit* (or DEPRECATED `ClassAd`)) – The Submit description or ClassAd describing the job cluster.
- **count** (*int*) – The number of jobs to submit to the job cluster. Defaults to 1.

- **spool** (*bool*) – If `True`, jobs will be submitted in a spooling hold mode so that input files can be spooled to a remote *condor_schedd* daemon before starting the jobs. This parameter is necessary for jobs submitted to a remote *condor_schedd* that use HTCondor file transfer. When True, job will be left in the HOLD state until the *spool()* method is called.
- **ad_results** (list[*ClassAd*]) – DEPRECATED. If set to a list and a raw job ClassAd is passed as the first argument, the list object will contain the job ads that were submitted.

Returns
a *SubmitResult*, containing the cluster ID, cluster ClassAd and range of Job ids of the submitted job(s). If using the deprecated first argument, the return value will be an int and ad_results may contain submitted jobs ClassAds.

Return type
SubmitResult or int

submitMany(*cluster_ad*, *proc_ads*, *spool=False*, *ad_results=None*) → int :

Submit multiple jobs to the *condor_schedd* daemon, possibly including several distinct processes.

Parameters

- **cluster_ad** (*ClassAd*) – The base ad for the new job cluster; this is the same format as in the *submit()* method.
- **proc_ads** (*list*) – A list of 2-tuples; each tuple has the format of `(proc_ad, count)`. For each list entry, this will result in count jobs being submitted inheriting from both `cluster_ad` and `proc_ad`.
- **spool** (*bool*) – If `True`, the client inserts the necessary attributes into the job for it to have the input files spooled to a remote *condor_schedd* daemon. This parameter is necessary for jobs submitted to a remote *condor_schedd* that use HTCondor file transfer. When True, job will be left in the HOLD state until the *spool()* method is called.
- **ad_results** (list[*ClassAd*]) – If set to a list, the list object will contain the job ads resulting from the job submission.

Returns
The newly created cluster ID.

Return type
int

spool(*ad_list*) → None :

Spools the files specified in a list of job ClassAds to the *condor_schedd*.

Parameters
ad_list (list[`ClassAds`]) – A list of job descriptions; typically, this is the list returned by the `jobs()` method on the submit result object.

Raises
RuntimeError – if there are any errors.

retrieve(*arg1*, *arg2*) → None :

Retrieve the output sandbox from one or more jobs.

Parameters
job_spec (str or list[*ClassAd*]) – An expression matching the list of job output sandboxes to retrieve.

refreshGSIProxy(*cluster*, *proc*, *proxy_filename*, *lifetime*) → int :

Refresh the GSI proxy of a job; the job's proxy will be replaced the contents of the provided `proxy_filename`.

Note: Depending on the lifetime of the proxy in `proxy_filename`, the resulting lifetime may be shorter than the desired lifetime.

Parameters

- **cluster** (*int*) – Cluster ID of the job to alter.
- **proc** (*int*) – Process ID of the job to alter.
- **proxy_filename** (*str*) – The name of the file containing the new proxy for the job.
- **lifetime** (*int*) – Indicates the desired lifetime (in seconds) of the delegated proxy. A value of `0` specifies to not shorten the proxy lifetime. A value of `-1` specifies to use the value of configuration variable `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`.

reschedule() → None :

Send reschedule command to the schedd.

export_jobs(*job_spec*, *export_dir*, *new_spool_dir*) → object :

Export one or more job clusters from the queue to put those jobs into the externally managed state.

Parameters

- **job_spec** (*list[str] or str or* ExprTree) – The job specification. It can either be a list of job IDs or a string specifying a constraint. Only jobs matching this description will be acted upon.
- **export_dir** (*str*) – The path to the directory that exported jobs will be written into.
- **new_spool_dir** (*str*) – The path to the base directory that exported jobs will use as IWD while they are exported

Returns
A ClassAd containing information about the export operation.

Return type
ClassAd

import_exported_job_results(*import_dir*) → object :

Import results from previously exported jobs, and take those jobs back out of the externally managed state.

Parameters
import_dir (*str*) – The path to the modified form of a previously-exported directory.

Returns
A ClassAd containing information about the import operation.

Return type
ClassAd

unexport_jobs(*job_spec*) → object :

Unexport one or more job clusters that were previously exported from the queue.

Parameters
job_spec (*list[str] or str or* ExprTree) – The job specification. It can either be a

list of job IDs or a string specifying a constraint. Only jobs matching this description will be acted upon.

Returns
A ClassAd containing information about the unexport operation.

Return type
ClassAd

class htcondor.**JobAction**

An enumeration describing the actions that may be performed on a job in queue.

The values of the enumeration are:

Hold

Put a job on hold, vacating a running job if necessary. A job will stay in the hold state until explicitly acted upon by the admin or owner.

Release

Release a job from the hold state, returning it to `Idle`.

Suspend

Suspend the processes of a running job (on Unix platforms, this triggers a `SIGSTOP`). The job's processes stay in memory but no longer get scheduled on the CPU.

Continue

Continue a suspended jobs (on Unix, `SIGCONT`). The processes in a previously suspended job will be scheduled to get CPU time again.

Remove

Remove a job from the Schedd's queue, cleaning it up first on the remote host (if running). This requires the remote host to acknowledge it has successfully vacated the job, meaning `Remove` may not be instantaneous.

RemoveX

Immediately remove a job from the schedd queue, even if it means the job is left running on the remote resource.

Vacate

Cause a running job to be killed on the remote resource and return to idle state. With `Vacate`, jobs may be given significant time to cleanly shut down.

VacateFast

Vacate a running job as quickly as possible, without providing time for the job to cleanly terminate.

class htcondor.**Transaction**

> **Warning:** Transaction was deprecated in version 10.7.0 and will be removed in a future release.

DEPRECATED. An ongoing transaction in the HTCondor schedd.

class htcondor.**TransactionFlags**

Enumerated flags affecting the characteristics of a transaction.

The values of the enumeration are:

NonDurable

Non-durable transactions are changes that may be lost when the *condor_schedd* crashes. `NonDurable` is used for performance, as it eliminates extra `fsync()` calls.

SetDirty

This marks the changed ClassAds as dirty, causing an update notification to be sent to the *condor_shadow* and the *condor_gridmanager*, if they are managing the job.

ShouldLog

Causes any changes to the job queue to be logged in the relevant job event log.

class htcondor.**QueryOpts**

Enumerated flags sent to the *condor_schedd* during a query to alter its behavior.

The values of the enumeration are:

Default

Queries should use default behaviors, and return jobs for all users.

AutoCluster

Instead of returning job ads, return an ad per auto-cluster.

GroupBy

Instead of returning job ads, return an ad for each unique combination of values for the attributes in the projection. Similar to AutoCluster, but using the projection as the significant attributes for auto-clustering.

DefaultMyJobsOnly

Queries should use all default behaviors, and return jobs only for the current user.

SummaryOnly

Instead of returning job ads, return only the final summary ad.

IncludeClusterAd

Query should return raw cluster ads as well as job ads if the cluster ads match the query constraint.

class htcondor.**BlockingMode**

An enumeration that controls the behavior of query iterators once they are out of data.

The values of the enumeration are:

Blocking

Sets the iterator to block until more data is available.

NonBlocking

Sets the iterator to return immediately if additional data is not available.

class htcondor.**HistoryIterator**

An iterator over ads in the history produced by *Schedd.history()*.

class htcondor.**QueryIterator**

An iterator class for managing results of the *Schedd.query()* and *Schedd.xquery()* methods.

nextAdsNonBlocking() → list :

Retrieve as many ads are available to the iterator object.

If no ads are available, returns an empty list. Does not throw an exception if no ads are available or the iterator is finished.

Returns
Zero-or-more job ads.

Return type
list[*ClassAd*]

tag() → str :

Retrieve the tag associated with this iterator; when using the *poll()* method, this is useful to distinguish multiple iterators.

Returns
The query's tag.

done() → bool :

Returns
`True` if the iterator is finished; `False` otherwise.

Return type
bool

watch() → int :

Returns an `inotify`-based file descriptor; if this descriptor is given to a `select()` instance, `select` will indicate this file descriptor is ready to read whenever there are more jobs ready on the iterator.

If `inotify` is not available on this platform, this will return `-1`.

Returns
A file descriptor associated with this query.

Return type
int

htcondor.**poll**(*queries*, *timeout_ms=20000*) → BulkQueryIterator :

Wait on the results of multiple query iterators.

This function returns an iterator which yields the next ready query iterator. The returned iterator stops when all results have been consumed for all iterators.

Parameters
active_queries (list[*QueryIterator*]) – Query iterators as returned by xquery().

Returns
An iterator producing the ready *QueryIterator*.

Return type
BulkQueryIterator

class htcondor.**BulkQueryIterator**

Returned by *poll()*, this iterator produces a sequence of *QueryIterator* objects that have ads ready to be read in a non-blocking manner.

Once there are no additional available iterators, *poll()* must be called again.

class htcondor.**JobStatus**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

An enumeration of HTCondor job status values.

IDLE

RUNNING

REMOVED

COMPLETED

HELD

TRANSFERRING_OUTPUT

SUSPENDED

8.4.3 Submitting Jobs

class htcondor.**Submit**

An object representing a job submit description. It uses the same submit language as *condor_submit*.

The submit description contains `key = value` pairs and implements the python dictionary protocol, including the `get`, `setdefault`, `update`, `keys`, `items`, and `values` methods. Values in the submit discription language have no data type; they are all stored as strings.

object __init__(tuple args, dict kwds) :

param input

Submit descriptors as a string containing the text of a submit file or as `key = value` pairs in a dictionary, or as keyword arguments.

Only the single multi-line string form can contain a `QUEUE` statement.

For example, these calls all produce identical submit descriptions:

```
from_file = htcondor.Submit(
    """
    executable = /bin/sleep
    arguments = 5s
    log = $(ClusterId).log
    My.CustomAttribute = "foobar"
    """
)

# create an empty submit object, then populate it as a dict
# use of classad.quote here insures that the value is properly␣
↪escaped as a classad string
submit_dict = htcondor.Submit()
submit_dict["executable"] = "/bin/sleep"
submit_dict["arguments"] = "5s"
submit_dict["log"] = "$(ClusterId).log"
submit_dict["My.CustomAttribute"] = classad.quote("foobar")

# initialize a submit object from a python dict
# note that values should be strings
mydict = {
    "executable": "/bin/sleep",
    "arguments": "5s",
    "log": "$(ClusterId).log",
    "My.CustomAttribute": classad.quote("foobar"),
}
from_dict = htcondor.Submit(mydict)

# initialize a submit object from keyword arguments
# the **{} is a trick to get a keyword argument that contains a .
from_kwargs = htcondor.Submit(
    executable="/bin/sleep",
```

(continues on next page)

(continued from previous page)

```
    arguments="5s",
    log="$(ClusterId).log",
    **{ "My.CustomAttribute": classad.quote("foobar") }
)
```

If a string initalizer is used, it may include a single *condor_submit* `QUEUE` statement at the end. If omitted, the submit description is initially empty.

The arguments to the `QUEUE` statement will be stored in the `QArgs` member of this class and can be passed to `schedd.Submit()` as the itemdata iterator like this

```
sub = htcondor.Submit(
    """
    executable = /bin/sleep
    QUEUE arguments in (1s, 10s, 5m)
    """
)
schedd.Submit(sub, count=1, itemdata=sub.itemdata())
```

type input
dict or str

queue()

Warning: Submit.queue() was deprecated in version 10.7.0 and will be removed in a future release. Use Schedd.submit() instead.

queue((Submit)self, (Transaction)txn [, (int)count=0 [, (object)ad_results=None]]) -> int :

This method is DEPRECATED. Use *Schedd.submit()* instead.

Submit the current object to a remote queue.

param txn
An active transaction object (see *Schedd.transaction()*).

type txn
Transaction

param int count
The number of jobs to create (defaults to `0`). If not specified, or a value of `0` is given the `QArgs` member of this class is used to determine the number of procs to submit. If no `QArgs` were specified, one job is submitted.

param ad_results
A list to receive the ClassAd resulting from this submit. As with *Schedd.submit()*, this is often used to later spool the input files.

return
The ClusterID of the submitted job(s).

rtype
int

raises RuntimeError
if the submission fails.

queue_with_itemdata()

> **Warning:** Submit.queue_with_itemdata() was deprecated in version 10.7.0 and will be removed in a future release. Use Schedd.submit() instead.

queue_with_itemdata((Submit)self, (Transaction)txn [, (int)count=1 [, (object)itemdata=None [, (bool)spool=False]]]) -> SubmitResult :

This method is DEPRECATED. Use *Schedd.submit()* instead.

Submit the current object to a remote queue.

param txn
An active transaction object (see *Schedd.transaction()*).

type txn
Transaction

param int count
A queue count for each item from the iterator, defaults to 1.

param from
an iterator of strings or dictionaries containing the itemdata for each job as in `queue in` or `queue from`.

param bool spool
Modify the job ClassAds to indicate that it should wait for input before starting. defaults to false.

return
a *SubmitResult*, containing the cluster ID, cluster ClassAd and range of Job ids Cluster ID of the submitted job(s).

rtype
SubmitResult

raises RuntimeError
if the submission fails.

expand(*attr*) → str :

Expand all macros for the given attribute.

Parameters
attr (*str*) – The name of the relevant attribute.

Returns
The value of the given attribute; all macros are expanded.

Return type
str

jobs(*count=0*, *itemdata=None*, *clusterid=1*, *procid=0*, *qdate=0*, *owner=''*) → SubmitJobsIterator :

Turn the current object into a sequence of simulated job ClassAds

Parameters

- **count** (*int*) – the queue count for each item in the from list, defaults to 1
- **from** – a iterator of strings or dictionaries containing the itemdata for each job e.g. 'queue in' or 'queue from'

- **clusterid** (*int*) – the value to use for ClusterId when making job ads, defaults to 1
- **procid** (*int*) – the initial value for ProcId when making job ads, defaults to 0
- **qdate** (*str*) – a UNIX timestamp value for the QDATE attribute of the jobs, 0 means use the current time.
- **owner** (*str*) – a string value for the Owner attribute of the job

Returns
An iterator for the resulting job ads.

Raises
RuntimeError – if valid job ads cannot be made

procs(*count=0*, *itemdata=None*, *clusterid=1*, *procid=0*, *qdate=0*, *owner=''*) → SubmitJobsIterator :

Turn the current object into a sequence of simulated job proc ClassAds. The first ClassAd will be the cluster ad plus a ProcId attribute

Parameters

- **count** (*int*) – the queue count for each item in the from list, defaults to 1
- **from** – a iterator of strings or dictionaries containing the foreach data e.g. 'queue in' or 'queue from'
- **clusterid** (*int*) – the value to use for ClusterId when making job ads, defaults to 1
- **procid** (*int*) – the initial value for ProcId when making job ads, defaults to 0
- **qdate** (*str*) – a UNIX timestamp value for the QDATE attribute of the jobs, 0 means use the current time.
- **owner** (*str*) – a string value for the Owner attribute of the job

Returns
An iterator for the resulting job ads.

Raises
RuntimeError – if valid job ads cannot be made

itemdata(*qargs=''*) → QueueItemsIterator :

Create an iterator over itemdata derived from a queue statement.

For example `itemdata("matching *.dat")` would return an iterator of filenames that end in `.dat` from the current directory. This is the same iterator used by *condor_submit* when processing `QUEUE` statements.

Parameters
queue (*str*) – a submit file queue statement, or the arguments to a submit file queue statement.

Returns
An iterator for the resulting items

getQArgs() → str :

Returns arguments specified in the `QUEUE` statement passed to the constructor. These are the arguments that will be used by the *Submit.itemdata()* method if not overridden.

setQArgs(*args*) → None :

Sets the arguments to be used by subsequent calls to the *Submit.itemdata()*.

Parameters
args (*str*) – The arguments to pass to the `QUEUE` statement.

static from_dag(*filename*, *options={}*) → Submit :

Constructs a new *Submit* that could be used to submit the DAG described by the file found at `filename`.

This static method essentially does the first half of the work that *condor_submit_dag* does: it produces the submit description for the DAGMan job that will execute the DAG. However, in addition to writing this submit description to disk, it also produces a *Submit* object with the same information that can be submitted via the normal Python bindings submit machinery.

Parameters

- **filename** (*str*) – The path to the DAG description file.
- **options** (*dict*) – Additional arguments to *condor_submit_dag*. Supports `dagman` *(str)*, `force` *(bool)*, `schedd-daemon-ad-file` *(str)*, `schedd-address-file` *(str)*, `AlwaysRunPost` *(bool)*, `maxidle` *(int)*, `maxjobs` *(int)*, `MaxPre` *(int)*, `MaxPost` *(int)*, `UseDagDir` *(bool)*, `debug` *(int)*, `outfile_dir` *(str)*, `config` *(str)*, `batch-name` *(str)*, `load_save` *(str)*, `AutoRescue` *(bool)*, `DoRescueFrom` *(int)*, `AllowVersionMismatch` *(bool)*, `do_recurse` *(bool)*, `update_submit` *(bool)*, `import_env` *(bool)*, `include_env` *(str)*, `insert_env` *(str)*, `DumpRescue` *(bool)*, `valgrind` *(bool)*, `priority` *(int)*, `suppress_notification` *(bool)*, `DoRecov` *(bool)*

Returns
A *Submit* description for the DAG described in `filename`

Return type
Submit

setSubmitMethod(*method_value=-1*, *allow_reserved_values=False*) → None :

Sets the **Job Ad** attribute `JobSubmitMethod` to passed over number. `method_value` is recommended to be set to a value of `100` or greater to avoid confusion to pre-set values. Negative numbers will result in `JobSubmitMethod` to not be defined in the **Job Ad**. If wanted, any number can be set by passing `True` to `allow_reserved_values`. This allows any positive number to be set to `JobSubmitMethod`. This includes all reserved numbers. **Note~** Setting of `JobSubmitMethod` must occur before job is submitted to Schedd.

Parameters

- **method_value** (*int*) – Value set to `JobSubmitMethod`.
- **allow_reserved_values** (*bool*) – Boolean that allows any number to be set to `JobSubmitMethod`.

getSubmitMethod() → int :

Returns
`JobSubmitMethod` attribute value. See table or use *condor_q -help Submit* for values.

Return type
int

class htcondor.**QueueItemsIterator**

An iterator over itemdata produced by *Submit.itemdata()*.

class htcondor.**SubmitResult**

cluster() → int :

Returns
the ClusterID of the submitted jobs.

Return type
int

clusterad() → ClassAd :

Returns
the cluster Ad of the submitted jobs.

Return type
classad.ClassAd

first_proc() → int :

Returns
the first ProcID of the submitted jobs.

Return type
int

num_procs() → int :

Returns
the number of submitted jobs.

Return type
int

8.4.4 Interacting with Negotiators

class htcondor.**Negotiator**(*ad*)

This class provides a query interface to the *condor_negotiator*. It primarily allows one to query and set various parameters in the fair-share accounting.

Parameters
location_ad (*ClassAd* or DaemonLocation) – A ClassAd or DaemonLocation describing the *condor_negotiator* location and version. If omitted, the default pool negotiator is assumed.

deleteUser(*user*) → None :

Delete all records of a user from the Negotiator's fair-share accounting.

Parameters
user (*str*) – A fully-qualified user name (USER@DOMAIN).

getPriorities(*rollup*) → list :

Retrieve the pool accounting information, one per entry. Returns a list of accounting ClassAds.

Parameters
rollup (*bool*) – Set to True if accounting information, as applied to hierarchical group quotas, should be summed for groups and subgroups.

Returns
A list of accounting ads, one per entity.

Return type
list[*ClassAd*]

getResourceUsage(*user*) → list :

Get the resources (slots) used by a specified user.

Parameters
user (*str*) – A fully-qualified user name (USER@DOMAIN).

Returns
List of ads describing the resources (slots) in use.

Return type
list[*ClassAd*]

resetAllUsage() → None :

Reset all usage accounting. All known user records in the negotiator are deleted.

resetUsage(*user*) → None :

Reset all usage accounting of the specified user.

Parameters
user (*str*) – A fully-qualified user name (USER@DOMAIN).

setBeginUsage(*user*, *value*) → None :

Manually set the time that a user begins using the pool.

Parameters

- **user** (*str*) – A fully-qualified user name (USER@DOMAIN).
- **value** (*int*) – The Unix timestamp of initial usage.

setCeiling(*user*, *ceiling*) → None :

Set the submitter ceiling of a specified user.

Parameters

- **user** (*str*) – A fully-qualified user name (USER@DOMAIN).
- **ceiling** (*float*) – The ceiling t be set for the submitter; must be greater-than or equal-to -1.0.

setLastUsage(*user*, *value*) → None :

Manually set the time that a user last used the pool.

Parameters

- **user** (*str*) – A fully-qualified user name (USER@DOMAIN).
- **value** (*int*) – The Unix timestamp of last usage.

setFactor(*user*, *factor*) → None :

Set the priority factor of a specified user.

Parameters

- **user** (*str*) – A fully-qualified user name (USER@DOMAIN).
- **factor** (*float*) – The priority factor to be set for the user; must be greater-than or equal-to 1.0.

setPriority(*user*, *prio*) → None :

Set the real priority of a specified user.

Parameters

- **user** (*str*) – A fully-qualified user name (USER@DOMAIN).
- **prio** (*float*) – The priority to be set for the user; must be greater-than 0.0.

setUsage(*user*, *usage*) → None :

Set the accumulated usage of a specified user.

Parameters

- **user** (*str*) – A fully-qualified user name (USER@DOMAIN).
- **usage** (*float*) – The usage, in hours, to be set for the user.

8.4.5 Managing Starters and Claims

class htcondor.**Startd**(*ad=None*)

A class that represents a Startd.

Parameters

locaton_ad – A ClassAd or DaemonLocation describing the the startd location and version. If omitted, the local startd is assumed.

drainJobs(*drain_type=0*, *on_completion=0*, *check_expr='true'*, *start_expr='false'*, *reason='by command'*) → str :

Begin draining jobs from the startd.

Parameters

- **drain_type** (*DrainTypes*) – How fast to drain the jobs. Defaults to DRAIN_GRACEFUL if not specified.
- **on_completion** (*int*) – Whether the startd should start accepting jobs again once draining is complete. Otherwise, it will remain in the drained state. Values are 0 for Nothing, 1 for Resume, 2 for Exit, 3 for Restart. Defaults to 0.
- **check_expr** (str or ExprTree) – An expression string that must evaluate to true for all slots for draining to begin. Defaults to 'true'.
- **start_expr** (str or ExprTree) – The expression that the startd should use while draining.
- **reason** (*str*) – A string describing the reason for draining. defaults to "by command"

Returns

An opaque request ID that can be used to cancel draining via *Startd.cancelDrainJobs()*

Return type

str

cancelDrainJobs(*request_id=''*) → None :

Cancel a draining request.

Parameters

request_id (*str*) – Specifies a draining request to cancel. If not specified, all draining requests for this startd are canceled.

class htcondor.**DrainTypes**

Draining policies that can be sent to a *condor_startd*.

The values of the enumeration are:

Fast

Graceful

Quick

class htcondor.**VacateTypes**

Vacate policies that can be sent to a *condor_startd*.

The values of the enumeration are:

Fast

Graceful

8.4.6 Security Management

class htcondor.**Credd**(*ad=None*)

A class for sending Credential commands to a Credd, Schedd or Master.

Parameters
location_ad (*ClassAd* or DaemonLocation) – A ClassAd or DaemonLocation describing the Credd, Schedd or Master location. If omitted, the local schedd is assumed.

add_password(*password*, *user=''*) → None :

Store the password in the Credd for the current user (or for the given user).

Parameters

- **password** (*str*) – The password.
- **user** (*str*) – Which user to store the credential for (defaults to the current user).

delete_password(*user=''*) → None :

Delete the password in the Credd for the current user (or for the given user).

Parameters
user (*str*) – Which user to store the credential for (defaults to the current user).

query_password(*user=''*) → bool :

Check to see if the current user (or the given user) has a password stored in the Credd.

Parameters
user (*str*) – Which user to store the credential for (defaults to the current user).

Returns
bool

add_user_cred(*credtype*, *credential*, *user=''*) → None :

Store a credential in the Credd for the current user (or for the given user).

Parameters

- **credtype** (*CredTypes*) – The type of credential to store.
- **credential** (*bytes*) – The credential to store.
- **user** (*str*) – Which user to store the credential for (defaults to the current user).

delete_user_cred(*credtype*, *user=''*) → None :

Delete a credential of the given credtype for the current user (or for the given user).

Parameters

- **credtype** (*CredTypes*) – The type of credential to delete.

- **user** (*str*) – Which user to store the credential for (defaults to the current user).

query_user_cred(*credtype*, *user=''*) → int :

Query whether the current user (or the given user) has a credential of the given type stored.

Parameters

- **credtype** (*CredTypes*) – The type of credential to query for.
- **user** (*str*) – Which user to store the credential for (defaults to the current user).

Returns

The time that the user credential was last updated, or `None` if there is no credential

add_user_service_cred(*credtype*, *credential*, *service*, *handle=''*, *user=''*) → None :

Store a credential in the Credd for the current user, or for the given user.

To specify multiple credential for the same service (e.g., you want to transfer files from two different accounts that are on the same service), give each a unique `handle`.

Parameters

- **credtype** (*CredTypes*) – The type of credential to store.
- **credential** (*bytes*) – The credential to store.
- **service** (*str*) – The service name.
- **handle** (*str*) – Optional service handle (defaults to no handle).
- **user** (*str*) – Which user to store the credential for (defaults to the current user).

delete_user_service_cred(*credtype*, *service*, *handle=''*, *user=''*) → None :

Delete a credential of the given `credtype` for service `service` for the current user (or for the given `user`).

Parameters

- **credtype** (*CredTypes*) – The type of credential to delete.
- **service** (*str*) – The service name.
- **handle** (*str*) – Optional service handle (defaults to no handle).
- **user** (*str*) – Which user to store the credential for (defaults to the current user).

query_user_service_cred(*credtype*, *service*, *handle=''*, *user=''*) → CredStatus :

Query whether the current user (or the given `user`) has a credential of the given `credtype` stored.

Parameters

- **credtype** (*CredTypes*) – The type of credential to check storage for.
- **service** (*str*) – The service name.
- **handle** (*str*) – Optional service handle (defaults to no handle).
- **user** (*str*) – Which user to store the credential for (defaults to the current user).

Returns

CredStatus

check_user_service_creds(*credtype*, *services*, *user=''*) → CredCheck :

Check to see if the current user (or the given `user`) has a given set of service credentials, and if any credentials are missing, create a temporary URL that can be used to acquire the missing service credentials.

Parameters

- **credtype** (*CredTypes*) – The type of credentials to check for.
- **services** (List[*classad.ClassAd*]) – The list of services that are needed.
- **user** (*str*) – Which user to store the credential for (defaults to the current user).

Returns
CredCheck

class htcondor.**CredTypes**

The types of credentials that can be managed by a *condor_credd*.

The values of the enumeration are:

Password

Kerberos

OAuth

class htcondor.**CredCheck**

class htcondor.**CredStatus**

class htcondor.**SecMan**(*arg1*)

A class that represents the internal HTCondor security state.

If a security session becomes invalid, for example, because the remote daemon restarts, reuses the same port, and the client continues to use the session, then all future commands will fail with strange connection errors. This is the only mechanism to invalidate in-memory sessions.

The *SecMan* can also behave as a context manager; when created, the object can be used to set temporary security configurations that only last during the lifetime of the security object.

invalidateAllSessions() → None :

Invalidate all security sessions. Any future connections to a daemon will cause a new security session to be created.

ping(*ad*, *command='DC_NOP'*) → ClassAd :

Perform a test authorization against a remote daemon for a given command.

Parameters

- **ad** (str or *ClassAd*) – The ClassAd of the daemon as returned by *Collector.locate()*; alternately, the sinful string can be given directly as the first parameter.
- **command** – The DaemonCore command to try; if not given, `'DC_NOP'` will be used.

Returns
An ad describing the results of the test security negotiation.

Return type
ClassAd

getCommandString(*command_int*) → str :

Return the string name corresponding to a given integer command.

Parameters
command_int (*int*) – The integer command to get the string name of.

setConfig(*key*, *value*) → None :

Set a temporary configuration variable; this will be kept for all security sessions in this thread for as long as the *SecMan* object is alive.

Parameters

- **key** (*str*) – Configuration key to set.
- **value** (*str*) – Temporary value to set.

setPoolPassword(*new_pass*) → None :

Set the pool password.

Parameters
new_pass (*str*) – Updated pool password to use for new security negotiations.

setTag(*tag*) → None :

Set the authentication context tag for the current thread.

All security sessions negotiated with the same tag will only be utilized when that tag is active.

For example, if thread A has a tag set to `'Joe'` and thread B has a tag set to `'Jane'`, then all security sessions negotiated for thread A will not be used for thread B.

Parameters
tag (*str*) – New tag to set.

setToken(*token*) → None :

Set the token used for auth.

Parameters
token (*Token*) – The object representing the token contents

class htcondor.**Token**(*contents*)

A class representing a generated HTCondor authentication token.

Parameters
contents (*str*) – The contents of the token.

write(*tokenfile=None*) → None :

Write the contents of the token into the appropriate token directory on disk.

Parameters
tokenfile – Filename inside the user token directory where the token will be written.

class htcondor.**TokenRequest**(*identity=''*, *bounding_set=None*, *lifetime=-1*)

A class representing a request for a HTCondor authentication token.

Parameters

- **identity** (*str*) – Requested identity from the remote daemon (the empty string implies condor user).
- **bounding_set** (*list[str]*) – A list of authorizations that the token is restricted to.
- **lifetime** (*int*) – Requested lifetime, in seconds, that the token will be valid for.

done() → bool :

Check to see if the token request has completed.

Returns
`True` if the request is complete; `False` otherwise. May throw an exception.

Return type
bool

property request_id

The ID of the request at the remote daemon.

result(*timeout=0*) → Token :

Return the result of the token request. Will block until the token request is approved or the timeout is hit (a timeoute of 0, the default, indicates this method may block indefinitely).

Returns
The token resulting from this request.

Return type
Token

submit(*ad=None*) → None :

Submit the token request to a remote daemon.

Parameters
ad (*ClassAd*) – ClassAd describing the location of the remote daemon.

8.4.7 Reading Job Events

The following is a complete example of submitting a job and waiting (forever) for it to finish. The next example implements a time-out.

```
#!/usr/bin/env python3

import htcondor

# Create a job description.  It _must_ set `log` to create a job event log.
logFileName = "sleep.log"
submit = htcondor.Submit(
    f"""
    executable = /bin/sleep
    transfer_executable = false
    arguments = 5

    log = {logFileName}
    """
)

# Submit the job description, creating the job.
result = htcondor.Schedd().submit(submit, count=1)
clusterID = result.cluster()

# Wait (forever) for the job to finish.
jel = htcondor.JobEventLog(logFileName)
for event in jel.events(stop_after=None):
    # HTCondor appends to job event logs by default, so if you run
    # this example more than once, there will be more than one job
    # in the log.  Make sure we have the right one.
    if event.cluster != clusterID or event.proc != 0:
        continue
```

(continues on next page)

(continued from previous page)

```
    if event.type == htcondor.JobEventType.JOB_TERMINATED:
        if(event["TerminatedNormally"]):
            print(f"Job terminated normally with return value {event['ReturnValue']}.")
        else:
            print(f"Job terminated on signal {event['TerminatedBySignal']}.");
        break

    if event.type in { htcondor.JobEventType.JOB_ABORTED,
                       htcondor.JobEventType.JOB_HELD,
                       htcondor.JobEventType.CLUSTER_REMOVE }:
        print("Job aborted, held, or removed.")
        break

    # We expect to see the first three events in this list, and allow
    # don't consider the others to be terminal.
    if event.type not in { htcondor.JobEventType.SUBMIT,
                           htcondor.JobEventType.EXECUTE,
                           htcondor.JobEventType.IMAGE_SIZE,
                           htcondor.JobEventType.JOB_EVICTED,
                           htcondor.JobEventType.JOB_SUSPENDED,
                           htcondor.JobEventType.JOB_UNSUSPENDED }:
        print(f"Unexpected job event: {event.type}!");
        break
```

The following example includes a deadline for the job to finish. To make it quick to run the example, the deadline is only ten seconds; real jobs will almost always take considerably longer. You can change `arguments = 20` to `arguments = 5` to verify that this example correctly detects the job finishing. For the same reason, we check once a second to see if the deadline has expired. In practice, you should check much less frequently, depending on how quickly your script needs to react and how long you expect the job to last. In most cases, even once a minute is more frequent than necessary or appropriate on shared resources; every five minutes is better.

```
#!/usr/bin/env python3

import time
import htcondor

# Create a job description.  It _must_ set `log` to create a job event log.
logFileName = "sleep.log"
submit = htcondor.Submit(
    f"""
    executable = /bin/sleep
    transfer_executable = false
    arguments = 20

    log = {logFileName}
    """
)

# Submit the job description, creating the job.
result = htcondor.Schedd().submit(submit, count=1)
clusterID = result.cluster()
```

(continues on next page)

(continued from previous page)

```
def waitForJob(deadline):
    jel = htcondor.JobEventLog(logFileName)
    while time.time() < deadline:
        # In real code, this should be more like stop_after=300; see above.
        for event in jel.events(stop_after=1):
            # HTCondor appends to job event logs by default, so if you run
            # this example more than once, there will be more than one job
            # in the log.  Make sure we have the right one.
            if event.cluster != clusterID or event.proc != 0:
                continue
            if event.type == htcondor.JobEventType.JOB_TERMINATED:
                if(event["TerminatedNormally"]):
                    ␣
↪print(f"Job terminated normally with return value {event['ReturnValue']}.")
                else:
                    print(f"Job terminated on signal {event['TerminatedBySignal']}.");
                return True
            if event.type in { htcondor.JobEventType.JOB_ABORTED,
                               htcondor.JobEventType.JOB_HELD,
                               htcondor.JobEventType.CLUSTER_REMOVE }:
                print("Job aborted, held, or removed.")
                return True
            # We expect to see the first three events in this list, and allow
            # don't consider the others to be terminal.
            if event.type not in { htcondor.JobEventType.SUBMIT,
                                   htcondor.JobEventType.EXECUTE,
                                   htcondor.JobEventType.IMAGE_SIZE,
                                   htcondor.JobEventType.JOB_EVICTED,
                                   htcondor.JobEventType.JOB_SUSPENDED,
                                   htcondor.JobEventType.JOB_UNSUSPENDED }:
                print(f"Unexpected job event: {event.type}!");
                return True
    else:
        print("Deadline expired.")
        return False

# Wait no more than 10 seconds for the job finish.
waitForJob(time.time() + 10);
```

Note that which job events are terminal, expected, or allowed may vary somewhat from job to job; for instance, it's possible to submit a job which releases itself from certain hold conditions.

class htcondor.**JobEventLog**(*filename*)

Reads user job event logs from `filename`.

By default, it blocks waiting for new events, but it may be used to poll for them:

```
import htcondor

jel = htcondor.JobEventLog("file.log")

# Read all currently-available events without blocking.
```

(continues on next page)

(continued from previous page)

```
for event in jel.events(stop_after=0):
    print(event)

print("We found the the end of file")
```

A pickled *JobEventLog* resumes iterating over events where it left off if and only if, after being unpickled, the job event log file is identical except for appended events.

Parameters
filename (*str*) – A file containing a user job event log.

events(*stop_after*) → object :

Return an iterator over *JobEvent* objects from the filename given in the constructor. By default, the iterator blocks forever waiting for new events.

Parameters
stop_after (*int*) – After how many seconds should the iterator stop waiting for new events?

If `None` (the default), wait forever.

If `0`, never wait. Does not block.

For any other value, wait (block) for that many seconds for a new event, raising `StopIteration` if one does not appear. (This does not invalidate the iterator.)

close() → None :

Closes any open underlying file. This object will no longer iterate.

class htcondor.**JobEvent**

Represents a single job event from the job event log. Use *JobEventLog* to get an iterator over the job events from a file.

Because all events have `type`, `cluster`, `proc`, and `timestamp`, those are accessed via attributes (see below).

The rest of the information in the *JobEvent* can be accessed by key. *JobEvent* behaves like a read-only Python `dict`, with `get`, `keys`, `items`, and `values` methods, and supports `len` and `in` (`if "attribute" in job_event`, for example).

Attention: Although the attribute `type` is a *JobEventType* type, when acting as dictionary, a *JobEvent* object returns types as if it were a *ClassAd*, so comparisons to enumerated values must use the == operator. (No current event type has *ExprTree* values.)

type

The event type.

Return type
JobEventType

cluster

The `clusterid` of the job the event is for.

Return type
int

proc

The `procid` of the job the event is for.

Return type
int

timestamp
The timestamp of the event.

Return type
str

get(*key*, *default=None*) → object :
As `dict.get()`.

keys() → list :
As `dict.keys()`.

values() → list :
As `dict.values()`.

items() → list :
As `dict.items()`.

class htcondor.**JobEventType**
The type event of a user log event; corresponds to `ULogEventNumber` in the C++ source.

The values of the enumeration are:

SUBMIT

EXECUTE

EXECUTABLE_ERROR

CHECKPOINTED

JOB_EVICTED

JOB_TERMINATED

IMAGE_SIZE

SHADOW_EXCEPTION

GENERIC

JOB_ABORTED

JOB_SUSPENDED

JOB_UNSUSPENDED

JOB_HELD

JOB_RELEASED

NODE_EXECUTE

NODE_TERMINATED

POST_SCRIPT_TERMINATED

GLOBUS_SUBMIT

GLOBUS_SUBMIT_FAILED

GLOBUS_RESOURCE_UP

GLOBUS_RESOURCE_DOWN

REMOTE_ERROR

JOB_DISCONNECTED

JOB_RECONNECTED

JOB_RECONNECT_FAILED

GRID_RESOURCE_UP

GRID_RESOURCE_DOWN

GRID_SUBMIT

JOB_AD_INFORMATION

JOB_STATUS_UNKNOWN

JOB_STATUS_KNOWN

JOB_STAGE_IN

JOB_STAGE_OUT

ATTRIBUTE_UPDATE

PRESKIP

CLUSTER_SUBMIT

CLUSTER_REMOVE

FACTORY_PAUSED

FACTORY_RESUMED

NONE

FILE_TRANSFER

RESERVE_SPACE

RELEASE_SPACE

FILE_COMPLETE

FILE_USED

FILE_REMOVED

class htcondor.**FileTransferEventType**

The event type for file transfer events; corresponds to `FileTransferEventType` in the C++ source.

The values of the enumeration are:

IN_QUEUED

IN_STARTED

IN_FINISHED

OUT_QUEUED

OUT_STARTED

OUT_FINISHED

8.4.8 HTCondor Configuration

htcondor.**param = <htcondor.htcondor._Param object>**

Provides dictionary-like access the HTCondor configuration.

An instance of *_Param*. Upon importing the *htcondor* module, the HTCondor configuration files are parsed and populate this dictionary-like object.

htcondor.**reload_config**() → None :

Reload the HTCondor configuration from disk.

class htcondor.**_Param**

A dictionary-like object for the local HTCondor configuration; the keys and values of this object are the keys and values of the HTCondor configuration.

The `get`, `setdefault`, `update`, `keys`, `items`, and `values` methods of this class have the same semantics as a Python dictionary.

Writing to a `_Param` object will update the in-memory HTCondor configuration.

class htcondor.**RemoteParam**(*ad*)

The *RemoteParam* class provides a dictionary-like interface to the configuration of an HTCondor daemon. The `get`, `setdefault`, `update`, `keys`, `items`, and `values` methods of this class have the same semantics as a Python dictionary.

Parameters
ad (*ClassAd*) – An ad containing the location of the remote daemon.

refresh() → None :

Rebuilds the dictionary based on the current configuration of the daemon.

htcondor.**platform**() → str :

Returns the platform of HTCondor this module is running on.

htcondor.**version**() → str :

Returns the version of HTCondor this module is linked against.

8.4.9 HTCondor Logging

htcondor.**enable_debug**() → None :

Enable debugging output from HTCondor, where output is sent to `stderr`. The logging level is controlled by the `TOOL_DEBUG` parameter.

htcondor.**enable_log**() → None :

Enable debugging output from HTCondor, where output is sent to a file. The log level is controlled by the parameter `TOOL_DEBUG`, and the file used is controlled by `TOOL_LOG`.

htcondor.**log**(*level*, *msg*) → None :

Log a message using the HTCondor logging subsystem.

Parameters

- **level** (*LogLevel*) – The log category and formatting indicator. Multiple LogLevel enum attributes may be OR'd together.
- **msg** (*str*) – A message to log.

class htcondor.**LogLevel**

The log level attribute to use with *log()*. Note that HTCondor mixes both a class (debug, network, all) and the header format (Timestamp, PID, NoHeader) within this enumeration.

The values of the enumeration are:

Always

Audit

Config

DaemonCore

Error

FullDebug

Hostname

Job

Machine

Network

NoHeader

PID

Priv

Protocol

Security

Status

SubSecond

Terse

Timestamp

Verbose

8.4.10 Esoteric Functionality

htcondor.**send_command**(*ad*, *dc*, *target*) → None :

Send a command to an HTCondor daemon specified by a location ClassAd.

Parameters

- **ad** (*ClassAd*) – Specifies the location of the daemon (typically, found by using *Collector.locate()*).
- **dc** (*DaemonCommands*) – A command type
- **target** (*str*) – An additional command to send to a daemon. Some commands require additional arguments; for example, sending `DaemonOff` to a *condor_master* requires one to specify which subsystem to turn off.

class htcondor.**DaemonCommands**

An enumeration of various state-changing commands that can be sent to a HTCondor daemon using *send_command()*.

The values of the enumeration are:

DaemonOn

DaemonOff

DaemonOffFast

DaemonOffPeaceful

DaemonsOn

DaemonsOff

DaemonsOffFast

DaemonsOffPeaceful

OffFast

OffForce

OffGraceful

OffPeaceful

Reconfig

Restart

RestartPeacful

SetForceShutdown

SetPeacefulShutdown

htcondor.**send_alive**(*[ad=None, pid=None, timeout=None*) → None :

Send a keep alive message to an HTCondor daemon.

This is used when the python process is run as a child daemon under the *condor_master*.

Parameters

- **ad** (*ClassAd*) – A *ClassAd* specifying the location of the daemon. This ad is typically found by using *Collector.locate()*.
- **pid** (*int*) – The process identifier for the keep alive. The default value of `None` uses the value from `os.getpid()`.
- **timeout** (*int*) – The number of seconds that this keep alive is valid. If a new keep alive is not received by the condor_master in time, then the process will be terminated. The default value is controlled by configuration variable `NOT_RESPONDING_TIMEOUT`.

htcondor.**set_subsystem**(*subsystem*, *type=htcondor.htcondor.SubsystemType(15)*) → None :

Set the subsystem name for the object.

The subsystem is primarily used for the parsing of the HTCondor configuration file.

Parameters

- **name** (*str*) – The subsystem name.
- **daemon_type** (*SubsystemType*) – The HTCondor daemon type. The default value of Auto infers the type from the name parameter.

class htcondor.**SubsystemType**

An enumeration of known subsystem names.

The values of the enumeration are:

Collector

Daemon

Dagman

GAHP

Job

Master

Negotiator

Schedd

Shadow

SharedPort

Startd

Starter

Submit

Tool

8.4.11 Exceptions

For backwards-compatibility, the exceptions in this module inherit from the built-in exceptions raised in earlier (pre-v8.9.9) versions.

class htcondor.**HTCondorException**

Never raised. The parent class of all exceptions raised by this module.

class htcondor.**HTCondorEnumError**

Raised when a value must be in an enumeration, but isn't.

class htcondor.**HTCondorInternalError**

Raised when HTCondor encounters an internal error.

class htcondor.**HTCondorIOError**

Raised instead of `IOError` for backwards compatibility.

class htcondor.**HTCondorLocateError**

Raised when HTCondor cannot locate a daemon.

class htcondor.**HTCondorReplyError**

Raised when HTCondor received an invalid reply from a daemon, or the daemon's reply indicated that it encountered an error.

class htcondor.**HTCondorTypeError**

Raised instead of `TypeError` for backwards compatibility.

class htcondor.**HTCondorValueError**

Raised instead of `ValueError` for backwards compatibility.

8.4.12 Thread Safety

Most of the `htcondor` module is protected by a lock that prevents multiple threads from executing locked functions at the same time. When two threads both want to call locked functions or methods, they will wait in line to execute them one at a time (the ordering between threads is not guaranteed beyond "first come first serve"). Examples of locked functions include: *Schedd.query()*, *Submit.queue()*, and *Schedd.edit()*.

Threads that are not trying to execute locked `htcondor` functions will be allowed to proceed normally.

This locking may cause unexpected slowdowns when using `htcondor` from multiple threads simultaneously.

8.5 htcondor.htchirp API Reference

htcondor.htchirp is a Python Chirp client compatible with the *condor_starter* Chirp proxy server. It is intended for use inside a running HTCondor job to access files on the submit machine or to query and modify job ClassAd attributes. Files can be read, written, or removed. Job attributes can be read, and most attributes can be updated.

Jobs that use *htcondor.htchirp* module must have the attribute `WantIOProxy` set to `true` in the job ClassAd (`want_io_proxy = true` in the submit description). *htcondor.htchirp* only works for jobs run in the vanilla, paralllel, and java universes.

htcondor.htchirp provides two objects for interacting with the *condor_starter* Chirp proxy server, *HTChirp* and *condor_chirp()*.

We recommend using *HTChirp* as a context manager, which automatically handles openining and closing the connection to the *condor_starter* Chirp proxy server:

```
from htcondor.htchirp import HTChirp

with HTChirp() as chirp:
    # inside this block, the connection is open
    i = chirp.get_job_attr("IterationNum")
    chirp.set_job_attr("IterationNum") = i + 1
```

The connection may be manually opened and closed using *HTChirp.connect()* and *HTChirp.disconnect()*.

condor_chirp() is a wrapper around *HTChirp* that takes a string containing a *condor_chirp* command (with arguments) and returns the value from the relevant *HTChirp* method.

class htcondor.htchirp.**HTChirp**(*host=None*, *port=None*, *auth=['cookie']*, *cookie=None*, *timeout=10*)

Chirp client for HTCondor

A Chirp client compatible with the HTCondor Chirp implementation.

If the host and port of a Chirp server are not specified, you are assumed to be running in a HTCondor job with `$_CONDOR_CHIRP_CONFIG` that contains the host, port, and cookie for connecting to the embedded chirp proxy.

Parameters

- **host** – the hostname or ip of the Chirp server
- **port** – the port of the Chirp server
- **auth** – a list of authentication methods to try
- **cookie** – the cookie string, if trying cookie authentication
- **timeout** – socket timeout, in seconds

connect(*auth_method=None*)

Connect to and authenticate with the Chirp server

Parameters
auth_method – If set, try the specific authentication method

is_connected()

Check if Chirp client is connected.

disconnect()

Close connection with the Chirp server

fetch(*remote_file*, *local_file*)

Copy a file from the submit machine to the execute machine.

Parameters

- **remote_file** – Path to file to be sent from the submit machine
- **local_file** – Path to file to be written to on the execute machine

Returns
Bytes written

put(*local_file*, *remote_file*, *flags='wct'*, *mode=None*)

Copy a file from the execute machine to the submit machine.

Specifying flags other than 'wct' (i.e. 'create or truncate file') when putting large files is not recommended as the entire file must be read into memory.

To put individual bytes into a file on the submit machine instead of an entire file, see the write() method.

Parameters

- **local_file** – Path to file to be sent from the execute machine
- **remote_file** – Path to file to be written to on the submit machine
- **flags** – File open modes (one or more of 'rwatcx') [default: 'wct']
- **mode** – Permission mode to set [default: 0777]

Returns
Size of written file

remove(*remote_file*)

Remove a file from the submit machine.

Parameters
remote_file – Path to file on the submit machine

get_job_attr(*job_attribute*)

Get the value of a job ClassAd attribute.

Parameters
job_attribute – The job attribute to query

Returns
The value of the job attribute as a string

set_job_attr(*job_attribute*, *attribute_value*)

Set the value of a job ClassAd attribute.

Parameters

- **job_attribute** – The job attribute to set
- **attribute_value** – The job attribute's new value

get_job_attr_delayed(*job_attribute*)

Get the value of a job ClassAd attribute from the local Starter.

This may differ from the value in the Schedd.

Parameters
job_attribute – The job attribute to query

Returns
The value of the job attribute as a string

set_job_attr_delayed(*job_attribute*, *attribute_value*)

Set the value of a job ClassAd attribute.

This variant of set_job_attr will not push the update immediately, but rather as a non-durable update during the next communication between starter and shadow.

Parameters

- **job_attribute** – The job attribute to set
- **attribute_value** – The job attribute's new value

ulog(*text*)

Log a generic string to the job log.

Parameters
text – String to log

read(*remote_path*, *length*, *offset=None*, *stride_length=None*, *stride_skip=None*)

Read up to 'length' bytes from a file on the remote machine.

Optionally, start at an offset and/or retrieve data in strides.

Parameters

- **remote_path** – Path to file
- **length** – Number of bytes to read
- **offset** – Number of bytes to offset from beginning of file
- **stride_length** – Number of bytes to read per stride
- **stride_skip** – Number of bytes to skip per stride

Returns
Data read from file

write(*data*, *remote_path*, *flags='w'*, *mode=None*, *length=None*, *offset=None*, *stride_length=None*, *stride_skip=None*)

Write bytes to a file on the remote matchine.

Optionally, specify the number of bytes to write, start at an offset, and/or write data in strides.

Parameters

- **data** – Bytes to write
- **remote_path** – Path to file
- **flags** – File open modes (one or more of 'rwatcx') [default: 'w']
- **mode** – Permission mode to set [default: 0777]
- **length** – Number of bytes to write [default: len(data)]
- **offset** – Number of bytes to offset from beginning of file
- **stride_length** – Number of bytes to write per stride
- **stride_skip** – Number of bytes to skip per stride

Returns
Number of bytes written

rename(*old_path*, *new_path*)

Rename (move) a file on the remote machine.

Parameters

- **old_path** – Path to file to be renamed
- **new_path** – Path to new file name

unlink(*remote_file*)

Delete a file on the remote machine.

Parameters
remote_file – Path to file

rmdir(*remote_path*, *recursive=False*)

Delete a directory on the remote machine.

The directory must be empty unless recursive is set to True.

Parameters

- **remote_path** – Path to directory
- **recursive** – If set to True, recursively delete remote_path

rmall(*remote_path*)

Recursively delete an entire directory on the remote machine.

Parameters
remote_path – Path to directory

mkdir(*remote_path*, *mode=None*)

Create a new directory on the remote machine.

Parameters

- **remote_path** – Path to new directory
- **mode** – Permission mode to set [default: 0777]

getfile(*remote_file*, *local_file*)

Retrieve an entire file efficiently from the remote machine.

Parameters

- **remote_file** – Path to file to be sent from remote machine
- **local_file** – Path to file to be written to on local machine

Returns
Bytes written

putfile(*local_file*, *remote_file*, *mode=None*)

Store an entire file efficiently to the remote machine.

This method will create or overwrite the file on the remote machine. If you want to append to a file, use the write() method.

Parameters

- **local_file** – Path to file to be sent from local machine
- **remote_file** – Path to file to be written to on remote machine
- **mode** – Permission mode to set [default: 0777]

Returns
Size of written file

getdir(*remote_path*, *stat_dict=False*)

List a directory on the remote machine.

Parameters

- **remote_path** – Path to directory
- **stat_dict** – If set to True, return a dict of file metadata

Returns
List of files, unless stat_dict is True

getlongdir(*remote_path*)

List a directory and all its file metadata on the remote machine.

Parameters
remote_path – Path to directory

Returns
A dict of file metadata

whoami()

Get the user's current identity with respect to this server.

Returns
The user's identity

whoareyou(*remote_host*)

Get the server's identity with respect to the remote host.

Parameters
remote_host – Remote host

Returns
The server's identity

link(*old_path*, *new_path*, *symbolic=False*)

Create a link on the remote machine.

Parameters

- **old_path** – File path to link from on the remote machine
- **new_path** – File path to link to on the remote machine
- **symbolic** – If set to True, use a symbolic link

symlink(*old_path*, *new_path*)

Create a symbolic link on the remote machine.

Parameters

- **old_path** – File path to symlink from on the remote machine
- **new_path** – File path to symlink to on the remote machine

readlink(*remote_path*)

Read the contents of a symbolic link.

Parameters
remote_path – File path on the remote machine

Returns
Contents of the link

stat(*remote_path*)

Get metadata for file on the remote machine.

If remote_path is a symbolic link, examine its target.

Parameters
remote_path – Path to file

Returns
Dict of file metadata

`lstat`(*remote_path*)

Get metadata for file on the remote machine.

If remote path is a symbolic link, examine the link.

Parameters
`remote_path` – Path to file

Returns
Dict of file metadata

`statfs`(*remote_path*)

Get metadata for a file system on the remote machine.

Parameters
`remote_path` – Path to examine

Returns
Dict of filesystem metadata

`access`(*remote_path*, *mode_str*)

Check access permissions.

Parameters

- `remote_path` – Path to examine
- `mode_str` – Mode to check (one or more of 'frwx')

Raises
`NotAuthorized` – If any access mode is not authorized

`chmod`(*remote_path*, *mode*)

Change permission mode of a path on the remote machine.

Parameters

- `remote_path` – Path
- `mode` – Permission mode to set

`chown`(*remote_path*, *uid*, *gid*)

Change the UID and/or GID of a path on the remote machine.

If remote_path is a symbolic link, change its target.

Parameters

- `remote_path` – Path
- `uid` – UID
- `gid` – GID

`lchown`(*remote_path*, *uid*, *gid*)

Changes the ownership of a file or directory.

If the path is a symbolic link, change the link.

Parameters

- `remote_path` – Path
- `uid` – UID
- `gid` – GID

truncate(*remote_path*, *length*)

Truncates a file on the remote machine to a given number of bytes.

Parameters

- **remote_path** – Path to file
- **length** – Truncated length

utime(*remote_path*, *actime*, *mtime*)

Change the access and modification times of a file on the remote machine.

Parameters

- **remote_path** – Path to file
- **actime** – Access time, in seconds (Unix epoch)
- **mtime** – Modification time, in seconds (Unix epoch)

htcondor.htchirp.**condor_chirp**(*chirp_args*, *return_exit_code=False*)

Call HTChirp methods using condor_chirp-style commands

See https://htcondor.readthedocs.io/en/latest/man-pages/condor_chirp.html for a list of commands, or use a Python interpreter to run `htchirp.py --help`.

Parameters

- **chirp_args** – List or string of arguments as would be passed to condor_chirp
- **return_exit_code** – If `True`, format and print return value in condor_chirp-style, and return 0 (success) or 1 (failure) (defaults to `False`).

Returns

Return value from the HTChirp method called, unless `return_exit_code=True` (see above).

8.6 htcondor.dags API Reference

Attention: This is not documentation for DAGMan itself! If you run into DAGMan jargon that isn't explained here, see *DAGMan Introduction*.

8.6.1 Creating DAGs

class htcondor.dags.**DAG**(*dagman_config=None*, *dagman_job_attributes=None*, *max_jobs_by_category=None*, *dot_config=None*, *jobstate_log=None*, *node_status_file=None*)

This object represents the entire DAGMan workflow, including both the execution graph and miscellaneous configuration options.

It contains the individual *NodeLayer* and *SubDAG* that are the "logical" nodes in the graph, created by the *layer()* and *subdag()* methods respectively.

Parameters

- **dagman_config** (Optional[Mapping[str, Any], None]) – A mapping of DAGMan configuration options.
- **dagman_job_attributes** (Optional[Mapping[str, Any], None]) – A mapping that describes additional HTCondor JobAd attributes for the DAGMan job itself.

- **max_jobs_by_category** (Optional[Mapping[str, int], None]) – A mapping that describes the maximum number of jobs (values) that should be run simultaneously from each category (keys).
- **dot_config** (Optional[*DotConfig*, None]) – Configuration options for writing a DOT file, as a *DotConfig*.
- **jobstate_log** (Optional[Path, None]) – The path to the jobstate log. If not given, the jobstate log will not be written.
- **node_status_file** (Optional[*NodeStatusFile*, None]) – Configuration options for the node status file, as a *NodeStatusFile*.

describe()

Return a tabular description of the DAG's structure.

Return type
str

property edges: Iterator[Tuple[Tuple[*BaseNode*, *BaseNode*], *BaseEdge*]]

Iterate over `((parent, child), edge)` tuples, for every edge in the graph.

Return type
Iterator[Tuple[Tuple[*BaseNode*, *BaseNode*], *BaseEdge*]]

final(***kwargs*)

Create the `FINAL` node of the DAG. A DAG can only have one `FINAL` node; if you call this method multiple times, it will override any previous calls. To customize the `FINAL` node after creation, modify the *FinalNode* instance that it returns.

Return type
FinalNode

glob(*pattern*)

Return a *Nodes* of the nodes in the DAG whose names match the glob `pattern`.

Return type
Nodes

layer(***kwargs*)

Create a new *NodeLayer* in the graph with no parents or children. Keyword arguments are forwarded to *NodeLayer*.

Return type
NodeLayer

property leaves: *Nodes*

A *Nodes* of the nodes in the DAG that have no children.

Return type
Nodes

property node_to_children: Dict[*BaseNode*, *Nodes*]

Return a dictionary that maps each node to a *Nodes* containing its children. The *Nodes* will be empty if the node has no children.

Return type
Dict[*BaseNode*, *Nodes*]

property node_to_parents: Dict[*BaseNode*, *Nodes*]

Return a dictionary that maps each node to a *Nodes* containing its parents. The *Nodes* will be empty if the node has no parents.

Return type
Dict[*BaseNode*, *Nodes*]

property nodes: *Nodes*

Iterate over all of the nodes in the DAG, in no particular order.

Return type
Nodes

property roots: *Nodes*

A *Nodes* of the nodes in the DAG that have no parents.

Return type
Nodes

select(*selector*)

Return a *Nodes* of the nodes in the DAG that satisfy `selector`. `selector` should be a function which takes a single *BaseNode* and returns `True` (will be included) or `False` (will not be included).

Return type
Nodes

subdag(***kwargs*)

Create a new *SubDAG* in the graph with no parents or children. Keyword arguments are forwarded to *SubDAG*.

Return type
SubDAG

walk(*order=WalkOrder.DEPTH_FIRST*)

Iterate over all of the nodes in the DAG, starting from the roots (i.e., the nodes with no parents), in either depth-first or breadth-first order.

Sibling order is not specified, and may be different in different calls to this method.

Parameters
order (*WalkOrder*) – Walk depth-first (children before siblings) or breadth-first (siblings before children).

Return type
Iterator[*BaseNode*]

walk_ancestors(*node*, *order=WalkOrder.DEPTH_FIRST*)

Iterate over all of the ancestors (i.e., parents, parents of parents, etc.) of some node, in either depth-first or breadth-first order.

Sibling order is not specified, and may be different in different calls to this method.

Parameters

- **node** (*BaseNode*) – The node to begin walking from. It will not be included in the results.
- **order** (*WalkOrder*) – Walk depth-first (parents before siblings) or breadth-first (siblings before parents).

Return type
Iterator[*BaseNode*]

walk_descendants(*node*, *order=WalkOrder.DEPTH_FIRST*)

Iterate over all of the descendants (i.e., children, children of children, etc.) of some node, in either depth-first or breadth-first order.

Sibling order is not specified, and may be different in different calls to this method.

Parameters

- **node** (*BaseNode*) – The node to begin walking from. It will not be included in the results.
- **order** (*WalkOrder*) – Walk depth-first (children before siblings) or breadth-first (siblings before children).

Return type
Iterator[*BaseNode*]

class htcondor.dags.**WalkOrder**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

An enumeration for keeping track of which order to walk through a graph. Depth-first means that parents/children will be visited before siblings. Breadth-first means that siblings will be visited before parents/children.

BREADTH_FIRST = 'BREADTH'

DEPTH_FIRST = 'DEPTH'

Nodes and Node-likes

class htcondor.dags.**BaseNode**(*dag*, ***, *name*, *dir=None*, *noop=False*, *done=False*, *retries=None*, *retry_unless_exit=None*, *pre=None*, *post=None*, *pre_skip_exit_code=None*, *priority=0*, *category=None*, *abort=None*)

This is the superclass for all node-like objects (things that can be the logical nodes in a *DAG*, like *NodeLayer* and *SubDAG*).

Generally, you do not need to construct nodes yourself; instead, they are created by calling methods like *DAG.layer()*, *DAG.subdag()*, *BaseNode.child_layer()*, and so forth. These methods automatically attach the new node to the same *DAG* as the node you called the method on.

Parameters

- **dag** (*DAG*) – Which *DAG* to attach this node to.
- **name** (str) – The human-readable name of this node.
- **dir** (Optional[Path, None]) – The directory to submit from. If None, it will be the directory the DAG itself was submitted from.
- **noop** (Union[bool, Mapping[int, bool]]) – If this is True, this node will be skipped and marked as completed, no matter what it says it does. For a *NodeLayer*, this can be dictionary mapping individual underlying node indices to their desired value.
- **done** (Union[bool, Mapping[int, bool]]) – If this is True, this node will be considered already completed. For a *NodeLayer*, this can be dictionary mapping individual underlying node indices to their desired value.
- **retries** (Optional[int, None]) – The number of times to retry the node if it fails (defined by retry_unless_exit).
- **retry_unless_exit** (Optional[int, None]) – If the node exits with this code, it will not be retried.
- **pre** (Optional[*Script*, None]) – A *Script* to run before the node itself.

- **post** (Optional[*Script*, None]) – A *Script* to run after the node itself.
- **pre_skip_exit_code** (Optional[int, None]) – If the pre-script exits with this code, the node will be skipped.
- **priority** (int) – The internal priority for DAGMan to run this node.
- **category** (Optional[str, None]) – Which CATEGORY this node belongs to.
- **abort** (Optional[*DAGAbortCondition*, None]) – A *DAGAbortCondition* which may cause the entire DAG to stop if this node exits in a certain way.

add_children(**nodes*, *edge=None*)

Makes all of the nodes children of this node.

Parameters

- **nodes** – The nodes to make children of this node.
- **edge** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.

Returns
self – This method returns self.

Return type
BaseNode

add_parents(**nodes*, *edge=None*)

Makes all of the nodes parents of this node.

Parameters

- **nodes** – The nodes to make parents of this node.
- **edge** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.

Returns
self – This method returns self.

Return type
BaseNode

child_layer(*edge=None*, ***kwargs*)

Create a new *NodeLayer* which is a child of this node.

Parameters

- **edge** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.
- **kwargs** – Additional keyword arguments are passed to the *NodeLayer* constructor.

Returns
node_layer – The newly-created node layer.

Return type
NodeLayer

child_subdag(*edge=None*, ***kwargs*)

Create a new *SubDAG* which is a child of this node.

Parameters

- **edge** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.
- **kwargs** – Additional keyword arguments are passed to the *SubDAG* constructor.

Returns
subdag – The newly-created sub-DAG.

Return type
SubDAG

property children: *Nodes*

Return a *Nodes* containing all of the children of this node.

Return type
Nodes

parent_layer(*edge=None*, ***kwargs*)

Create a new *NodeLayer* which is a parent of this node.

Parameters

- **edge** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.
- **kwargs** – Additional keyword arguments are passed to the *NodeLayer* constructor.

Returns
node_layer – The newly-created node layer.

Return type
NodeLayer

parent_subdag(*edge=None*, ***kwargs*)

Create a new *SubDAG* which is a parent of this node.

Parameters

- **edge** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.
- **kwargs** – Additional keyword arguments are passed to the *SubDAG* constructor.

Returns
subdag – The newly-created sub-DAG.

Return type
SubDAG

property parents: *Nodes*

Return a *Nodes* containing all of the parents of this node.

Return type
Nodes

remove_children(**nodes*)

Makes sure that the nodes are **not** children of this node.

Parameters
nodes – The nodes to remove edges from.

Returns
self – This method returns self.

Return type
BaseNode

remove_parents(**nodes*)

Makes sure that the `nodes` are **not** parents of this node.

Parameters
nodes – The nodes to remove edges from.

Returns
self – This method returns `self`.

Return type
BaseNode

walk_ancestors(*order=WalkOrder.DEPTH_FIRST*)

Walk over all of the ancestors of this node, in the given order.

Return type
Iterator[*BaseNode*]

walk_descendants(*order=WalkOrder.DEPTH_FIRST*)

Walk over all of the descendants of this node, in the given order.

Return type
Iterator[*BaseNode*]

class htcondor.dags.**NodeLayer**(*dag*, *, *submit_description=None*, *vars=None*, ***kwargs*)

Bases: *BaseNode*

Represents a "layer" of actual `JOB` nodes that share a submit description and edge relationships. Each underlying actual node's attributes may be customized using `vars`.

Parameters

- **dag** (*DAG*) – The DAG to connect this node to.
- **submit_description** (Union[*Submit*, None, Path]) – The HTCondor submit description for this node. Can be either an *htcondor.Submit* object or a Path to an existing submit file on disk.
- **vars** (Optional[Iterable[Dict[str, str]], None]) – The `VARS` for this logical node; one actual node will be created for each dictionary in the `vars`.
- **kwargs** – Additional keyword arguments are passed to the *BaseNode* constructor.

class htcondor.dags.**SubDAG**(*dag*, *, *dag_file*, ***kwargs*)

Bases: *BaseNode*

Represents a `SUBDAG` in the graph.

See *SUBDAG EXTERNAL* for more information on sub-DAGs.

Parameters

- **dag** (*DAG*) – The DAG to connect this node to.
- **dag_file** (Path) – The `pathlib.Path` to where the sub-DAG's DAG description file is (or will be).
- **kwargs** – Additional keyword arguments are passed to the *BaseNode* constructor.

class htcondor.dags.**FinalNode**(*dag*, *submit_description=None*, ***kwargs*)

Bases: *BaseNode*

Represents the FINAL node in a DAG.

See *Final Node* for more information on the FINAL node.

Parameters

- **dag** (*DAG*) – The DAG to connect this node to.
- **submit_description** (Union[*Submit*, None, Path]) – The HTCondor submit description for this node. Can be either an *htcondor.Submit* object or a Path to an existing submit file on disk.
- **kwargs** – Additional keyword arguments are passed to the *BaseNode* constructor.

class htcondor.dags.**Nodes**(**nodes*)

This class represents an arbitrary collection of *BaseNode*. In many cases, especially when manipulating the structure of the graph, it can be used as a replacement for directly iterating over collections of nodes.

Parameters

nodes (Union[*BaseNode*, Iterable[*BaseNode*]]) – The logical nodes that will be in this *Nodes*.

add_children(**nodes*, *type=None*)

Makes all of the nodes children of all of the nodes in this *Nodes*.

Parameters

- **nodes** – The nodes to make children of this *Nodes*.
- **type** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.

Returns

self – This method returns self.

Return type

Nodes

add_parents(**nodes*, *type=None*)

Makes all of the nodes parents of all of the nodes in this *Nodes*.

Parameters

- **nodes** – The nodes to make parents of this *Nodes*.
- **type** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.

Returns

self – This method returns self.

Return type

Nodes

child_layer(*type=None*, ***kwargs*)

Create a new *NodeLayer* which is a child of all of the nodes in this *Nodes*.

Parameters

- **type** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.

- **kwargs** – Additional keyword arguments are passed to the *NodeLayer* constructor.

Returns
node_layer – The newly-created node layer.

Return type
NodeLayer

child_subdag(*type=None*, ***kwargs*)

Create a new *SubDAG* which is a child of all of the nodes in this *Nodes*.

Parameters

- **type** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.
- **kwargs** – Additional keyword arguments are passed to the *SubDAG* constructor.

Returns
subdag – The newly-created sub-DAG.

Return type
SubDAG

parent_layer(*type=None*, ***kwargs*)

Create a new *NodeLayer* which is a parent of all of the nodes in this *Nodes*.

Parameters

- **type** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.
- **kwargs** – Additional keyword arguments are passed to the *NodeLayer* constructor.

Returns
node_layer – The newly-created node layer.

Return type
NodeLayer

parent_subdag(*type=None*, ***kwargs*)

Create a new *SubDAG* which is a parent of all of the nodes in this *Nodes*.

Parameters

- **type** (Optional[*BaseEdge*, None]) – The type of edge to use; an instance of a concrete subclass of *BaseEdge*. If None, a *ManyToMany* edge will be used.
- **kwargs** – Additional keyword arguments are passed to the *SubDAG* constructor.

Returns
subdag – The newly-created sub-DAG.

Return type
SubDAG

remove_children(**nodes*)

Makes sure that the nodes are **not** children of all of the nodes in this *Nodes*.

Parameters
nodes – The nodes to remove edges from.

Returns
self – This method returns self.

Return type
Nodes

remove_parents(**nodes*)

Makes sure that the `nodes` are **not** parents of any of the nodes in this *Nodes*.

Parameters
nodes – The nodes to remove edges from.

Returns
self – This method returns `self`.

Return type
Nodes

walk_ancestors(*order=WalkOrder.DEPTH_FIRST*)

Walk over all of the ancestors of all of the nodes in this *Nodes*, in the given order.

walk_descendants(*order=WalkOrder.DEPTH_FIRST*)

Walk over all of the descendants of all of the nodes in this *Nodes*, in the given order.

Edges

class htcondor.dags.**BaseEdge**

An abstract class that represents the edge between two logical nodes in the DAG.

abstract get_edges(*parent*, *child*, *join_factory*)

This abstract method is used by the writer to figure out which nodes in the parent and child should be connected by an actual DAGMan edge. It should yield (or simply return an iterable of) individual edge specifications.

Each edge specification is a tuple containing two elements: the first is a group of parent node indices, the second is a group of child node indices. Either (but not both) may be replaced by a special `JoinNode` object provided by `JoinFactory.get_join_node()`. An instance of this class is passed into this function by the writer; you should not create one yourself.

You may yield any number of edge specifications, but the more compact you can make the representation (i.e., fewer edge specifications, each with fewer elements), the better. This is where join nodes are helpful: they can turn "many-to-many" relationships into a significantly smaller number of actual edges ($2N$ instead of N^2).

A *SubDAG* or a zero-vars *NodeLayer* both implicitly have a single node index, `0`. See the source code of *ManyToMany* for a simple pattern for dealing with this.

Parameters

- **parent** (*BaseNode*) – The parent, a concrete subclass of *BaseNode*.
- **child** (*BaseNode*) – The child, a concrete subclass of *BaseNode*.
- **join_factory** (`JoinFactory`) – An instance of `JoinFactory` that will be provided by the writer.

Return type
`Iterable[Union[Tuple[Tuple[int], Tuple[int]], Tuple[Tuple[int], JoinNode], Tuple[JoinNode, Tuple[int]]]]`

class htcondor.dags.**OneToOne**

This edge connects two layers "linearly": each underlying node in the child layer is a child of the corresponding underlying node with the same index in the parent layer. The parent and child layers must have the same number of underlying nodes.

class htcondor.dags.**ManyToMany**

This edge connects two layers "densely": every node in the child layer is a child of every node in the parent layer.

class htcondor.dags.**Grouper**(*parent_chunk_size=1*, *child_chunk_size=1*)

This edge connects two layers in "chunks". The nodes in each layer are divided into chunks based on their respective chunk sizes (given in the constructor). Chunks are then connected like a *OneToOne* edge.

The number of chunks in each layer must be the same, and each layer must be evenly-divided into chunks (no leftover underlying nodes).

When both chunk sizes are 1 this is identical to a *OneToOne* edge, and you should use that edge instead because it produces a more compact representation.

Parameters

- **parent_chunk_size** (int) – The number of nodes in each chunk in the parent layer.
- **child_chunk_size** (int) – The number of nodes in each chunk in the child layer.

class htcondor.dags.**Slicer**(*parent_slice=slice(None, None, None)*, *child_slice=slice(None, None, None)*)

This edge connects individual nodes in the layers, selected by slices. Each node from the parent layer that is in the parent slice is joined, one-to-one, with the matching node from the child layer that is in the child slice.

Parameters

- **parent_slice** (slice) – The slice to use for the parent layer.
- **child_slice** (slice) – The slice to use for the child layer.

Node Configuration

class htcondor.dags.**Script**(*executable*, *arguments=None*, *retry=False*, *retry_status=1*, *retry_delay=0*)

Parameters

- **executable** (Union[str, Path]) – The path to the executable to run.
- **arguments** (Optional[List[str], None]) – The individual arguments to the executable. Keep in mind that these are evaluated as soon as the *Script* is created!
- **retry** (bool) – True if the script can be retried on failure.
- **retry_status** (int) – If the script exits with this status, the script run will be considered a failure for the purposes of retrying.
- **retry_delay** (int) – The number of seconds to wait after a script failure before retrying.

class htcondor.dags.**DAGAbortCondition**(*node_exit_value*, *dag_return_value=None*)

Represents the configuration of a node's DAG abort condition.

See *ABORT-DAG-ON* for more information about DAG aborts.

Parameters

- **node_exit_value** (int) – If the underlying node exits with this value, the DAG will be aborted.

- **dag_return_value** (Optional[int, None]) – If the DAG is aborted via this condition, it will exit with this code, if given. If not given, it will exit with the same return value that the node did.

Writing a DAG to Disk

htcondor.dags.**write_dag**(*dag*, *dag_dir*, *dag_file_name='dagfile.dag'*, *node_name_formatter=None*)

Write out the given DAG to the given directory. This includes the DAG description file itself, as well as any associated submit descriptions.

Parameters

- **dag** (*DAG*) – The DAG to write the description for.
- **dag_dir** (Path) – The directory to write the DAG files to.
- **dag_file_name** (Optional[str, None]) – The name of the DAG description file itself.
- **node_name_formatter** (Optional[*NodeNameFormatter*, None]) – The *NodeNameFormatter* to use for generating underlying node names. If not provided, the default is *SimpleFormatter*.

Returns

dag_file_path – The path to the DAG description file; can be passed to *htcondor.Submit.from_dag()* if you convert it to a string, like `Submit.from_dag(str(write_dag(...)))`.

Return type

pathlib.Path

class htcondor.dags.**NodeNameFormatter**

An abstract base class that represents a certain way of formatting and parsing underlying node names.

abstract generate(*layer_name*, *node_index*)

This method should generate a single node name, given the name of the layer and the index of the underlying node inside the layer.

Return type

str

abstract parse(*node_name*)

This method should convert a single node name back into a layer name and underlying node index. Node names must be invertible for *rescue()* to work.

Return type

Tuple[str, int]

class htcondor.dags.**SimpleFormatter**(*separator=':'*, *index_format='{:d}'*, *offset=0*)

A no-frills *NodeNameFormatter* that produces underlying node names like `LayerName-5`.

8.6.2 DAG Configuration

class htcondor.dags.**DotConfig**(*path*, *update=False*, *overwrite=True*, *include_file=None*)

A *DotConfig* holds the configuration options for whether and how DAGMan will produce a DOT file representing its execution graph.

See *Visualizing DAGs* for more information.

Parameters

- **path** (Path) – The path to write the DOT file to.
- **update** (bool) – If True, the DOT file will be updated as the DAG executes. If False, it will be written once at startup.
- **overwrite** (bool) – If True, the DOT file will be updated in-place. If False, new DOT files will be created next to the original.
- **include_file** (Optional[Path, None]) – Include the contents of the file at this path in the DOT file.

class htcondor.dags.**NodeStatusFile**(*path*, *update_time=None*, *always_update=False*)

A *NodeStatusFile* holds the configuration options for whether and how DAGMan will write a file containing node status information.

See *Capturing the Status of Nodes in a File* for more information.

Parameters

- **path** (Path) – The path to write the node status file to.
- **update_time** (Optional[int, None]) – The minimum interval to write new information to the node status file.
- **always_update** (Optional[bool, None]) – Always update the node status file after the update_time, even if there are no changes from the previous update.

8.6.3 Rescue DAGs

htcondor.dags can read information from a DAGMan rescue file and apply it to your DAG as it is being constructed.

See *The Rescue DAG* for more information on Rescue DAGs.

htcondor.dags.**rescue**(*dag*, *rescue_file*, *formatter=None*)

Applies state recorded in a DAGMan rescue file to the dag. The dag will be modified in-place.

Warning: Running this function on a *DAG* **replaces** any existing DONE information on **all** of its nodes. Every node will have a dictionary for its done attribute. If you want to edit this information manually, always run this function **first**, then make the desired changes on top.

Warning: This function cannot detect changes in node names. If node names are different in the rescue file compared to the *DAG*, this function will not behave as expected.

Parameters

- **dag** (*DAG*) – The DAG to apply the rescue state to.

- **rescue_file** (Path) – The file to get rescue state from. Use the *find_rescue_file()* helper function to find the right rescue file.
- **formatter** (Optional[*NodeNameFormatter*, None]) – The node name formatter that was used to write out the original DAG.

Return type
None

htcondor.dags.**find_rescue_file**(*dag_dir*, *dag_file_name='dagfile.dag'*)

Finds the latest rescue file in a DAG directory (just like DAGMan itself would).

Parameters

- **dag_dir** (Path) – The directory to search in.
- **dag_file_name** (str) – The base name of the DAG description file; the same name you would pass to *write_dag()*.

Returns
rescue_file – The path to the latest rescue file found in the dag_dir.

Return type
pathlib.Path

8.7 htcondor.personal API Reference

class htcondor.personal.**PersonalPool**(*local_dir=None*, *config=None*, *raw_config=None*, *detach=False*, *use_config=True*)

A *PersonalPool* is responsible for managing the lifecycle of a personal HTCondor pool. It can be used to start and stop a personal pool, and can also "attach" to an existing personal pool that is already running.

Parameters

- **local_dir** (Optional[Path, None]) – The local directory for the personal HTCondor pool. All configuration and state for the personal pool will be stored in this directory.
- **config** (Mapping[str, str]) – HTCondor configuration parameters to inject, as a mapping of key-value pairs.
- **raw_config** (Optional[str, None]) – Raw HTCondor configuration language to inject, as a string.
- **detach** (bool) – If True, the personal HTCondor pool will not be shut down when this object is destroyed (e.g., by stopping Python). Defaults to False.
- **use_config** (bool) – If True, the environment variable CONDOR_CONFIG will be set during initialization, such that this personal pool appears to be the local HTCondor pool for all operations in this Python session, even ones that don't go through the *PersonalPool* object. The personal pool will also be initialized. Defaults to True.

classmethod attach(*local_dir=None*)

Make a new *PersonalPool* attached to an existing personal pool that is already running in local_dir.

Parameters
local_dir (Optional[Path, None]) – The local directory for the existing personal pool.

Returns
self – This method returns self.

Return type
PersonalPool

property collector

The *htcondor.Collector* for the personal pool's collector.

detach()

Detach the personal pool (as in the constructor argument), and return `self`.

Return type
PersonalPool

get_config_val(*macro*, *default=None*)

Get the value of a configuration macro. The value will be "evaluated", meaning that other configuration macros or functions inside it will be expanded.

Parameters

- **macro** (`str`) – The configuration macro to look up the value for.
- **default** (`Optional[str, None]`) – If not `None`, and the config macro has no value, return this instead. If `None`, a `KeyError` will be raised instead.

Returns
value – The evaluated value of the configuration macro.

Return type
str

initialize(*overwrite_config=True*)

Initialize the personal pool by creating its local directory and writing out configuration files.

The contents of the local directory (except for the configuration file if `overwrite_config=True`) will not be overridden.

Parameters
overwrite_config – If `True`, the existing configuration file will be overwritten with the configuration set up in the constructor. If `False` and there is an existing configuration file, an exception will be raised. Defaults to `True`.

Returns
self – This method returns `self`.

Return type
PersonalPool

run_command(*args*, *stdout=-1*, *stderr=-1*, *universal_newlines=True*, ***kwargs*)

Execute a command in a subprocess against this personal pool, using `subprocess.run()` with good defaults for executing HTCondor commands. All of the keyword arguments of this function are passed directly to `subprocess.run()`.

Parameters

- **args** (`List[str]`) – The command to run, and its arguments, as a list of strings.
- **kwargs** – All keyword arguments (including `stdout`, `stderr`, and `universal_newlines`) are passed to `subprocess.run()`.

Returns
completed_process

Return type
subprocess.CompletedProcess

property schedd

The *htcondor.Schedd* for the personal pool's schedd.

start()

Start the personal condor (bringing it to the READY state from either UNINITIALIZED or INITIALIZED).

Returns
self – This method returns self.

Return type
PersonalPool

property state

The current *PersonalPoolState* of the personal pool.

stop()

Stop the personal condor, bringing it from the READY state to STOPPED.

Returns
self – This method returns self.

Return type
PersonalPool

use_config()

Returns a *SetCondorConfig* context manager that sets CONDOR_CONFIG to point to the configuration file for this personal pool.

who()

Return the result of condor_who -quick, as a *classad.ClassAd*. If condor_who -quick fails, or the output can't be parsed into a sensible who ad, this method returns an empty ad.

Return type
ClassAd

class htcondor.personal.**PersonalPoolState**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: str, Enum

An enumeration of the possible states that a *PersonalPool* can be in.

UNINITIALIZED = 'UNINITIALIZED'

INITIALIZED = 'INITIALIZED'

STARTING = 'STARTING'

READY = 'READY'

STOPPING = 'STOPPING'

STOPPED = 'STOPPED'

class htcondor.personal.**SetCondorConfig**(*config_file*)

A context manager. Inside the block, the Condor config file is the one given to the constructor. After the block, it is reset to whatever it was before the block was entered.

Parameters
config_file (Path) – The path to an HTCondor configuration file.

CHAPTER

NINE

CHIRP: JOBS WRITING USER DATA TO THE AP

Chirp is a set of commands that a running job can invoke on the EP to send or receive custom user data to or from the AP. It is one of the few HTCondor features that only runs in a running job on the EP.

Common uses for chirp include appending to the job event log to log on the AP the completion percentage of the job. Or, say, a job has three different phases: preparation, activity, and cleanup. With chirp, the job can ask HTCondor to append an event to the job event log informing the AP and the user there what phase the job has entered. For example, a running job could run the command line tool:

```
$ /usr/libexec/condor_chirp ulog "I have reached stage 3"
```

In addition to the user log, with chirp, the job can read from or write to the job's classad as it exists in the schedd. Note that a static copy of the job ad, in the state that it existed at job startup is dropped into the job's scratch directory. You can find this file by inspecting the environment variable $_CONDOR_JOB_AD. But to see attributes which have been updated on the AP after the job has started, including attributes which may have been changed with the *condor_qedit* command, you will need to use chirp:

```
$ /usr/libexec/condor_chirp set_job_ad_attr MyCurrentStatus '"Stage 3"'
```

As always with passing classad expressions or values through the shell, be careful with quoting. Also note that these commands don't need to, and indeed can not pass the job cluster or proc id as an argument – the job is implicitly the one that is running, and chirp cannot write to any other job.

As there is some cost to writing to the instance of the job ad inside the schedd, chirp also supports delayed job ad updates. This is on by default, and any job ad attribute whose name begins with "Chirp" is considered a delayed updated. Any updates to these attributes will be batched together and send when the starter needs to send another update to the shadow, for any reasons, or when there are 100 (by default) pending delayed updates.

Chirp may be used from a command line tool, see the *condor_chirp* man page for full details.

Alternatively, python programs can natively run chirp commands, see the htchirp bindings for more details on this method.

This service is off by default; it may be enabled by placing in the submit description file:

```
want_io_proxy = True
```

This places the needed attribute into the job ClassAd.

The Chirp wire protocol used by the starter is fully documented at http://ccl.cse.nd.edu/software/chirp/.

CHAPTER

TEN

CLOUD COMPUTING

Although HTCondor has long supported accessing cloud resources as though they were part of the Grid, the differences between clouds and the Grid have made it difficult to convert access into utility; a job in the Grid universe starts a virtual machine, rather than the user's executable.

We offer two solutions to this problem. The first, a tool called *condor_annex*, helps users or administrators extend an existing HTCondor pool with cloud resources. The second is an easy way to create an entire HTCondor pool from scratch on the cloud, using our *Google Cloud Marketplace Entry*.

The rest of this chapter is concerned with using the *condor_annex* tool to add nodes to an existing HTCondor pool; it includes instructions on how to create a single-node HTCondor installation as a normal user so that you can expand it with cloud resources. It also discusses how to manually construct a *HTCondor in the Cloud* using *condor_annex*.

10.1 Introduction

To be clear, our concern throughout this chapter is with commercial services which rent computational resources over the Internet at short notice and charge in small increments (by the minute or the hour). Currently, the *condor_annex* tool supports only AWS. AWS can start booting a new virtual machine as quickly as a few seconds after the request; barring hardware failure, you will be able to continue renting that VM until you stop paying the hourly charge. The other cloud services are broadly similar.

If you already have access to the Grid, you may wonder why you would want to begin cloud computing. The cloud services offer two major advantages over the Grid: first, cloud resources are typically available more quickly and in greater quantity than from the Grid; and second, because cloud resources are virtual machines, they are considerably more customizable than Grid resources. The major disadvantages are, of course, cost and complexity (although we hope that *condor_annex* reduces the latter).

We illustrate these advantages with what we anticipate will be the most common uses for *condor_annex*.

10.1.1 Use Case: Deadlines

With the ability to acquire computational resources in seconds or minutes and retain them for days or weeks, it becomes possible to rapidly adjust the size - and cost - of an HTCondor pool. Giving this ability to the end-user avoids the problems of deciding who will pay for expanding the pool and when to do so. We anticipate that the usual cause for doing so will be deadlines; the end-user has the best knowledge of their own deadlines and how much, in monetary terms, it's worth to complete their work by that deadline.

10.1.2 Use Case: Capabilities

Cloud services may offer (virtual) hardware in configurations unavailable in the local pool, or in quantities that it would be prohibitively expensive to provide on an on-going basis. Examples (from 2017) may include GPU-based computation, or computations requiring a terabyte of main memory. A cloud service may also offer fast and cloud-local storage for shared data, which may have substantial performance benefits for some workflows. Some cloud providers (for example, AWS) have pre-populated this storage with common public datasets, to further ease adoption.

By using cloud resources, an HTCondor pool administrator may also experiment with or temporarily offer different software and configurations. For example, a pool may be configured with a maximum job runtime, perhaps to reduce the latency of fair-share adjustments or to protect against hung jobs. Adding cloud resources which permit longer-running jobs may be the least-disruptive way to accomodate a user whose jobs need more time.

10.1.3 Use Case: Capacities

It may be possible for an HTCondor administrator to lower the cost of their pool by increasing utilization and meeting peak demand with cloud computing.

10.1.4 Use Case: Experimental Convenience

Although you can experiment with many different HTCondor configurations using *condor_annex* and HTCondor running as a normal user, some configurations may require elevated privileges. In other situations, you may not be to create an unprivileged HTCondor pool on a machine because that would violate the acceptable-use policies, or because you can't change the firewall, or because you'd use too much bandwidth. In those cases, you can instead "seed" the cloud with a single-node HTCondor installation and expand it using *condor_annex*. See *HTCondor in the Cloud* for instructions.

10.2 HTCondor Annex User's Guide

A user of *condor_annex* may be a regular job submitter, or she may be an HTCondor pool administrator. This guide will cover basic *condor_annex* usage first, followed by advanced usage that may be of less interest to the submitter. Users interested in customizing *condor_annex* should consult the *HTCondor Annex Customization Guide*.

10.2.1 Considerations and Limitations

When you run *condor_annex*, you are adding (virtual) machines to an HTCondor pool. As a submitter, you probably don't have permission to add machines to the HTCondor pool you're already using; generally speaking, security concerns will forbid this. If you're a pool administrator, you can of course add machines to your pool as you see fit. By default, however, *condor_annex* instances will only start jobs submitted by the user who started the annex, so pool administrators using *condor_annex* on their users' behalf will probably want to use the **-owners** option or **-no-owner** flag; see the *condor_annex* man page. Once the new machines join the pool, they will run jobs as normal.

Submitters, however, will have to set up their own personal HTCondor pool, so that *condor_annex* has a pool to join, and then work with their pool administrator if they want to move their existing jobs to their new pool. Otherwise, jobs will have to be manually divided (removed from one and resubmitted to the other) between the pools. For instructions on creating a personal HTCondor pool, preparing an AWS account for use by *condor_annex*, and then configuring *condor_annex* to use that account, see the *Using condor_annex for the First Time* section.

Starting in v8.7.1, *condor_annex* will check for inbound access to the collector (usually port 9618) before starting an annex (it does not support other network topologies). When checking connectivity from AWS, the IP(s) used by the

AWS Lambda function implementing this check may not be in the same range(s) as those used by AWS instance; please consult AWS's list of all their IP[2] when configuring your firewall.

Starting in v8.7.2, *condor_annex* requires that the AWS secret (private) key file be owned by the submitting user and not readable by anyone else. This helps to ensure proper attribution.

10.2.2 Basic Usage

This section assumes you're logged into a Linux machine an that you've already configured *condor_annex*. If you haven't, see the *Using condor_annex for the First Time* section.

All the terminal commands (shown in a box without a title) and file edits (shown in a box with an emphasized filename for a title) in this section take place on the Linux machine. In this section, we follow the common convention that the commands you type are preceded by by '$' to distinguish them from any expected output; don't copy that part of each of the following lines. (Lines which end in a '\' continue on the following line; be sure to copy both lines. Don't copy the '\' itself.)

What You'll Need to Know

To create a HTCondor annex with on-demand instances, you'll need to know two things:

1. A name for it. "MyFirstAnnex" is a fine name for your first annex.
2. How many instances you want. For your first annex, when you're checking to make sure things work, you may only want one instance.

10.2.3 Start an Annex

Entering the following command will start an annex named "MyFirstAnnex" with one instance. *condor_annex* will print out what it's going to do, and then ask you if that's OK. You must type 'yes' (and hit enter) at the prompt to start an annex; if you do not, *condor_annex* will print out instructions about how to change whatever you may not like about what it said it was going to do, and then exit.

```
$ condor_annex -count 1 -annex-name MyFirstAnnex
Will request 1 m4.large on-demand instance for 0.83 hours. Each instance will
terminate after being idle for 0.25 hours.
Is that OK? (Type 'yes' or 'no'): yes
Starting annex...
Annex started. Its identity with the cloud provider is
'TestAnnex0_f2923fd1-3cad-47f3-8e19-fff9988ddacf'. It will take about three
minutes for the new machines to join the pool.
```

You won't need to know the annex's identity with the cloud provider unless something goes wrong.

Before starting the annex, *condor_annex* (v8.7.1 and later) will check to make sure that the instances will be able to contact your pool. Contact the Linux machine's administrator if *condor_annex* reports a problem with this step.

[2] https://ip-ranges.amazonaws.com/ip-ranges.json

Instance Types

Each instance type provides a different number (and/or type) of CPU cores, amount of RAM, local storage, and the like. We recommend starting with 'm4.large', which has 2 CPU cores and 8 GiB of RAM, but you can see the complete list of instance types at the following URL:

https://aws.amazon.com/ec2/instance-types/

You can specify an instance type with the -aws-on-demand-instance-type flag.

Leases

By default, *condor_annex* arranges for your annex's instances to be terminated after 0.83 hours (50 minutes) have passed. Once it's in place, this lease doesn't depend on the Linux machine, but it's only checked every five minutes, so give your deadlines a lot of cushion to make you don't get charged for an extra hour. The lease is intended to help you conserve money by preventing the annex instances from accidentally running forever. You can specify a lease duration (in decimal hours) with the -duration flag.

If you need to adjust the lease for a particular annex, you may do so by specifying an annex name and a duration, but not a count. When you do so, the new duration is set starting at the current time. For example, if you'd like "MyFirstAnnex" to expire eight hours from now:

```
$ condor_annex -annex-name MyFirstAnnex -duration 8
Lease updated.
```

Idle Time

By default, *condor_annex* will configure your annex's instances to terminate themselves after being idle for 0.25 hours (fifteen minutes). This is intended to help you conserve money in case of problems or an extended shortage of work. As noted in the example output above, you can specify a max idle time (in decimal hours) with the -idle flag. *condor_annex* considers an instance idle if it's unclaimed (see *condor_startd Policy Configuration* for a definition), so it won't get tricked by jobs with long quiescent periods.

Tagging your Annex's Instances

By default, *condor_annex* adds a tag, `htcondor:AnnexName`, to each instance in the annex; its value is the annex's name (as entered on the command line). You may add additional tags via the command-line option `-tag`, which must be followed by a tag name and a value for that tag (as separate arguments). You may specify any number of tags (up to the maximum supported by the cloud provider) by adding additional `-tag` options to the command line.

Starting Multiple Annexes

You may have up to fifty (or fewer, depending what else you're doing with your AWS account) differently-named annexes running at the same time. Running *condor_annex* again with the same annex name before stopping that annex will both add instances to it and change its duration. Only instances which start up after an invocation of *condor_annex* will respect that invocation's max idle time. That may include instances still starting up from your previous (first) invocation of *condor_annex*, so be sure your instances have all joined the pool before running *condor_annex* again with the same annex name if you're changing the max idle time. Each invocation of *condor_annex* requests a certain number of instances of a given type; you may specify the instance type, the count, or both with each invocation, but doing so does not change the instance type or count of any previous request.

10.2.4 Monitor your Annex

You can find out if an instance has successfully joined the pool in the following way:

```
$ condor_annex status
Name                                OpSys      Arch   State     Activity     Load

slot1@ip-172-31-48-84.ec2.internal LINUX      X86_64 Unclaimed Benchmarking  0.0
slot2@ip-172-31-48-84.ec2.internal LINUX      X86_64 Unclaimed Idle          0.0

Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

X86_64/LINUX     2     0       0         2       0          0        0      0
Total            2     0       0         2       0          0        0      0
```

This example shows that the annex instance you requested has joined your pool. (The default annex image configures one static slot for each CPU it finds on start-up.)

You may instead use *condor_status*:

```
$ condor_status -annex MyFirstAnnex
slot1@ip-172-31-48-84.ec2.internal  LINUX     X86_64 Unclaimed Idle 0.640 3767
slot2@ip-172-31-48-84.ec2.internal  LINUX     X86_64 Unclaimed Idle 0.640 3767

 Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain
X86_64/LINUX     2     0       0         2       0          0        0      0
Total            2     0       0         2       0          0        0      0
```

You can also get a report about the instances which have not joined your pool:

```
$ condor_annex -annex MyFirstAnnex -status
STATE          COUNT
pending            1
TOTAL              1
Instances not in the pool, grouped by state:
pending i-06928b26786dc7e6e
```

Monitoring Multiple Annexes

The following command reports on all annex instance which have joined the pool, regardless of which annex they're from:

```
$ condor_status -annex
slot1@ip-172-31-48-84.ec2.internal  LINUX     X86_64 Unclaimed Idle 0.640 3767
slot2@ip-172-31-48-84.ec2.internal  LINUX     X86_64 Unclaimed Idle 0.640 3767
slot1@ip-111-48-85-13.ec2.internal  LINUX     X86_64 Unclaimed Idle 0.640 3767
slot2@ip-111-48-85-13.ec2.internal  LINUX     X86_64 Unclaimed Idle 0.640 3767

Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain
X86_64/LINUX     4     0       0         4       0          0        0      0
Total            4     0       0         4       0          0        0      0
```

The following command reports about instance which have not joined the pool, regardless of which annex they're from:

```
$ condor_annex -status
NAME                        TOTAL running
NamelessTestA                   2       2
NamelessTestB                   3       3
NamelessTestC                   1       1

NAME                        STATUS  INSTANCES...
NamelessTestA               running i-075af9ccb40efb162 i-0bc5e90066ed62dd8
NamelessTestB               running i-02e69e85197f249c2 i-0385f59f482ae6a2e
 i-06191feb755963edd
NamelessTestC               running i-09da89d40cde1f212
```

The ellipsis in the last column (INSTANCES...) is to indicate that it's a very wide column and may wrap (as it has in the example), not that it has been truncated.

The following command combines these two reports:

```
$ condor_annex status
Name                                OpSys      Arch   State     Activity     Load

slot1@ip-172-31-48-84.ec2.internal LINUX      X86_64 Unclaimed Benchmarking  0.0
slot2@ip-172-31-48-84.ec2.internal LINUX      X86_64 Unclaimed Idle          0.0

Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

X86_64/LINUX     2     0       0         2       0          0        0      0
Total            2     0       0         2       0          0        0      0

Instance ID         not in Annex  Status  Reason (if known)
i-075af9ccb40efb162 NamelessTestA running -
i-0bc5e90066ed62dd8 NamelessTestA running -
i-02e69e85197f249c2 NamelessTestB running -
i-0385f59f482ae6a2e NamelessTestB running -
i-06191feb755963edd NamelessTestB running -
i-09da89d40cde1f212 NamelessTestC running -
```

10.2.5 Run a Job

Starting in v8.7.1, the default behaviour for an annex instance is to run only jobs submitted by the user who ran the *condor_annex* command. If you'd like to allow other users to run jobs, list them (separated by commas; don't forget to include yourself) as arguments to the -owner flag when you start the instance. If you're creating an annex for general use, use the -no-owner flag to run jobs from anyone.

Also starting in v8.7.1, the default behaviour for an annex instance is to run only jobs which have the MayUseAWS attribute set (to true). To submit a job with MayUseAWS set to true, add `+MayUseAWS = TRUE` to the submit file somewhere before the queue command. To allow an existing job to run in the annex, use condor_q_edit. For instance, if you'd like cluster 1234 to run on AWS:

```
$ condor_qedit 1234 "MayUseAWS = TRUE"
Set attribute "MayUseAWS" for 21 matching jobs.
```

10.2.6 Stop an Annex

The following command shuts HTCondor off on each instance in the annex; if you're using the default annex image, doing so causes each instance to shut itself down. HTCondor does not provide a direct method terminating *condor_annex* instances.

```
$ condor_off -annex MyFirstAnnex
Sent "Kill-Daemon" command for "master" to master ip-172-31-48-84.ec2.internal
```

Stopping Multiple Annexes

The following command turns off all annex instances in your pool, regardless of which annex they're from:

```
$ condor_off -annex
Sent "Kill-Daemon" command for "master" to master ip-172-31-48-84.ec2.internal
Sent "Kill-Daemon" command for "master" to master ip-111-48-85-13.ec2.internal
```

10.2.7 Using Different or Multiple AWS Regions

It sometimes advantageous to use multiple AWS regions, or convenient to use an AWS region other than the default, which is `us-east-1`. To change the default, set the configuration macro ANNEX_DEFAULT_AWS_REGION to the new default. (If you used the *condor_annex* automatic setup, you can edit the `user_config` file in `.condor directory` in your home directory; this file uses the normal HTCondor configuration file syntax. (See *Ordered Evaluation to Set the Configuration*.) Once you do this, you'll have to re-do the setup, as setup is region-specific.

If you'd like to use multiple AWS regions, you can specify which reason to use on the command line with the **-aws-region** flag. Each region may have zero or more annexes active simultaneously.

10.2.8 Advanced Usage

The previous section covered using what AWS calls "on-demand" instances. (An "instance" is "a single occurrence of something," in this case, a virtual machine. The intent is to distinguish between the active process that's pretending to be a real piece of hardware - the "instance" - and the template it used to start it up, which may also be called a virtual machine.) An on-demand instance has a price fixed by AWS; once acquired, AWS will let you keep it running as long as you continue to pay for it.

In constrast, a "Spot" instance has a price determined by an (automated) auction; when you request a "Spot" instance, you specify the most (per hour) you're willing to pay for that instance. If you get an instance, however, you pay only what the spot price is for that instance; in effect, AWS determines the spot price by lowering it until they run out of instances to rent. AWS advertises savings of up to 90% over on-demand instances.

There are two drawbacks to this cheaper type of instance: first, you may have to wait (indefinitely) for instances to become available at your preferred price-point; the second is that your instances may be taken away from you before you're done with them because somebody else will pay more for them. (You won't be charged for the hour in which AWS kicks you off an instance, but you will still owe them for all of that instance's previous hours.) Both drawbacks can be mitigated (but not eliminated) by bidding the on-demand price for an instance; of course, this also minimizes your savings.

Determining an appropriate bidding strategy is outside the purview of this manual.

Using AWS Spot Fleet

condor_annex supports Spot instances via an AWS technology called "Spot Fleet". Normally, when you request instances, you request a specific type of instance (the default on-demand instance is, for instance, 'm4.large'.) However, in many cases, you don't care too much about how many cores an intance has - HTCondor will automatically advertise the right number and schedule jobs appropriately, so why would you? In such cases - or in other cases where your jobs will run acceptably on more than one type of instance - you can make a Spot Fleet request which says something like "give me a thousand cores as cheaply as possible", and specify that an 'm4.large' instance has two cores, while 'm4.xlarge' has four, and so on. (The interface actually allows you to assign arbitrary values - like HTCondor slot weights - to each instance type[1], but the default value is core count.) AWS will then divide the current price for each instance type by its core count and request spot instances at the cheapest per-core rate until the number of cores (not the number of instances!) has reached a thousand, or that instance type is exhausted, at which point it will request the next-cheapest instance type.

(At present, a Spot Fleet only chooses the cheapest price within each AWS region; you would have to start a Spot Fleet in each AWS region you were willing to use to make sure you got the cheapest possible price. For fault tolerance, each AWS region is split into independent zones, but each zone has its own price. Spot Fleet takes care of that detail for you.)

In order to create an annex via a Spot Fleet, you'll need a file containing a JSON blob which describes the Spot Fleet request you'd like to make. (It's too complicated for a reasonable command-line interface.) The AWS web console can be used to create such a file; the button to download that file is (currently) in the upper-right corner of the last page before you submit the Spot Fleet request; it is labeled 'JSON config'. You may need to create an IAM role the first time you make a Spot Fleet request; please do so before running *condor_annex*.

- You must select the instance role profile used by your on-demand instances for *condor_annex* to work. This value will have been stored in the configuration macro `ANNEX_DEFAULT_ODI_INSTANCE_PROFILE_ARN` by the setup procedure.
- You must select a security group which allows inbound access on HTCondor's port (9618) for *condor_annex* to work. You may use the value stored in the configuration macro `ANNEX_DEFAULT_ODI_SECURITY_GROUP_IDS` by the setup procedure; this security group also allows inbound SSH access.
- If you wish to be able to SSH to your instances, you must select an SSH key pair (for which you have the corresponding private key); this is not required for *condor_ssh_to_job*. You may use the value stored in the configuration macro `ANNEX_DEFAULT_ODI_KEY_NAME` by the setup procedure.

Specify the JSON configuration file using **-aws-spot-fleet-config-file**, or set the configuration macro ANNEX_DEFAULT_SFR_CONFIG_FILE to the full path of the file you just downloaded, if you'd like it to become your default configuration for Spot annexes. Be aware that *condor_annex* does not alter the validity period if one is set in the Spot Fleet configuration file. You should remove the references to 'ValidFrom' and 'ValidTo' in the JSON file to avoid confusing surprises later.

Additionally, be aware that *condor_annex* uses the Spot Fleet API in its "request" mode, which means that an annex created with Spot Fleet has the same semantics with respect to replacement as it would otherwise: if an instance terminates for any reason, including AWS taking it away to give to someone else, it is not replaced.

You must specify the number of cores (total instance weight; see above) using **-slots**. You may also specify **-aws-spot-fleet**, if you wish; doing so may make this *condor_annex* invocation more self-documenting. You may use other options as normal, excepting those which begin with **-aws-on-demand**, which indicates an option specific to on-demand instances.

[1] Strictly speaking, to each "launch specification"; see the explanation below, in the section AWS Instance User Data.

Custom HTCondor Configuration

When you specify a custom configuration, you specify the full path to a configuration directory which will be copied to the instance. The customizations performed by *condor_annex* will be applied to a temporary copy of this directory before it is uploaded to the instance. Those customizations consist of creating two files: password_file.pl (named that way to ensure that it isn't ever accidentally treated as configuration), and 00ec2-dynamic.config. The former is a password file for use by the pool password security method, which if configured, will be used by *condor_annex* automatically. The latter is an HTCondor configuration file; it is named so as to sort first and make it easier to over-ride with whatever configuration you see fit.

AWS Instance User Data

HTCondor doesn't interfere with this in any way, so if you'd like to set an instance's user data, you may do so. However, as of v8.7.2, the **-user-data** options don't work for on-demand instances (the default type). If you'd like to specify user data for your Spot Fleet -driven annex, you may do so in four different ways: on the command-line or from a file, and for all launch specifications or for only those launch specifications which don't already include user data. These two choices correspond to the absence or presence of a trailing **-file** and the absence or presence of **-default** immediately preceding **-user-data**.

A "launch specification," in this context, means one of the virtual machine templates you told Spot Fleet would be an acceptable way to accomodate your resource request. This usually corresponds one-to-one with instance types, but this is not required.

Expert Mode

The *condor_annex* manual page lists the "expert mode" options.

Four of the "expert mode" options set the URLs used to access AWS services, not including the CloudFormation URL needed by the **-setup** flag. You may change the CloudFormation URL by changing the HTCondor configuration macro ANNEX_DEFAULT_CF_URL , or by supplying the URL as the third parameter after the **-setup** flag. If you change any of the URLs, you may need to change all of the URLs - Lambda functions and CloudWatch events in one region don't work with instances in another region.

You may also temporarily specify a different AWS account by using the access (**-aws-access-key-file**) and secret key (**-aws-secret-key-file**) options. Regular users may have an accounting reason to do this.

The options labeled "developers only" control implementation details and may change without warning; they are probably best left unused unless you're a developer.

10.3 Using *condor_annex* for the First Time

This guide assumes that you already have an AWS account, as well as a log-in account on a Linux machine with a public address and a system administrator who's willing to open a port for you. All the terminal commands (shown in a box) and file edits (show in a box whose first line begins with a # and names a file) take place on the Linux machine. You can perform the web-based steps from wherever is convenient, although it will save you some copying if you run the browser on the Linux machine.

If your Linux machine will be an EC2 instance, read *Using Instance Credentials* first; by taking some care in how you start the instance, you can save yourself some drudgery.

Before using *condor_annex* for the first time, you'll have to do three things:

1. install a personal HTCondor
2. prepare your AWS account

3. configure *condor_annex*

Instructions for each follow.

10.3.1 Install a Personal HTCondor

We recommend that you install a personal HTCondor to make use of *condor_annex*; it's simpler to configure that way. Follow the *Hand-Installation of HTCondor on a Single Machine with User Privileges* instructions. Make sure you install HTCondor version 8.7.8 or later.

Once you have a working personal HTCondor installation, continue with the additional setup instructions below, that are specific to using *condor_annex*.

> In the following instructions, it is assumed that the local installation has been done in the folder `~/condor-8.7.8`. Change this path depending on your HTCondor version and how you followed the installation instructions.

Configure Public Interface

The default personal HTCondor uses the "loopback" interface, which basically just means it won't talk to anyone other than itself. For *condor_annex* to work, your personal HTCondor needs to use the Linux machine's public interface. In most cases, that's as simple as adding the following lines:

```
# ~/condor-8.7.8/local/condor_config.local

NETWORK_INTERFACE = *
CONDOR_HOST = $(FULL_HOSTNAME)
```

Restart HTCondor to force the changes to take effect:

```
$ condor_restart
Sent "Restart" command to local master
```

To verify that this change worked, repeat the steps under the *Install a Personal HTCondor* section. Then proceed onto the next section.

Configure a Pool Password

In this section, you'll configure your personal HTCondor to use a pool password. This is a simple but effective method of securing HTCondor's communications to AWS.

Add the following lines:

```
# ~/condor-8.7.8/local/condor_config.local

SEC_PASSWORD_FILE = $(LOCAL_DIR)/condor_pool_password

SEC_DAEMON_INTEGRITY = REQUIRED
SEC_DAEMON_AUTHENTICATION = REQUIRED
SEC_DAEMON_AUTHENTICATION_METHODS = PASSWORD
SEC_NEGOTIATOR_INTEGRITY = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION_METHODS = PASSWORD
```

(continues on next page)

(continued from previous page)

```
SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD
ALLOW_DAEMON = condor_pool@*
```

You also need to run the following command, which prompts you to enter a password:

```
$ condor_store_cred -c add -f `condor_config_val SEC_PASSWORD_FILE`
Enter password:
```

Enter a password.

Tell HTCondor about the Open Port

By default, HTCondor will use port 9618. If the Linux machine doesn't already have HTCondor installed, and the admin is willing to open that port, then you don't have to do anything. Otherwise, you'll need to add a line like the following, replacing '9618' with whatever port the administrator opened for you.

```
# ~/condor-8.7.8/local/condor_config.local

COLLECTOR_HOST = $(FULL_HOSTNAME):9618
```

Activate the New Configuration

Force HTCondor to read the new configuration by restarting it:

```
$ condor_restart
```

10.3.2 Prepare your AWS account

Since v8.7.1, the *condor_annex* tool has included a -setup command which will prepare your AWS account.

Using Instance Credentials

If you will not be running *condor_annex* on an EC2 instance, skip to *Obtaining an Access Key*.

When you start an instance on EC2[1], you can grant it some of your AWS privileges, for instance, for starting instances. This (usually) means that any user logged into the instance can, for instance, start instances (as you). A given collection of privileges is called an "instance profile"; a full description of them is outside the scope of this document. If, however, you'll be the only person who can log into the instance you're creating and on which you will be running *condor_annex*, it may be simpler to start an instance with your privileges than to deal with *Obtaining an Access Key*.

You will need a privileged instance profile; if you don't already have one, you will only need to create it once. When launching an instance with the EC2 console, step 3 (labelled 'Configure Instance Details') includes an entry for 'IAM role'; the AWS web interface creates the corresponding instance profile for you automatically. If you've already created a privileged role, select it here and carry on launching your instance as usual. If you haven't:

1. Follow the 'Create new IAM role' link.
2. Click the 'Create Role' button.

[1] You may assign an intance profile to an EC2 instance when you launch it, or at any subsequent time, through the AWS web console (or other interfaces with which you may be familiar). If you start the instance using HTCondor's EC2 universe, you may specify the IAM instance profile with the **ec2_iam_profile_name** or **ec2_iam_profile_arn** submit commands.

3. Select 'EC2' under "the service that will use this role".
4. Click the 'Next: Permissions' button.
5. Select 'Administrator Access' and click the 'Next: Tags' button.
6. Click the 'Next: Review' button.
7. Enter a role name; 'HTCondorAnnexRole' is fine.
8. Click the 'Create role' button.

When you switch back to the previous tab, you may need to click the circular arrow (refresh) icon before you can select the role name you entered in the second-to-last step.

If you'd like step-by-step instructions for creating a HTCondor-in-the-Cloud, see *HTCondor in the Cloud*.

You can skip to *Configure condor_annex* once you've completed these steps.

Obtaining an Access Key

In order to use AWS, *condor_annex* needs a pair of security tokens (like a user name and password). Like a user name, the "access key" is (more or less) public information; the corresponding "secret key" is like a password and must be kept a secret. To help keep both halves secret, *condor_annex* (and HTCondor) are never told these keys directly; instead, you tell HTCondor which file to look in to find each one.

Create those two files now; we'll tell you how to fill them in shortly. By convention, these files exist in your ~/.condor directory, which is where the -setup command will store the rest of the data it needs.

```
$ mkdir ~/.condor
$ cd ~/.condor
$ touch publicKeyFile privateKeyFile
$ chmod 600 publicKeyFile privateKeyFile
```

The last command ensures that only you can read or write to those files.

To donwload a new pair of security tokens for *condor_annex* to use, go to the IAM console at the following URL; log in if you need to:

https://console.aws.amazon.com/iam/home?region=us-east-1#/users

The following instructions assume you are logged in as a user with the privilege to create new users. (The 'root' user for any account has this privilege; other accounts may as well.)

1. Click the "Add User" button.
2. Enter name in the **User name** box; "annex-user" is a fine choice.
3. Click the check box labelled "Programmatic access".
4. Click the button labelled "Next: Permissions".
5. Select "Attach existing policies directly".
6. Type "AdministratorAccess" in the box labelled "Filter".
7. Click the check box on the single line that will appear below (labelled "AdministratorAccess").
8. Click the "Next: review" button (you may need to scroll down).
9. Click the "Create user" button.
10. From the line labelled "annex-user", copy the value in the column labelled "Access key ID" to the file publicKeyFile.

11. On the line labelled “annex-user”, click the “Show” link in the column labelled “Secret access key”; copy the revealed value to the file privateKeyFile.
12. Hit the “Close” button.

The ‘annex-user’ now has full privileges to your account.

10.3.3 Configure *condor_annex*

The following command will setup your AWS account. It will create a number of persistent components, none of which will cost you anything to keep around. These components can take quite some time to create; *condor_annex* checks each for completion every ten seconds and prints an additional dot (past the first three) when it does so, to let you know that everything’s still working.

```
$ condor_annex -setup
Creating configuration bucket (this takes less than a minute)....... complete.
Creating Lambda functions (this takes about a minute)........ complete.
Creating instance profile (this takes about two minutes)................... complete.
Creating security group (this takes less than a minute)..... complete.
Setup successful.
```

Checking the Setup

You can verify at this point (or any later time) that the setup procedure completed successfully by running the following command.

```
$ condor_annex -check-setup
Checking for configuration bucket... OK.
Checking for Lambda functions... OK.
Checking for instance profile... OK.
Checking for security group... OK.
```

You’re ready to run *condor_annex*!

Undoing the Setup Command

There is not as yet a way to undo the setup command automatically, but it won’t cost you anything extra to leave your account setup for *condor_annex* indefinitely. If, however, you want to be tidy, you may delete the components setup created by going to the CloudFormation console at the following URL and deleting the entries whose names begin with ‘HTCondorAnnex-‘:

https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks?filter=active

The setup procedure also creates an SSH key pair which may be useful for debugging; the private key was stored in ~/.condor/HTCondorAnnex-KeyPair.pem. To remove the corresponding public key from your AWS account, go to the key pair console at the following URL and delete the ‘HTCondorAnnex-KeyPair’ key:

https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#KeyPairs:sort=keyName

10.4 HTCondor Annex Customization Guide

Aside from the configuration macros (see the *HTCondor Annex Configuration* section), the major way to ustomize *condor_annex* is my customizing the default disk image. Because the implementation of *condor_annex* varies from service to service, and that implementation determines the constraints on the disk image, the this section is divided by service.

10.4.1 Amazon Web Services

Requirements for an Annex-compatible AMI are driven by how *condor_annex* securely transports HTCondor configuration and security tokens to the instances; we will discuss that implementation briefly, to help you understand the requirements, even though it will hopefully never matter to you.

Resource Requests

For on-demand or Spot instances, we begin by making a single resource request whose client token is the annex name concatenated with an underscore and then a newly-generated GUID. This construction allows us to terminate on-demand instances belonging to a particular annex (by its name), as well as discover the annex name from inside an instance.

An on-demand instance may obtain its instance ID directly from the AWS metadata server, and then ask another AWS API for that instance ID's client token. Since GUIDs do not contain underscores, we can be certain that anything to the left of the last underscore is the annex's name.

An instance started by a Spot Fleet has a client token generated by the Spot Fleet. Instead of performing a direct lookup, a Spot Fleet instance must therefore determine which Spot Fleet started it, and then obtain that Spot Fleet's client token. A Spot Fleet will tag an instance with the Spot Fleet's identity after the instance starts up. This usually only takes a few minutes, but the default image waits for up to 50 minutes, since you're already paying for the first hour anyway.

Secure Transport

At this point, the instance knows its annex's name. This allows the instance to construct the name of the tarball it should download (config-AnnexName.tar.gz), but does not tell it from where a file with that name should be downloaded.

(Because the user data associated with resource request is not secure, and because we want to leave the user data available for its normal usage, we can't just encode the tarball or its location in the user data.)

The instance determines from which S3 bucket to download by asking the metadata server which role the instance is playing. (An instance without a role is unable to make use of any AWS services without acquiring valid AWS tokens through some other method.) The instance role created by the setup procedure includes permission to read files matching the pattern config-*.tar.gz from a particular private S3 bucket. If the instance finds permissions matching that pattern, it assumes that the corresponding S3 bucket is the one from which it should download, and does so; if successful, it untars the file in `/etc/condor/config.d`.

In v8.7.1, the script executing these steps is named `49ec2-instance.sh`, and is called during configuration when HTCondor first starts up.

In v8.7.2, the script executing these steps is named `condor-annex-ec2`, and is called during system start-up.

The HTCondor configuration and security tokens are at this point protected on the instance's disk by the usual filesystem permissions. To prevent HTCondor jobs from using the instance's permissions to do anything, but in particular download their own copy of the security tokens, the last thing the script does is use the Linux kernel firewall to forbid any non-root process from accessing the metadata server.

Image Requirements

Thus, to work with *condor_annex*, an AWS AMI must:

- Fetch the HTCondor configuration and security tokens from S3;
- configure HTCondor to turn off after it's been idle for too long;
- and turn off the instance when the HTCondor master daemon exits.

The second item could be construed as optional, but if left unimplemented, will disable the **-idle** command-line option.

The default disk image implements the above as follows:

- with a configuration script (/etc/condor/49ec2-instance.sh);
- with a single configuration item (`STARTD_NOCLAIM_SHUTDOWN`);
- with a configuration item (`DEFAULT_MASTER_SHUTDOWN_SCRIPT`) and the corresponding script (/etc/condor/master_shutdown.sh), which just turns around and runs shutdown -h now.

We also strongly recommend that every *condor_annex* disk image:

- Advertise, in the master and startd, the instance ID.
- Use the instance's public IP, by setting `TCP_FORWARDING_HOST` .
- Turn on communications integrity and encryption.
- Encrypt the run directories.
- Restrict access to the EC2 meta-data server to root.

The default disk image is configured to do all of this.

Instance Roles

To explain the last point immediately above, EC2 stores (temporary) credentials for the role, if any, associated with an instance on that instance's meta-data server, which may be accessed via HTTP at a well-known address (currently `169.254.169.254`). Unless otherwise configured, any process in the instance can access the meta-data server and thereby make use of the instance's credentials.

Until version 8.9.0, there was no HTCondor-based reason to run an EC2 instance with an instance role. Starting in 8.9.0, however, HTCondor gained the ability to use the instance role's credentials to run EC2 universe jobs and *condor_annex* commands. This has several advantages over copying credentials into the instance: it may be more convenient, and if you're the only user of the instance, it's more secure, because the instance's credentials expire when the instance does.

However, wanting to allow (other) users to run jobs on or submit jobs to your instance may not mean you want them to able to act with the instance's privileges (e.g., starting more instances on your account). Although securing your instances ultimately remains your responsibility, the default images we provide for *condor_annex*, and the condor-annex-ec2 package, both use the kernel-level firewall to prevent access to the metadata server by any process not owned by root. Because this firewall rule is added during the boot sequence, it will be in place before HTCondor can start any user jobs, and should therefore be effective in preventing access to the instance's credentials by normal users or their jobs.

10.5 HTCondor Annex Configuration

While the configuration macros in this section may be set by the HTCondor administrator, they are intended for the user-specific HTCondor configuration file (usually `~/.condor/user_config`). Although we document every macro, we expect that users will generally only want to change a few of them, listed in the *User Settings* section; the entries required in by *condor_annex* in other sections will be generated by its setup procedure.

Subsequent sections deal with logging (*Logging*), are for expert users (*Expert Settings*), or for HTCondor developers (*Developer Settings*).

10.5.1 User Settings

ANNEX_DEFAULT_AWS_REGION
The default region when using AWS. Defaults to 'us-east-1'.

ANNEX_DEFAULT_LEASE_DURATION
The duration of an annex if not specified on the command-line; specified in seconds. Defaults to 50 minutes.

ANNEX_DEFAULT_UNCLAIMED_TIMEOUT
How long an annex instances should stay idle before shutting down; specified in seconds. Defaults to 15 minutes.

ANNEX_DEFAULT_ODI_KEY_NAME
The name of the SSH key pair *condor_annex* should use by default. No default.

ANNEX_DEFAULT_ODI_INSTANCE_TYPE
The AWS instance type to use for on-demand instances if not specified. No default, but the *condor_annex* setup procedure sets this to 'm4.large'.

ANNEX_DEFAULT_ODI_IMAGE_ID
The AWS AMI to use for on-demand instance if not specified. No default, but the *condor_annex* setup procedure sets this to 'ami-35b13223'.

ANNEX_DEFAULT_SFR_CONFIG_FILE
The JSON configuration file use by *condor_annex* when creating a Spot-based annex. No default.

10.5.2 Logging

By default, running *condor_annex* creates three logs: the *condor_annex* log, the annex GAHP log, and the annex audit log. The default location for these logs is the same directory as the user-specific HTCondor configuration file (usually ~/.condor/user_config). *condor_annex* sets the `LOG` macro to this directory when reading its configuration.

The *condor_annex* log is a daemon-style log. It is configured as if *condor_annex* were a daemon with subsystem type `ANNEX`; see *Daemon Logging Configuration File Entries* for details.

condor_annex uses special helper programs, called GAHPs, to interact with the different cloud services. These programs do their own logging, writing to the annex GAHP log. The annex GAHP log is configured as if it were a daemon, but with subsystem type `ANNEX_GAHP`; see *Daemon Logging Configuration File Entries* for details.

The annex audit log records two lines for each invocation of *condor_annex*: the command as issued and the results as returned. The location of the audit log is set by `ANNEX_AUDIT_LOG` , which is the `AUDIT`-level log for the `ANNEX` subsystem; see `<SUBSYS>_<LEVEL>_LOG` (in *Daemon Logging Configuration File Entries*) for details. Because annex creation commands typically make extensive use of values set in configuration, *condor_annex* will write the configuration it used for annex creation commands into the audit log if `ANNEX_DEBUG` includes `D_AUDIT:2`.

10.5.3 Expert Settings

ANNEX_DEFAULT_EC2_URL
The AWS EC2 endpoint that *condor_annex* should use. Defaults to 'https://ec2.us-east-1.amazonaws.com'.

ANNEX_DEFAULT_CWE_URL
The AWS CloudWatch Events endpoint that *condor_annex* should use. Defaults to 'https://events.us-east-1.amazonaws.com'.

ANNEX_DEFAULT_LAMBDA_URL
The AWS Lambda endpoint that *condor_annex* should use. Defaults to 'https://lambda.us-east-1.amazonaws.com'.

ANNEX_DEFAULT_S3_URL
The AWS S3 endpoint that *condor_annex* should use. Defaults to 'https://s3.amazonaws.com'.

ANNEX_DEFAULT_CF_URL
The AWS CloudFormation endpoint that *condor_annex* should use. Defaults to 'https://cloudformation.us-east-1.amazonaws.com'.

ANNEX_DEFAULT_ACCESS_KEY_FILE
The full path to the AWS access key file *condor_annex* should use. No default. If "FROM INSTANCE", *condor_annex* will assume it's running on an EC2 instance and try to use that instance's credentials.

ANNEX_DEFAULT_SECRET_KEY_FILE
The full path to the AWS secret key file *condor_annex* should use. No default. If "FROM INSTANCE", *condor_annex* will assume it's running on an EC2 instance and try to use that instance's credentials.

ANNEX_DEFAULT_S3_BUCKET
A private S3 bucket that the `ANNEX_DEFAULT_ACCESS_KEY_FILE` and `ANNEX_DEFAULT_SECRET_KEY_FILE` may write to. No default.

ANNEX_DEFAULT_ODI_SECURITY_GROUP_IDS
The default security group for on-demand annexes. Must permit inbound HTCondor (port 9618).

10.5.4 Developer Settings

ANNEX_DEFAULT_CONNECTIVITY_FUNCTION_ARN
The name (or ARN) of the Lambda function on AWS which *condor_annex* should use to check if the configured collector can be contacted from AWS.

ANNEX_DEFAULT_ODI_INSTANCE_PROFILE_ARN
The ARN of the instance profile *condor_annex* should use. No default.

ANNEX_DEFAULT_ODI_LEASE_FUNCTION_ARN
The Lambda function which implements the lease (duration) for on-demand instances. No default.

ANNEX_DEFAULT_SFR_LEASE_FUNCTION_ARN
The Lambda function which implements the lease (duration) for Spot instances. No default.

10.6 HTCondor in the Cloud

Although any HTCondor pool for which each node was running on a cloud resource could fairly be described as a "HTCondor in the Cloud", in this section we concern ourselves with creating such pools using *condor_annex*. The basic idea is start only a single instance manually – the "seed" node – which constitutes all of the HTCondor infrastructure required to run both *condor_annex* and jobs.

10.6.1 The HTCondor in the Cloud Seed

A seed node hosts the HTCondor pool infrastructure (the parts that aren't execute nodes). While HTCondor will try to reconnect to running jobs if the instance hosting the schedd shuts down, you would need to take additional precautions – making sure the seed node is automatically restarted, that it comes back quickly (faster than the job reconnect timeout), and that it comes back with the same IP address(es), among others – to minimize the amount of work-in-progress lost. We therefore recommend against using an interruptible instance for the seed node.

10.6.2 Security

Your cloud provider may allow you grant an instance privileges (e.g., the privilege of starting new instances). This can be more convenient (because you don't have to manually copy credentials into the instance), but may be risky if you allow others to log into the instance (possibly allowing them to take advantage of the instance's privileges). Conversely, copying credentials into the instance makes it easy to forget to remove them before creating an image of that instance (if you do).

10.6.3 Making a HTCondor in the Cloud

The general instructions are simple:

1. Start an instance from a seed image. Grant it privileges if you want. (See above).
2. If you did not grant the instance privileges, copy your credentials to the instance.
3. Run *condor_annex*.

AWS-Specific Instructions

The following instructions create a HTCondor-in-the-Cloud using the default seed image.

1. Go to the EC2 console.
2. Click the 'Launch Instance' button.
3. Click on 'Community AMIs'.
4. Search for `Condor-in-the-Cloud Seed`. (The AMI ID is `ami-00eeb25291cfad66f`.) Click the 'Select' button.
5. Choose an instance type. (Select `m5.large` if you have no preference.)
6. Click the 'Next: Configure Instance Details' button.
7. For 'IAM Role', select the role you created in *Using Instance Credentials*, or follow those instructions now.
8. Click '6. Configure Security Group'. This creates a firewall rule to allow you to log into your instance.
9. Click the 'Review and Launch' button.

10. Click the 'Launch' button.
11. Select an existing key pair if you have one; you will need the corresponding private key file to log in to your instance. If you don't have one, select 'Create a new key pair' and enter a name; 'HTCondor Annex' is fine. Click 'Download key pair'. Save the file some place you can access easily but others can't; you'll need it later.
12. Click through, then click the button labelled 'View Instances'.
13. The IPv4 address of your seed instance will be display. Use SSH to connect to that address as the 'ec2-user' with the key pair from two steps ago.

To grow your new HTCondor-in-the-Cloud from this seed, follow the instructions for using *condor_annex* for the first time, starting with *Configure condor_annex*. You can than proceed to *Start an Annex*.

10.6.4 Creating a Seed

A seed image is simply an image with:

- HTCondor installed
- HTCondor configured to:
  - be a central manager
  - be a submit node
  - allow *condor_annex* can add nodes
- a small script to set `TCP_FORWARDING_HOST` to the instance's public IP adress when the instance starts up.

More-detailed instructions for constructing a seed node on AWS are available. A RHEL 7.6 image built according to those instructions is available as public AMI `ami-00eeb25291cfad66f`.

10.7 Google Cloud Marketplace Entry

A solution for provisioning a pool using HTCondor 8.8 was made available on the Google Cloud Marketplace. It has been deprecated and will be removed at a future date.

10.8 Google Cloud HPC Toolkit

The Cloud HPC Toolkit is an Open Source solution for provisioning HPC and HTC solutions on Google Cloud Platform (GCP). Please consult the following resources for using the Toolkit to provision HTCondor on GCP:

- Cloud HPC Toolkit HTCondor Tutorial
- Cloud HPC Toolkit source code

CHAPTER

ELEVEN

GRID COMPUTING

11.1 Introduction

A goal of grid computing is to allow an authorized batch scheduler to send jobs to run on some remote pool, even when that remote pool is running a non-HTCondor system.

There are several mechanisms in HTCondor to do this.

Flocking allows HTCondor jobs submitted from one pool to execute on another, separate HTCondor pool. Flocking is enabled by configuration on both of the pools. An advantage to flocking is that jobs migrate from one pool to another based on the availability of machines to execute jobs. When the local HTCondor pool is not able to run the job (due to a lack of currently available machines), the job flocks to another pool. A second advantage to using flocking is that the submitting user does not need to be concerned with any aspects of the job. The user's submit description file (and the job's **universe**) are independent of the flocking mechanism. Flocking only works when the remote pool is also an HTCondor pool.

Glidein is the technique where *condor_startds* are submitted as jobs to some remote batch systems, and configured with report to, and expand the local HTCondor batch system. We call these jobs that run startds "pilot jobs", to distinguish them from the "payload jobs" which run the real user's domain work. HTCondor itself does not provide an implementation of glidein, there is a very complete implementation the HEP community has built, named GlideinWMS, and several HTCondor users have written their own glidein systems.

Other forms of grid computing are enabled by using the **grid universe** and further specified with the **grid_type**. For any HTCondor job, the job is submitted on a machine in the local HTCondor pool. The location where it is executed is identified as the remote machine or remote resource. These various grid computing mechanisms offered by HTCondor are distinguished by the software running on the remote resource. Often implementations of Glidein use grid universe to send the pilot jobs to a remote system.

When HTCondor is running on the remote resource, and the desired grid computing mechanism is to move the job from the local pool's job queue to the remote pool's job queue, it is called HTCondor-C. The job is submitted using the **grid universe**, and the **grid_type** is **condor**. HTCondor-C jobs have the advantage that once the job has moved to the remote pool's job queue, a network partition does not affect the execution of the job. A further advantage of HTCondor-C jobs is that the **universe** of the job at the remote resource is not restricted.

One disadvantage of grid universe is the destination must be declared in the submit file when condor_submit is run, locking the job to that remote site. The condor job router is a condor daemon which can periodically scan the scheduler's job queue, and change a vanilla universe job intended to run on the local cluster into a grid job, destined for a remote cluster. It can also be configured so that if this grid job is idle for too long, it can undo the transformation, so that the job isn't stuck forever in a remote queue.

Further specification of a **grid** universe job is done within the **grid_resource** command in a submit description file.

11.2 Connecting HTCondor Pools with Flocking

Flocking is HTCondor's way of allowing jobs that cannot immediately run within the pool of machines where the job was submitted to instead run on a different HTCondor pool. If a machine within HTCondor pool A can send jobs to be run on HTCondor pool B, then we say that jobs from machine A flock to pool B. Flocking can occur in a one way manner, such as jobs from machine A flocking to pool B, or it can be set up to flock in both directions. Configuration variables allow the *condor_schedd* daemon (which runs on each machine that may submit jobs) to implement flocking.

NOTE: Flocking to pools which use HTCondor's high availability mechanisms is not advised. See *High Availability of the Central Manager* for a discussion of the issues.

11.2.1 Flocking Configuration

The simplest flocking configuration sets a few configuration variables. If jobs from machine A are to flock to pool B, then in machine A's configuration, set the following configuration variables:

FLOCK_TO
: is a comma separated list of the central manager machines of the pools that jobs from machine A may flock to.

FLOCK_COLLECTOR_HOSTS
: is the list of *condor_collector* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as FLOCK_TO, and it would be defined with

```
FLOCK_COLLECTOR_HOSTS = $(FLOCK_TO)
```

FLOCK_NEGOTIATOR_HOSTS
: is the list of *condor_negotiator* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as FLOCK_TO, and it would be defined with

```
FLOCK_NEGOTIATOR_HOSTS = $(FLOCK_TO)
```

ALLOW_NEGOTIATOR_SCHEDD
: provides an access level and authorization list for the *condor_schedd* daemon to allow negotiation (for security reasons) with the machines within the pools that jobs from machine A may flock to. This configuration variable will not likely need to change from its default value as given in the sample configuration:

```
##  Now, with flocking we need to let the SCHEDD trust the other
##  negotiators we are flocking with as well.  You should normally
##  not have to change this either.
ALLOW_NEGOTIATOR_SCHEDD = $(CONDOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS), $(IP_ADDRESS)
```

: This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine. See the *Authorization* section for a discussion of security macros and their use.

The configuration macros that must be set in pool B are ones that authorize jobs from machine A to flock to pool B.

The configuration variables are more easily set by introducing a list of machines where the jobs may flock from. FLOCK_FROM is a comma separated list of machines, and it is used in the default configuration setting of the security macros that do authorization:

```
ALLOW_WRITE_COLLECTOR = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_WRITE_STARTD    = $(ALLOW_WRITE), $(FLOCK_FROM)
```

(continues on next page)

(continued from previous page)

```
ALLOW_READ_COLLECTOR  = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_READ_STARTD     = $(ALLOW_READ), $(FLOCK_FROM)
```

Wild cards may be used when setting the FLOCK_FROM configuration variable. For example, *.cs.wisc.edu specifies all hosts from the cs.wisc.edu domain.

Further, if using Kerberos or SSL authentication, then the setting becomes:

```
ALLOW_NEGOTIATOR = condor@$(UID_DOMAIN)/$(COLLECTOR_HOST)
```

To enable flocking in both directions, consider each direction separately, following the guidelines given.

11.2.2 Job Considerations

A particular job will only flock to another pool when it cannot currently run in the current pool.

The submission of jobs must consider the location of input, output and error files. The common case will be that machines within separate pools do not have a shared file system. Therefore, when submitting jobs, the user will need to enable file transfer mechanisms. These mechanisms are discussed in the *Submitting Jobs Without a Shared File System: HTCondor's File Transfer Mechanism* section.

11.3 The Grid Universe

11.3.1 HTCondor-C, The condor Grid Type

HTCondor-C allows jobs in one machine's job queue to be moved to another machine's job queue. These machines may be far removed from each other, providing powerful grid computation mechanisms, while requiring only HTCondor software and its configuration.

HTCondor-C is highly resistant to network disconnections and machine failures on both the submission and remote sides. An expected usage sets up Personal HTCondor on a laptop, submits some jobs that are sent to an HTCondor pool, waits until the jobs are staged on the pool, then turns off the laptop. When the laptop reconnects at a later time, any results can be pulled back.

HTCondor-C scales gracefully when compared with HTCondor's flocking mechanism. The machine upon which jobs are submitted maintains a single process and network connection to a remote machine, without regard to the number of jobs queued or running.

HTCondor-C Configuration

There are two aspects to configuration to enable the submission and execution of HTCondor-C jobs. These two aspects correspond to the endpoints of the communication: there is the machine from which jobs are submitted, and there is the remote machine upon which the jobs are placed in the queue (executed).

Configuration of a machine from which jobs are submitted requires a few extra configuration variables:

```
CONDOR_GAHP = $(SBIN)/condor_c-gahp
C_GAHP_LOG = /tmp/CGAHPLog.$(USERNAME)
C_GAHP_WORKER_THREAD_LOG = /tmp/CGAHPWorkerLog.$(USERNAME)
C_GAHP_WORKER_THREAD_LOCK = /tmp/CGAHPWorkerLock.$(USERNAME)
```

The acronym GAHP stands for Grid ASCII Helper Protocol. A GAHP server provides grid-related services for a variety of underlying middle-ware systems. The configuration variable `CONDOR_GAHP` gives a full path to the GAHP server utilized by HTCondor-C. The configuration variable `C_GAHP_LOG` defines the location of the log that the HTCondor GAHP server writes. The log for the HTCondor GAHP is written as the user on whose behalf it is running; thus the `C_GAHP_LOG` configuration variable must point to a location the end user can write to.

A submit machine must also have a *condor_collector* daemon to which the *condor_schedd* daemon can submit a query. The query is for the location (IP address and port) of the intended remote machine's *condor_schedd* daemon. This facilitates communication between the two machines. This *condor_collector* does not need to be the same collector that the local *condor_schedd* daemon reports to.

The machine upon which jobs are executed must also be configured correctly. This machine must be running a *condor_schedd* daemon. Unless specified explicitly in a submit file, `CONDOR_HOST` must point to a *condor_collector* daemon that it can write to, and the machine upon which jobs are submitted can read from. This facilitates communication between the two machines.

An important aspect of configuration is the security configuration relating to authentication. HTCondor-C on the remote machine relies on an authentication protocol to know the identity of the user under which to run a job. The following is a working example of the security configuration for authentication. This authentication method, CLAIMTOBE, trusts the identity claimed by a host or IP address.

```
SEC_DEFAULT_NEGOTIATION = OPTIONAL
SEC_DEFAULT_AUTHENTICATION_METHODS = CLAIMTOBE
```

Other working authentication methods are SSL, KERBEROS, and FS.

HTCondor-C Job Submission

Job submission of HTCondor-C jobs is the same as for any HTCondor job. The **universe** is **grid**. The submit command **grid_resource** specifies the remote *condor_schedd* daemon to which the job should be submitted, and its value consists of three fields. The first field is the grid type, which is **condor**. The second field is the name of the remote *condor_schedd* daemon. Its value is the same as the *condor_schedd* ClassAd attribute `Name` on the remote machine. The third field is the name of the remote pool's *condor_collector*.

The following represents a minimal submit description file for a job.

```
# minimal submit description file for an HTCondor-C job
universe = grid
```

(continues on next page)

(continued from previous page)

```
executable = myjob
output = myoutput
error = myerror
log = mylog

grid_resource = condor joe@remotemachine.example.com remotecentralmanager.example.com
+remote_jobuniverse = 5
+remote_requirements = True
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
queue
```

The remote machine needs to understand the attributes of the job. These are specified in the submit description file using the '+' syntax, followed by the string **remote_**. At a minimum, this will be the job's **universe** and the job's **requirements**. It is likely that other attributes specific to the job's **universe** (on the remote pool) will also be necessary. Note that attributes set with '+' are inserted directly into the job's ClassAd. Specify attributes as they must appear in the job's ClassAd, not the submit description file. For example, the **universe** is specified using an integer assigned for a job ClassAd `JobUniverse`. Similarly, place quotation marks around string expressions. As an example, a submit description file would ordinarily contain

```
when_to_transfer_output = ON_EXIT
```

This must appear in the HTCondor-C job submit description file as

```
+remote_WhenToTransferOutput = "ON_EXIT"
```

For convenience, the specific entries of **universe** and **remote_grid_resource** may be specified as **remote_** commands without the leading '+'. Instead of

```
+remote_universe = 5
```

the submit description file command may appear as

```
remote_universe = vanilla
```

Similarly, the command

```
+remote_gridresource = "condor schedd.example.com cm.example.com"
```

may be given as

```
remote_grid_resource = condor schedd.example.com cm.example.com
```

For the given example, the job is to be run as a **vanilla universe** job at the remote pool. The (remote pool's) *condor_schedd* daemon is likely to place its job queue data on a local disk and execute the job on another machine within the pool of machines. This implies that the file systems for the resulting submit machine (the machine specified by **remote_schedd**) and the execute machine (the machine that runs the job) will not be shared. Thus, the two inserted ClassAd attributes

```
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
```

are used to invoke HTCondor's file transfer mechanism.

For communication between *condor_schedd* daemons on the submit and remote machines, the location of the remote *condor_schedd* daemon is needed. This information resides in the *condor_collector* of the remote machine's pool. The third field of the **grid_resource** command in the submit description file says which *condor_collector* should be queried for the remote *condor_schedd* daemon's location. An example of this submit command is

```
grid_resource = condor schedd.example.com machine1.example.com
```

If the remote *condor_collector* is not listening on the standard port (9618), then the port it is listening on needs to be specified:

```
grid_resource = condor schedd.example.com machine1.example.com:12345
```

File transfer of a job's executable, `stdin`, `stdout`, and `stderr` are automatic. When other files need to be transferred using HTCondor's file transfer mechanism (see the *Submitting Jobs Without a Shared File System: HTCondor's File Transfer Mechanism* section), the mechanism is applied based on the resulting job universe on the remote machine.

HTCondor-C Jobs Between Differing Platforms

HTCondor-C jobs given to a remote machine running Windows must specify the Windows domain of the remote machine. This is accomplished by defining a ClassAd attribute for the job. Where the Windows domain is different at the submit machine from the remote machine, the submit description file defines the Windows domain of the remote machine with

```
+remote_NTDomain = "DomainAtRemoteMachine"
```

A Windows machine not part of a domain defines the Windows domain as the machine name.

11.3.2 The arc Grid Type

NorduGrid is a project to develop free grid middleware named the Advanced Resource Connector (ARC). See the NorduGrid web page (http://www.nordugrid.org) for more information about NorduGrid software.

NorduGrid ARC supports multiple job submission interfaces. The **arc** grid type uses their new REST interface.

HTCondor jobs may be submitted to ARC CE resources using the **grid** universe. The **grid_resource** command specifies the name of the ARC CE service as follows:

```
grid_resource = arc https://arc.example.com:443/arex/rest/1.0
```

Only the hostname portion of the URL is required. Appropriate defaults will be used for the other components.

ARC accepts X.509 credentials and SciTokens for authentication. You must specify one of these two credential types for your **arc** grid jobs. The submit description file command **x509userproxy** may be used to give the full path name of an X.509 proxy file. The submit description file command **scitokens_file** may be used to give the full path name of a SciTokens file. If both an X.509 proxy and a SciTokens file are provided, then only the SciTokens file is used for authentication. Whenever an X.509 proxy is provided, it is delegated to the ARC CE for use by the job.

ARC CE allows sites to define Runtime Environment (RTE) labels that alter the environment in which a job runs. Jobs can request one or move of these labels. For example, the `ENV/PROXY` label makes the user's X.509 proxy available to the job when it executes. Some of these labels have optional parameters for customization. The submit description file command **arc_rte** can be used to request one of more of these labels. It is a comma-delimited list. If a label supports optional parameters, they can be provided after the label separated by spaces. Here is an example showing use of two standard RTE labels, one with an optional parameter:

```
arc_rte = ENV/RTE,ENV/PROXY USE_DELEGATION_DB
```

ARC CE uses ADL (Activity Description Language) syntax to describe jobs. The specification of the language can be found here. HTCondor constructs an ADL description of the job based on attributes in the job ClassAd, but some ADL elements don't have an equivalent job ClassAd attribute. The submit description file command **arc_resources** can be used to specify these elements if they fall under the <Resources> element of the ADL. The value should be a chunk of XML text that could be inserted inside the <Resources> element. For example:

```
arc_resources = <NetworkInfo>gigabitethernet</NetworkInfo>
```

Similarly, submit description file command **arc_application** can be used to specify these elements if they fall under the <Application> element of the ADL.

11.3.3 The batch Grid Type (for SLURM, PBS, LSF, and SGE)

The **batch** grid type is used to submit to a local SLURM, PBS, LSF, or SGE system using the **grid** universe and the **grid_resource** command by placing a variant of the following into the submit description file.

```
grid_resource = batch slurm
```

The second argument on the right hand side will be one of `slurm`, `pbs`, `lsf`, or `sge`.

Submission to a batch system on a remote machine using SSH is also possible. This is described below.

The batch GAHP server is a piece of software called the blahp. The configuration parameters `BATCH_GAHP` and `BLAHPD_LOCATION` specify the locations of the main blahp binary and its dependent files, respectively. The blahp has its own configuration file, located at /etc/blah.config (`$(RELEASE_DIR)`/etc/blah.config for a tarball release).

The batch GAHP supports translating certain job ClassAd attributes into the corresponding batch system submission parameters. However, note that not all parameters are supported.

The following table summarizes how job ClassAd attributes will be translated into the corresponding Slurm job parameters.

Job ClassAd	Slurm
`RequestMemory`	`--mem`
`BatchRuntime`	`--time`
`BatchProject`	`--account`
`Queue`	`--partition`
`Queue`	`--clusters`
Unsupported	`--cpus-per-task`

Note that for Slurm, `Queue` is used for both `--partition` and `--clusters`. If you use the `partition@cluster` syntax, the partition will be set to whatever is before the `@`, and the cluster to whatever is after the `@`. If you only wish to set the cluster, leave out the partition (e.g. use `@cluster`).

You can specify batch system parameters that HTCondor doesn't have translations for using the **batch_extra_submit_args** command in the submit description file.

```
batch_extra_submit_args = --cpus-per-task=4 --qos=fast
```

The *condor_qsub* command line tool will take PBS/SGE style batch files or command line arguments and submit the job to HTCondor instead. See the *condor_qsub* manual page for details.

Remote batch Job Submission via SSH

HTCondor can submit jobs to a batch system on a remote machine via SSH. This requires an initial setup step that installs some binaries under your home directory on the remote machine and creates an SSH key that allows SSH authentication without the user typing a password. The setup command is *condor_remote_cluster*, which you should run at the command line.

```
condor_remote_cluster --add alice@login.example.edu slurm
```

Once this setup command finishes successfully, you can submit jobs for the remote batch system by including the username and hostname in the **grid_resource** command in your submit description file.

```
grid_resource = batch slurm alice@login.example.edu
```

Remote batch Job Submission via Reverse SSH

Submission to a batch system on a remote machine requires that HTCondor be able to establish an SSH connection using just an ssh key for authentication. If the remote machine doesn't allow ssh keys or requires Multi-Factor Authentication (MFA), then the SSH connection can be established in the reverse connection using the Reverse GAHP. This requires some extra setup and maintenance, and is not recommended if the normal SSH connection method can be made to work.

For the Reverse GAHP to work, your local machine must be reachable on the network from the remote machine on the SSH and HTCondor ports (22 and 9618, respectively). Also, your local machine must allow SSH logins using just an ssh key for authentication.

First, run the *condor_remote_cluster* as you would for a regular remote SSH setup.

```
condor_remote_cluster --add alice@login.example.edu slurm
```

Second, create an ssh key that's authorized to login to your account on your local machine and save the private key on the remote machine. The private key should not be protected with a passphrase. In the following examples, we'll assume the ssh private key is named `~/.ssh/id_rsa_rvgahp`.

Third, select a pathname on your local machine for a unix socket file that will be used by the Reverse GAHP components to communicate with each other. The Reverse GAHP programs will create the file as your user identity, so we suggest using a location under your home directory or /tmp. In the following examples, we'll use `/tmp/alice.rvgahp.socket`.

Fourth, on the remote machine, create a `~/bosco/glite/bin/rvgahp_ssh` shell script like this:

```
#!/bin/bash
exec ssh -o "ServerAliveInterval 60" -o "BatchMode yes" -i ~/.ssh/id_rsa_rvgahp␣
↪alice@submithost "/usr/sbin/rvgahp_proxy /tmp/alice.rvgahp.sock"
```

Run this script manually to ensure it works. It should print a couple messages from the *rvgahp_proxy* started on your local machine. You can kill the program once it's working correctly.

```
2022-03-23 13:06:08.304520 rvgahp_proxy[8169]: rvgahp_proxy starting...
2022-03-23 13:06:08.304766 rvgahp_proxy[8169]: UNIX socket: /tmp/alice.rvgahp.sock
```

Finally, run the *rvgahp_server* program on the remote machine. You must ensure it remains running during the entire time you are submitting and running jobs on the batch system.

```
~/bosco/glite/bin/rvgahp_server -b ~/bosco/glite
```

Now, you can submit jobs for the remote batch system. Adding the **–rvgahp-socket** option to your **grid_resource** submit command tells HTCondor to use the Reverse GAHP for the SSH connection.

```
grid_resource =␣
↪batch slurm alice@login.example.edu --rvgahp-socket /tmp/alice.rvgahp.sock
```

11.3.4 The EC2 Grid Type

HTCondor jobs may be submitted to clouds supporting Amazon's Elastic Compute Cloud (EC2) interface. The EC2 interface permits on-line commercial services that provide the rental of computers by the hour to run computational applications. They run virtual machine images that have been uploaded to Amazon's online storage service (S3 or EBS). More information about Amazon's EC2 service is available at http://aws.amazon.com/ec2.

The **ec2** grid type uses the EC2 Query API, also called the EC2 REST API.

EC2 Job Submission

HTCondor jobs are submitted to an EC2 service with the **grid** universe, setting the **grid_resource** command to **ec2**, followed by the service's URL. For example, partial contents of the submit description file may be

```
grid_resource = ec2 https://ec2.us-east-1.amazonaws.com/
```

(Replace 'us-east-1' with the AWS region you'd like to use.)

Since the job is a virtual machine image, most of the submit description file commands specifying input or output files are not applicable. The **executable** command is still required, but its value is ignored. It can be used to identify different jobs in the output of *condor_q*.

The VM image for the job must already reside in one of Amazon's storage service (S3 or EBS) and be registered with EC2. In the submit description file, provide the identifier for the image using **ec2_ami_id** .

This grid type requires access to user authentication information, in the form of path names to files containing the appropriate keys, with one exception, described below.

The **ec2** grid type has two different authentication methods. The first authentication method uses the EC2 API's built-in authentication. Specify the service with expected `http://` or `https://` URL, and set the EC2 access key and secret access key as follows:

```
ec2_access_key_id = /path/to/access.key
ec2_secret_access_key = /path/to/secret.key
```

The `euca3://` and `euca3s://` protocols must use this authentication method. These protocols exist to work correctly when the resources do not support the `InstanceInitiatedShutdownBehavior` parameter.

The second authentication method for the EC2 grid type is X.509. Specify the service with an `x509://` URL, even if the URL was given in another form. Use **ec2_access_key_id** to specify the path to the X.509 public key (certificate), which is not the same as the built-in authentication's access key. **ec2_secret_access_key** specifies the path to the X.509 private key, which is not the same as the built-in authentication's secret key. The following example illustrates the specification for X.509 authentication:

```
grid_resource = ec2 x509://service.example
ec2_access_key_id = /path/to/x.509/public.key
ec2_secret_access_key = /path/to/x.509/private.key
```

If using an X.509 proxy, specify the proxy in both places.

The exception to both of these cases applies when submitting EC2 jobs to an HTCondor running in an EC2 instance. If that instance has been configured with sufficient privileges, you may specify `FROM INSTANCE` for either **ec2_access_key_id** or **ec2_secret_access_key**, and HTCondor will use the instance's credentials. (AWS grants an EC2 instance access to temporary credentials, renewed over the instance's lifetime, based on the instance's assigned IAM (instance) profile and the corresponding IAM role. You may specify the this information when launching an instance or later, during its lifetime.)

HTCondor can use the EC2 API to create an SSH key pair that allows secure log in to the virtual machine once it is running. If the command **ec2_keypair_file** is set in the submit description file, HTCondor will write an SSH private key into the indicated file. The key can be used to log into the virtual machine. Note that modification will also be needed of the firewall rules for the job to incoming SSH connections.

An EC2 service uses a firewall to restrict network access to the virtual machine instances it runs. Typically, no incoming connections are allowed. One can define sets of firewall rules and give them names. The EC2 API calls these security groups. If utilized, tell HTCondor what set of security groups should be applied to each VM using the **ec2_security_groups** submit description file command. If not provided, HTCondor uses the security group **default**. This command specifies security group names; to specify IDs, use **ec2_security_ids** . This may be necessary when specifying a Virtual Private Cloud (VPC) instance.

To run an instance in a VPC, set **ec2_vpc_subnet** to the the desired VPC's specification string. The instance's IP address may also be specified by setting **ec2_vpc_id** .

The EC2 API allows the choice of different hardware configurations for instances to run on. Select which configuration to use for the **ec2** grid type with the **ec2_instance_type** submit description file command. HTCondor provides no default.

Certain instance types provide additional block devices whose names must be mapped to kernel device names in order to be used. The **ec2_block_device_mapping** submit description file command allows specification of these maps. A map is a device name followed by a colon, followed by kernel name; maps are separated by a commas, and/or spaces. For example, to specify that the first ephemeral device should be `/dev/sdb` and the second `/dev/sdc`:

```
ec2_block_device_mapping = ephemeral0:/dev/sdb, ephemeral1:/dev/sdc
```

Each virtual machine instance can be given up to 16 KiB of unique data, accessible by the instance by connecting to a well-known address. This makes it easy for many instances to share the same VM image, but perform different work. This data can be specified to HTCondor in one of two ways. First, the data can be provided directly in the submit description file using the **ec2_user_data** command. Second, the data can be stored in a file, and the file name is specified with the **ec2_user_data_file** submit description file command. This second option allows the use of binary data. If both options are used, the two blocks of data are concatenated, with the data from **ec2_user_data** occurring first. HTCondor performs the base64 encoding that EC2 expects on the data.

Amazon also offers an Identity and Access Management (IAM) service. To specify an IAM (instance) profile for an EC2 job, use submit commands **ec2_iam_profile_name** or **ec2_iam_profile_arn** .

Termination of EC2 Jobs

A protocol defines the shutdown procedure for jobs running as EC2 instances. The service is told to shut down the instance, and the service acknowledges. The service then advances the instance to a state in which the termination is imminent, but the job is given time to shut down gracefully.

Once this state is reached, some services other than Amazon cannot be relied upon to actually terminate the job. Thus, HTCondor must check that the instance has terminated before removing the job from the queue. This avoids the possibility of HTCondor losing track of a job while it is still accumulating charges on the service.

HTCondor checks after a fixed time interval that the job actually has terminated. If the job has not terminated after a total of four checks, the job is placed on hold.

Using Spot Instances

EC2 jobs may also be submitted to clouds that support spot instances. A spot instance differs from a conventional, or dedicated, instance in two primary ways. First, the instance price varies according to demand. Second, the cloud provider may terminate the instance prematurely. To start a spot instance, the submitter specifies a bid, which represents the most the submitter is willing to pay per hour to run the VM. Within HTCondor, the submit command **ec2_spot_price** specifies this floating point value. For example, to bid 1.1 cents per hour on Amazon:

```
ec2_spot_price = 0.011
```

Note that the EC2 API does not specify how the cloud provider should interpret the bid. Empirically, Amazon uses fractional US dollars.

Other submission details for a spot instance are identical to those for a dedicated instance.

A spot instance will not necessarily begin immediately. Instead, it will begin as soon as the price drops below the bid. Thus, spot instance jobs may remain in the idle state for much longer than dedicated instance jobs, as they wait for the price to drop. Furthermore, if the price rises above the bid, the cloud service will terminate the instance.

More information about Amazon's spot instances is available at http://aws.amazon.com/ec2/spot-instances/.

EC2 Advanced Usage

Additional control of EC2 instances is available in the form of permitting the direct specification of instance creation parameters. To set an instance creation parameter, first list its name in the submit command **ec2_parameter_names** , a space or comma separated list. The parameter may need to be properly capitalized. Also tell HTCondor the parameter's value, by specifying it as a submit command whose name begins with **ec2_parameter_**; dots within the parameter name must be written as underscores in the submit command name.

For example, the submit description file commands to set parameter `IamInstanceProfile.Name` to value `ExampleProfile` are

```
ec2_parameter_names = IamInstanceProfile.Name
ec2_parameter_IamInstanceProfile_Name = ExampleProfile
```

EC2 Configuration Variables

The configuration variables `EC2_GAHP` and `EC2_GAHP_LOG` must be set, and by default are equal to $(SBIN)/ec2_gahp and /tmp/EC2GahpLog.$(USERNAME), respectively.

The configuration variable `EC2_GAHP_DEBUG` is optional and defaults to D_PID; we recommend you keep D_PID if you change the default, to disambiguate between the logs of different resources specified by the same user.

Communicating with an EC2 Service

The **ec2** grid type does not presently permit the explicit use of an HTTP proxy.

By default, HTCondor assumes that EC2 services are reliably available. If an attempt to contact a service during the normal course of operation fails, HTCondor makes a special attempt to contact the service. If this attempt fails, the service is marked as down, and normal operation for that service is suspended until a subsequent special attempt succeeds. The jobs using that service do not go on hold. To place jobs on hold when their service becomes unavailable, set configuration variable `EC2_RESOURCE_TIMEOUT` to the number of seconds to delay before placing the job on hold. The default value of -1 for this variable implements an infinite delay, such that the job is never placed on hold. When setting this value, consider the value of configuration variable `GRIDMANAGER_RESOURCE_PROBE_INTERVAL` , which sets the number of seconds that HTCondor will wait after each special contact attempt before trying again.

By default, the EC2 GAHP enforces a 100 millisecond interval between requests to the same service. This helps ensure reliable service. You may configure this interval with the configuration variable EC2_GAHP_RATE_LIMIT, which must be an integer number of milliseconds. Adjusting the interval may result in higher or lower throughput, depending on the service. Too short of an interval may trigger rate-limiting by the service; while HTCondor will react appropriately (by retrying with an exponential back-off), it may be more efficient to configure a longer interval.

Secure Communication with an EC2 Service

The specification of a service with an `https://`, an `x509://`, or an `euca3s://` URL validates that service's certificate, checking that a trusted certificate authority (CA) signed it. Commercial EC2 service providers generally use certificates signed by widely-recognized CAs. These CAs will usually work without any additional configuration. For other providers, a specification of trusted CAs may be needed. Without, errors such as the following will be in the EC2 GAHP log:

```
06/13/13 15:16:16 curl_easy_perform() failed (60):
'Peer certificate cannot be authenticated with given CA certificates'.
```

Specify trusted CAs by including their certificates in a group of trusted CAs either in an on disk directory or in a single file. Either of these alternatives may contain multiple certificates. Which is used will vary from system to system, depending on the system's SSL implementation. HTCondor uses *libcurl*; information about the *libcurl* specification of trusted CAs is available at

http://curl.haxx.se/libcurl/c/curl_easy_setopt.html

The behavior when specifying both a directory and a file is undefined, although the EC2 GAHP allows it.

The EC2 GAHP will set the CA file to whichever variable it finds first, checking these in the following order:

1. The environment variable X509_CERT_FILE, set when the *condor_master* starts up.
2. The HTCondor configuration variable GAHP_SSL_CAFILE .

The EC2 GAHP supplies no default value, if it does not find a CA file.

The EC2 GAHP will set the CA directory given whichever of these variables it finds first, checking in the following order:

1. The environment variable X509_CERT_DIR, set when the *condor_master* starts up.
2. The HTCondor configuration variable GAHP_SSL_CADIR .

The EC2 GAHP supplies no default value, if it does not find a CA directory.

EC2 GAHP Statistics

The EC2 GAHP tracks, and reports in the corresponding grid resource ad, statistics related to resource's rate limit.

NumRequests:
The total number of requests made by HTCondor to this resource.

NumDistinctRequests:
The number of distinct requests made by HTCondor to this resource. The difference between this and NumRequests is the total number of retries. Retries are not unusual.

NumRequestsExceedingLimit:
The number of requests which exceeded the service's rate limit. Each such request will cause a retry, unless the maximum number of retries is exceeded, or if the retries have already taken so long that the signature on the original request has expired.

NumExpiredSignatures:
The number of requests which the EC2 GAHP did not even attempt to send to the service because signature expired. Signatures should not, generally, expire; a request's retries will usually - eventually - succeed.

11.3.5 The GCE Grid Type

HTCondor jobs may be submitted to the Google Compute Engine (GCE) cloud service. GCE is an on-line commercial service that provides the rental of computers by the hour to run computational applications. Its runs virtual machine images that have been uploaded to Google's servers. More information about Google Compute Engine is available at http://cloud.google.com/Compute.

GCE Job Submission

HTCondor jobs are submitted to the GCE service with the **grid** universe, setting the **grid_resource** command to **gce**, followed by the service's URL, your GCE project, and the desired GCE zone to be used. The submit description file command will be similar to:

```
grid_resource = gce https://www.googleapis.com/compute/v1 my_proj us-central1-a
```

Since the HTCondor job is a virtual machine image, most of the submit description file commands specifying input or output files are not applicable. The **executable** command is still required, but its value is ignored. It identifies different jobs in the output of *condor_q*.

The VM image for the job must already reside in Google's Cloud Storage service and be registered with GCE. In the submit description file, provide the identifier for the image using the **gce_image** command.

This grid type requires granting HTCondor permission to use your Google account. The easiest way to do this is to use the *gcloud* command-line tool distributed by Google. Find *gcloud* and documentation for it at https://cloud.google.com/compute/docs/gcloud-compute/. After installation of *gcloud*, run *gcloud auth login* and follow its directions. Once done with that step, the tool will write authorization credentials to the file `.config/gcloud/credentials` under your HOME directory.

Given an authorization file, specify its location in the submit description file using the **gce_auth_file** command, as in the example:

```
gce_auth_file = /path/to/auth-file
```

GCE allows the choice of different hardware configurations for instances to run on. Select which configuration to use for the **gce** grid type with the **gce_machine_type** submit description file command. HTCondor provides no default.

Each virtual machine instance can be given a unique set of metadata, which consists of name/value pairs, similar to the environment variables of regular jobs. The instance can query its metadata via a well-known address. This makes it easy for many instances to share the same VM image, but perform different work. This data can be specified to HTCondor in one of two ways. First, the data can be provided directly in the submit description file using the **gce_metadata** command. The value should be a comma-separated list of name=value settings, as the example:

```
gce_metadata = setting1=foo,setting2=bar
```

Second, the data can be stored in a file, and the file name is specified with the **gce_metadata_file** submit description file command. This second option allows a wider range of characters to be used in the metadata values. Each name=value pair should be on its own line. No white space is removed from the lines, except for the newline that separates entries.

Both options can be used at the same time, but do not use the same metadata name in both places.

HTCondor sets the following elements when describing the instance to the GCE server: **machineType**, **name**, **scheduling**, **disks**, **metadata**, and **networkInterfaces**. You can provide additional elements to be included in the instance description as a block of JSON. Write the additional elements to a file, and specify the filename in your submit file with the **gce_json_file** command. The contents of the file are inserted into HTCondor's JSON description of the instance, between a comma and the closing brace.

Here's a sample JSON file that sets two additional elements:

```
"canIpForward": True,
"description": "My first instance"
```

GCE Configuration Variables

The following configuration parameters are specific to the **gce** grid type. The values listed here are the defaults. Different values may be specified in the HTCondor configuration files. To work around an issue where long-running *gce_gahp* processes have trouble authenticating, the *gce_gahp* self-restarts periodically, with the default value of 24 hours. You can set the number of seconds between restarts using *GCE_GAHP_LIFETIME*, where zero means to never restart. Restarting the *gce_gahp* does not affect the jobs themselves.

```
GCE_GAHP      = $(SBIN)/gce_gahp
GCE_GAHP_LOG = /tmp/GceGahpLog.$(USERNAME)
GCE_GAHP_LIFETIME = 86400
```

11.3.6 The Azure Grid Type

HTCondor jobs may be submitted to the Microsoft Azure cloud service. Azure is an on-line commercial service that provides the rental of computers by the hour to run computational applications. It runs virtual machine images that have been uploaded to Azure's servers. More information about Azure is available at https://azure.microsoft.com.

Azure Job Submission

HTCondor jobs are submitted to the Azure service with the **grid** universe, setting the **grid_resource** command to **azure**, followed by your Azure subscription id. The submit description file command will be similar to:

```
grid_resource = azure 4843bfe3-1ebe-423e-a6ea-c777e57700a9
```

Since the HTCondor job is a virtual machine image, most of the submit description file commands specifying input or output files are not applicable. The **executable** command is still required, but its value is ignored. It identifies different jobs in the output of *condor_q*.

The VM image for the job must already be registered a virtual machine image in Azure. In the submit description file, provide the identifier for the image using the **azure_image** command.

This grid type requires granting HTCondor permission to use your Azure account. The easiest way to do this is to use the *az* command-line tool distributed by Microsoft. Find *az* and documentation for it at https://docs.microsoft.com/en-us/cli/azure/?view=azure-cli-latest. After installation of *az*, run *az login* and follow its directions. Once done with that step, the tool will write authorization credentials in a file under your HOME directory. HTCondor will use these credentials to communicate with Azure.

You can also set up a service account in Azure for HTCondor to use. This lets you limit the level of access HTCondor has to your Azure account. Instructions for creating a service account can be found here: https://htcondor.org/gahp/AzureGAHPSetup.docx.

Once you have created a file containing the service account credentials, you can specify its location in the submit description file using the **azure_auth_file** command, as in the example:

```
azure_auth_file = /path/to/auth-file
```

Azure allows the choice of different hardware configurations for instances to run on. Select which configuration to use for the **azure** grid type with the **azure_size** submit description file command. HTCondor provides no default.

Azure has many locations where instances can be run (i.e. multiple data centers distributed throughout the world). You can select which location to use with the **azure_location** submit description file command.

Azure creates an administrator account within each instance, which you can log into remote via SSH. You can select the name of the account with the **azure_admin_username** command. You can supply the name of a file containing an SSH public key that will allow access to the administrator account with the **azure_admin_key** command.

11.4 The HTCondor Job Router

The HTCondor Job Router is an add-on to the *condor_schedd* that transforms jobs from one type into another according to a configurable policy. This process of transforming the jobs is called job routing.

One example of how the Job Router can be used is for the task of sending excess jobs to one or more remote grid sites. The Job Router can transform the jobs such as vanilla universe jobs into grid universe jobs that use any of the grid types supported by HTCondor. The rate at which jobs are routed can be matched roughly to the rate at which the site is able to start running them. This makes it possible to balance a large work flow across multiple grid sites, a local HTCondor pool, and any flocked HTCondor pools, without having to guess in advance how quickly jobs will run and complete in each of the different sites.

Job Routing is most appropriate for high throughput work flows, where there are many more jobs than computers, and the goal is to keep as many of the computers busy as possible. Job Routing is less suitable when there are a small number of jobs, and the scheduler needs to choose the best place for each job, in order to finish them as quickly as possible. The Job Router does not know which site will run the jobs faster, but it can decide whether to send more jobs to a site, based on whether jobs already submitted to that site are sitting idle or not, as well as whether the site has experienced recent job failures.

11.4.1 Routing Mechanism

The *condor_job_router* daemon and configuration determine a policy for which jobs may be transformed and sent to grid sites. By default, a job is transformed into a grid universe job by making a copy of the original job ClassAd, and modifying some attributes in this copy of the job. The copy is called the routed copy, and it shows up in the job queue under a new job id.

Until the routed copy finishes or is removed, the original copy of the job passively mirrors the state of the routed job. During this time, the original job is not available for matchmaking, because it is tied to the routed copy. The original job also does not evaluate periodic expressions, such as `PeriodicHold`. Periodic expressions are evaluated for the routed copy. When the routed copy completes, the original job ClassAd is updated such that it reflects the final status of the job. If the routed copy is removed, the original job returns to the normal idle state, and is available for matchmaking or rerouting. If, instead, the original job is removed or goes on hold, the routed copy is removed.

Although the default mode routes vanilla universe jobs to grid universe jobs, the routing rules may be configured to do some other transformation of the job. It is also possible to edit the job in place rather than creating a new transformed version of the job.

The *condor_job_router* daemon utilizes a routing table, in which a ClassAd transform describes each site to where jobs may be sent.

There is also a list of pre-route and post-route transforms that are applied whenever a job is routed.

The routing table is given as a set of configuration macros. Each configuration macro is given in the job transform language. This is the same transform language used by the *condor_schedd* for job transforms. This language is similar to the *condor_submit* language, but has commands to describe the transform steps and optional macro values such as `MaxJobs` that can control the way the route is used.

When a route matches a job, and the *condor_job_router* is about to apply the routing transform, it will first apply all of the pre-route transforms that match that job, then it will apply the routing transform, then it will apply all of the post-route transforms that match the job.

In older versions the routing table was given as a list of ClassAds, and for backwards compatibility this form of configuration is still supported - It will be converted automatically into a set of job transforms.

11.4.2 Job Submission with Job Routing Capability

If Job Routing is set up, then the following items ought to be considered for jobs to have the necessary prerequisites to be considered for routing.

- Jobs appropriate for routing to the grid must not rely on access to a shared file system, or other services that are only available on the local pool. The job will use HTCondor's file transfer mechanism, rather than relying on a shared file system to access input files and write output files. In the submit description file, to enable file transfer, there will be a set of commands similar to

  ```
  should_transfer_files = YES
  when_to_transfer_output = ON_EXIT
  transfer_input_files = input1, input2
  transfer_output_files = output1, output2
  ```

  Vanilla universe jobs and most types of grid universe jobs differ in the set of files transferred back when the job completes. Vanilla universe jobs transfer back all files created or modified, while all grid universe jobs, except for HTCondor-C, only transfer back the **output** file, as well as those explicitly listed with **transfer_output_files** . Therefore, when routing jobs to grid universes other than HTCondor-C, it is important to explicitly specify all output files that must be transferred upon job completion.

- One configuration for routed jobs requires the jobs to identify themselves as candidates for Job Routing. This may be accomplished by inventing a ClassAd attribute that the configuration utilizes in setting the policy for job identification, and the job defines this attribute to identify itself. If the invented attribute is called `WantJobRouter`, then the job identifies itself as a job that may be routed by placing in the submit description file:

  ```
  +WantJobRouter = True
  ```

  This implementation can be taken further, allowing the job to first be rejected within the local pool, before being a candidate for Job Routing:

  ```
  +WantJobRouter = LastRejMatchTime =!= UNDEFINED
  ```

- As appropriate to the potential grid site, create a grid proxy, and specify it in the submit description file:

  ```
  x509userproxy = /tmp/x509up_u275
  ```

  This is not necessary if the *condor_job_router* daemon is configured to add a grid proxy on behalf of jobs.

Job submission does not change for jobs that may be routed.

```
$ condor_submit job1.sub
```

where `job1.sub` might contain:

```
universe = vanilla
executable = my_executable
output = job1.stdout
error = job1.stderr
log = job1.ulog
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
+WantJobRouter = LastRejMatchTime =!= UNDEFINED
x509userproxy = /tmp/x509up_u275
queue
```

The status of the job may be observed as with any other HTCondor job, for example by looking in the job's log file. Before the job completes, *condor_q* shows the job's status. Should the job become routed, a second job will enter the job queue. This is the routed copy of the original job. The command *condor_router_q* shows a more specialized view of routed jobs, as this example shows:

```
$ condor_router_q -S
   JOBS ST Route      GridResource
     40  I Site1      site1.edu/jobmanager-condor
     10  I Site2      site2.edu/jobmanager-pbs
      2  R Site3      condor submit.site3.edu condor.site3.edu
```

condor_router_history summarizes the history of routed jobs, as this example shows:

```
$ condor_router_history
Routed job history from 2007-06-27 23:38 to 2007-06-28 23:38

Site            Hours    Jobs    Runs
                      Completed Aborted
-------------------------------------------------------
Site1              10       2     0
Site2               8       2     1
Site3              40       6     0
-------------------------------------------------------
TOTAL              58      10     1
```

11.4.3 An Example Configuration

The following sample configuration sets up potential job routing to three routes (grid sites). Definitions of the configuration variables specific to the Job Router are in the *condor_job_router Configuration File Entries* section. One route a local SLURM cluster. A second route is cluster accessed via ARC CE. The third site is an HTCondor site accessed by HTCondor-C. The *condor_job_router* daemon does not know which site will be best for a given job. The policy implemented in this sample configuration stops sending more jobs to a site, if ten jobs that have already been sent to that site are idle.

These configuration settings belong in the local configuration file of the machine where jobs are submitted. Check that the machine can successfully submit grid jobs before setting up and using the Job Router. Typically, the single required element that needs to be added for SSL authentication is an X.509 trusted certification authority directory, in a place recognized by HTCondor (for example, `/etc/grid-security/certificates`).

Note that, as of version 8.5.6, the configuration language supports multi-line values, as shown in the example below (see the *Multi-Line Values* section for more details).

The list of enabled routes is specified by JOB_ROUTER_ROUTE_NAMES, routes will be considered in the order given by this configuration variable.

```
# define a global constraint, only jobs that match this will be considered for routing
JOB_ROUTER_SOURCE_JOB_CONSTRAINT = WantJobRouter

# define a default maximum number of jobs that will be matched to each route
# and a limit on the number of idle jobs a route may have before we stop using it.
JOB_ROUTER_DEFAULT_MAX_JOBS_PER_ROUTE = 200
JOB_ROUTER_DEFAULT_MAX_IDLE_JOBS_PER_ROUTE = 10

# This could be made an attribute of the job, rather than being hard-coded
ROUTED_JOB_MAX_TIME = 1440

# Now we define each of the routes to send jobs to
JOB_ROUTER_ROUTE_NAMES = Site1 Site2 CondorSite

JOB_ROUTER_ROUTE_Site1 @=rt
  GridResource = "batch slurm"
@rt

JOB_ROUTER_ROUTE_Site2 @=rt
  GridResource = "arc site2.edu"
  SET ArcRte = "ENV/PROXY"
@rt

JOB_ROUTER_ROUTE_CondorSite @=rt
  MaxIdleJobs = 20
  GridResource = "condor submit.site3.edu cm.site3.edu"
  SET remote_jobuniverse = 5
@rt

# define a pre-route transform that does the transforms all routes should do
JOB_ROUTER_PRE_ROUTE_TRANSFORM_NAMES = Defaults

JOB_ROUTER_TRANSFORM_Defaults @=jrd
   # remove routed job if it goes on hold or stays idle for over 6 hours
   SET PeriodicRemove = JobStatus == 5 || \
                        (JobStatus == 1 && (time() - QDate) > 3600*6))
   # delete the global SOURCE_JOB_CONSTRAINT attribute so that routed jobs will not be␣
↪routed again
   DELETE WantJobRouter
   SET Requirements = true
@jrd


# Reminder: you must restart HTCondor for changes to DAEMON_LIST to take effect.
DAEMON_LIST = $(DAEMON_LIST) JOB_ROUTER

# For testing, set this to a small value to speed things up.
# Once you are running at large scale, set it to a higher value
# to prevent the JobRouter from using too much cpu.
JOB_ROUTER_POLLING_PERIOD = 10
```

(continues on next page)

(continued from previous page)

```
#It is good to save lots of schedd queue history
#for use with the router_history command.
MAX_HISTORY_ROTATIONS = 20
```

11.4.4 Routing Table Entry Commands and Macro values

A route consists of a sequence of Macro values and commands which are applied in order to produce the routed job ClassAd. Certain macro names have special meaning when used in a router transform. These special macro names are listed below along a brief listing of the the transform commands. For a more detailed description of the transform commands refer to the *Transform Commands* section.

The conversion of a job to a routed copy will usually require the job ClassAd to be modified. The Routing Table specifies attributes of the different possible routes and it may specify specific modifications that should be made to the job when it is sent along a specific route. In addition to this mechanism for transforming the job, external programs may be invoked to transform the job. For more information, see the *Hooks for the Job Router* section.

The following attributes and instructions for modifying job attributes may appear in a Routing Table entry.

GridResource = <string>
: Specifies the value for the `GridResource` attribute that will be inserted into the routed copy of the job's ClassAd.

Requirements = <expr>
: A `Requirements` expression that identifies jobs that may be matched to the route. If there is a `JOB_ROUTER_SOURCE_JOB_CONSTRAINT` then only jobs that match that constraint *and* this `Requirements` expression can match this route.

MaxJobs = <integer>
: An integer maximum number of jobs permitted on the route at one time. The default is 100.

MaxIdleJobs = <integer>
: An integer maximum number of routed jobs in the idle state. At or above this value, no more jobs will be sent to this site. This is intended to prevent too many jobs from being sent to sites which are too busy to run them. If the value set for this attribute is too small, the rate of job submission to the site will slow, because the *condor_job_router* daemon will submit jobs up to this limit, wait to see some of the jobs enter the running state, and then submit more. The disadvantage of setting this attribute's value too high is that a lot of jobs may be sent to a site, only to site idle for hours or days. The default value is 50.

FailureRateThreshold = <float>
: A maximum tolerated rate of job failures. Failure is determined by the expression sets for the attribute `JobFailureTest` expression. The default threshold is 0.03 jobs/second. If the threshold is exceeded, submission of new jobs is throttled until jobs begin succeeding, such that the failure rate is less than the threshold. This attribute implements black hole throttling, such that a site at which jobs are sent only to fail (a black hole) receives fewer jobs.

JobFailureTest = <boolean expr>
: An expression evaluated for each job that finishes, to determine whether it was a failure. The default value if no expression is defined assumes all jobs are successful. Routed jobs that are removed are considered to be failures. An example expression to treat all jobs running for less than 30 minutes as failures is `target.RemoteWallClockTime < 1800`. A more flexible expression might reference a property or expression of the job that specifies a failure condition specific to the type of job.

SendIDTokens = <string expr>
: A string expression that lists the names of the IDTOKENS to add to the input file transfer list of the routed job. The string should list one or more of the IDTOKEN names specified by the `JOB_ROUTER_CREATE_IDTOKEN_NAMES` configuration variable. if `SendIDTokens` is not specified, then the value of the JobRouter configuration variable `JOB_ROUTER_SEND_ROUTE_IDTOKENS` will be used.

UseSharedX509UserProxy = <boolean epr>
: A boolean expression that when `True` causes the value of `SharedX509UserProxy` to be the X.509 user proxy for the routed job. Note that if the *condor_job_router* daemon is running as root, the copy of this file that is given to the job will have its ownership set to that of the user running the job. This requires the trust of the user. It is therefore recommended to avoid this mechanism when possible. Instead, require users to submit jobs with `X509UserProxy` set in the submit description file. If this feature is needed, use the boolean expression to only allow specific values of `target.Owner` to use this shared proxy file. The shared proxy file should be owned by the condor user. Currently, to use a shared proxy, the job must also turn on sandboxing by having the attribute `JobShouldBeSandboxed`.

SharedX509UserProxy = <string>
: A string representing file containing the X.509 user proxy for the routed job.

JobShouldBeSandboxed = <boolean expr>
: A boolean expression that when `True` causes the created copy of the job to be sandboxed. A copy of the input files will be placed in the *condor_schedd* daemon's spool area for the target job, and when the job runs, the output will be staged back into the spool area. Once all of the output has been successfully staged back, it will be copied again, this time from the spool area of the sandboxed job back to the original job's output locations. By default, sandboxing is turned off. Only to turn it on if using a shared X.509 user proxy or if direct staging of remote output files back to the final output locations is not desired.

EditJobInPlace = <boolean expr>
: A boolean expression that, when `True`, causes the original job to be transformed in place rather than creating a new transformed version (a routed copy) of the job. In this mode, the Job Router Hook `<Keyword>_HOOK_TRANSLATE_JOB` and transformation rules in the routing table are applied during the job transformation. The routing table attribute `GridResource` is ignored, and there is no default transformation of the job from a vanilla job to a grid universe job as there is otherwise. Once transformed, the job is still a candidate for matching routing rules, so it is up to the routing logic to control whether the job may be transformed multiple times or not. For example, to transform the job only once, an attribute could be set in the job ClassAd to prevent it from matching the same routing rule in the future. To transform the job multiple times with limited frequency, a timestamp could be inserted into the job ClassAd marking the time of the last transformation, and the routing entry could require that this timestamp either be undefined or older than some limit.

UNIVERSE <value>
: A universe name or integer value specifying the desired universe for the routed copy of the job. The default value

is 9, which is the **grid** universe.

SET <attr> <expr>
Sets the value of <attr> in the routed copy's job ClassAd to the specified value. An example of an attribute that might be set is `PeriodicRemove`. For example, if the routed job goes on hold or stays idle for too long, remove it and return the original copy of the job to a normal state.

DEFAULT <attr> <expr>
Sets the value of <attr> if the value is currently missing or undefined. This is equivalent to

```
if ! defined MY.<Attr>
  SET <Attr> <value>
endif
```

EVALSET <attr> <expr>
Defines an expression. The expression is evaluated, and the resulting value sets the value of the routed copy's job ClassAd attribute <attr>. Use this when the attribute must not be an expression or when information available only to the *condor_job_router* is needed to determine the value.

EVALMACRO <var> <expr>
Defines an expression. The expression is evaluated, and the resulting value is store in the temporary variable <var>. `$(var)` can the be used in later statements in this route or in a later transform that is part of this route. This is often use to evaluate complex expressions that can later be used in `if` statements in the route.

COPY <attr> <newattr>
Copies the value of <attr> from the original attribute name to a new attribute name in the routed copy. Useful to save the value of an expression that you intend to change as part of the route so that the value prior to routing is still visible in the job ClassAd.

COPY /<regex>/ <attrpat>
Copies all attributes that match the regular expression <regex> to new attribute names.

RENAME <attr> <newattr>
Renames the attribute <attr> to a new attribute name. This is the equivalent of a COPY statement followed by a DELETE statement.

RENAME /<regex>/ <attrpat>
Renames all attributes that match the regular expression <regex> to new attribute names.

DELETE <attr>
Deletes <attr> from the routed copy of the job ClassAd.

DELETE /<regex>/
Deletes all attributes that match the regular expression <regex> from the routed copy of the job.

11.4.5 Deprecated router configuration

Warning: The deprecated job router configuration macro JOB_ROUTER_DEFAULTS will be removed during the lifetime of the HTCondor V23 feature series in preparation of HTCondor V24.

Prior to version 8.9.7 the *condor_job_router* used a list of ClassAds to configure the routes. This form of configuration is still supported. It will be converted at load time to the new syntax.

A good place to learn about the syntax of ClassAds is the Informal Language Description in the C++ ClassAds tutorial: http://htcondor.org/classad/c++tut.html. Two essential differences distinguish the ClassAd syntax used by the *condor_job_router* from the syntax used in most other areas of HTCondor. In the router configuration, each ClassAd is surrounded by square brackets. And each assignment statement ends with a semicolon. Newlines are ignored by the parser. Thus When the ClassAd is embedded in an HTCondor configuration file, it may appear all on a single line, but the readability is often improved by inserting line continuation characters after each assignment statement. This is done in the examples. Unfortunately, this makes the insertion of comments into the configuration file awkward, because of the interaction between comments and line continuation characters in configuration files. An alternative is to use C-style comments (`/* ...*/`). Another alternative is to read in the routing table entries from a separate file, rather than embedding them in the HTCondor configuration file.

Note that, as of version 8.5.6, the configuration language supports multi-line values, as shown in the example below (see the *Multi-Line Values* section for more details).

As of version 8.8.7, the order in which routes are considered can be configured by specifying *JOB_ROUTER_ROUTE_NAMES*. Prior to that version the order in which routes were considered could not be specified and so routes were normally given mutually exclusive requirements.

```
# These settings become the default settings for all routes
# because they are merged with each route before the route is applied
JOB_ROUTER_DEFAULTS @=jrd
  [
    requirements=target.WantJobRouter is True;
    MaxIdleJobs = 10;
    MaxJobs = 200;

    /* now modify routed job attributes */
    /* remove routed job if it goes on hold or stays idle for over 6 hours */
    set_PeriodicRemove = JobStatus == 5 ||
                         (JobStatus == 1 && (time() - QDate) > 3600*6);
    delete_WantJobRouter = true;
    set_requirements = true;
  ]
  @jrd

# This could be made an attribute of the job, rather than being hard-coded
ROUTED_JOB_MAX_TIME = 1440

# Now we define each of the routes to send jobs on
JOB_ROUTER_ENTRIES @=jre
  [ GridResource = "batch slurm";
    name = "Site_1";
  ]
  [ GridResource = "arc site2.edu";
    name = "Site_2";
```

(continues on next page)

(continued from previous page)

```
    set_ArcRte = "ENV/PROXY";
  ]
  [ GridResource = "condor submit.site3.edu cm.site3.edu";
    name = "Site_3";
    set_remote_jobuniverse = 5;
  ]
  @jre

# Optionally define the order that routes should be considered
# uncomment this line to declare the order
#JOB_ROUTER_ROUTE_NAMES = Site_1 Site_2 Site_3
```

11.4.6 Deprecated Routing Table Entry ClassAd Attributes

Warning: The deprecated job router configuration macros JOB_ROUTER_ENTRIES, JOB_ROUTER_ENTRIES_FILE, and JOB_ROUTER_ENTRIES_CMD will be removed during the lifetime of the HTCondor V23 feature series in preparation of HTCondor V24.

In the deprecated *condor_job_router* configuration, each route is the result of merging the *JOB_ROUTER_DEFAULTS* ClassAd with one of the *JOB_ROUTER_ENTRIES* ClassAds, with attributes specified in *JOB_ROUTER_ENTRIES* overriding those specified in *JOB_ROUTER_DEFAULTS*.

Name
: An optional identifier that will be used in log messages concerning this route. If no name is specified, the default used will be the value of `GridResource`. The *condor_job_router* distinguishes routes and advertises statistics based on this attribute's value.

TargetUniverse
: An integer value specifying the desired universe for the routed copy of the job. The default value is 9, which is the **grid** universe.

OverrideRoutingEntry
: A boolean value that when `True`, indicates that this entry in the routing table replaces any previous entry in the table with the same name. When `False`, it indicates that if there is a previous entry by the same name, the previous entry should be retained and this entry should be ignored. The default value is `True`.

Set_<ATTR>
: Sets the value of <ATTR> in the routed copy's job ClassAd to the specified value. An example of an attribute that might be set is `PeriodicRemove`. For example, if the routed job goes on hold or stays idle for too long, remove it and return the original copy of the job to a normal state.

Eval_Set_<ATTR>
: Defines an expression. The expression is evaluated, and the resulting value sets the value of the routed copy's job

ClassAd attribute <ATTR>. Use this attribute to set a custom or local value, especially for modifying an attribute which may have been already specified in a default routing table.

Copy_<ATTR>
Defined with the name of a routed copy ClassAd attribute. Copies the value of <ATTR> from the original job ClassAd into the specified attribute named of the routed copy. Useful to save the value of an expression, before replacing it with something else that references the original expression.

Delete_<ATTR>
Deletes <ATTR> from the routed copy ClassAd. A value assigned to this attribute in the routing table entry is ignored.

CHAPTER

TWELVE

PLATFORM-SPECIFIC INFORMATION

The HTCondor Team strives to make HTCondor work the same way across all supported platforms. However, because HTCondor is a very low-level system which interacts closely with the internals of the operating systems on which it runs, this goal is not always possible to achieve. The following sections provide detailed information about using HTCondor on different computing platforms and operating systems.

12.1 Linux

This section provides information specific to the Linux port of HTCondor.

HTCondor is sensitive to changes in the following elements of the system:

- The kernel version
- The version of the GNU C library (glibc)

The HTCondor Team provides support for the distributions of Linux which are most popular amoung our users. Red Hat, Debian and their derivatives are currenty the most popular Linux distributions in our space, and we provide native packages of HTCondor for these flavors.

12.2 Microsoft Windows

Windows is a strategic platform for HTCondor, and therefore we have been working toward a complete port to Windows. Our goal is to make HTCondor every bit as capable on Windows as it is on Unix – or even more capable.

Porting HTCondor from Unix to Windows is a formidable task, because many components of HTCondor must interact closely with the underlying operating system.

This section contains additional information specific to running HTCondor on Windows. In order to effectively use HTCondor, first read the *Overview* chapter and the *Users' Manual*. If administrating or customizing the policy and set up of HTCondor, also read the *Administrators' Manual* chapter. After reading these chapters, review the information in this chapter for important information and differences when using and administrating HTCondor on Windows. For information on installing HTCondor for Windows, see *Windows (as Administrator)*.

12.2.1 Limitations under Windows

In general, this release for Windows works the same as the release of HTCondor for Unix. However, the following items are not supported in this version:

- **grid** universe jobs may not be submitted from a Windows platform, unless the grid type is **condor**.
- Accessing files via a network share that requires a Kerberos ticket (such as AFS) is not yet supported.

12.2.2 Supported Features under Windows

Except for those items listed above, most everything works the same way in HTCondor as it does in the Unix release. This release is based on the HTCondor Version 23.0.8 source tree, and thus the feature set is the same as HTCondor Version 23.0.8 for Unix. For instance, all of the following work in HTCondor:

- The ability to submit, run, and manage queues of jobs running on a cluster of Windows machines.
- All tools such as *condor_q*, *condor_status*, *condor_userprio*, are included.
- The ability to customize job policy using ClassAds. The machine ClassAds contain all the information included in the Unix version, including current load average, RAM and virtual memory sizes, integer and floating-point performance, keyboard/mouse idle time, etc. Likewise, job ClassAds contain a full complement of information, including system dependent entries such as dynamic updates of the job's image size and CPU usage.
- Everything necessary to run an HTCondor central manager on Windows.
- Security mechanisms.
- HTCondor for Windows can run jobs at a lower operating system priority level. Jobs can be suspended, soft-killed by using a WM_CLOSE message, or hard-killed automatically based upon policy expressions. For example, HTCondor can automatically suspend a job whenever keyboard/mouse or non-HTCondor created CPU activity is detected, and continue the job after the machine has been idle for a specified amount of time.
- HTCondor correctly manages jobs which create multiple processes. For instance, if an HTCondor job spawns multiple processes and HTCondor needs to kill the job, all processes created by the job will be terminated.
- In addition to interactive tools, users and administrators can receive information from HTCondor by e-mail (standard SMTP) and/or by log files.
- HTCondor includes a friendly GUI installation and set up program, which can perform a full install or deinstall of HTCondor. Information specified by the user in the set up program is stored in the system registry. The set up program can update a current installation with a new release using a minimal amount of effort.
- HTCondor can give a job access to the running user's Registry hive.

12.2.3 Secure Password Storage

In order for HTCondor to operate properly, it must at times be able to act on behalf of users who submit jobs. This is required on submit machines, so that HTCondor can access a job's input files, create and access the job's output files, and write to the job's log file from within the appropriate security context. On Unix systems, arbitrarily changing what user HTCondor performs its actions as is easily done when HTCondor is started with root privileges. On Windows, however, performing an action as a particular user or on behalf of a particular user requires knowledge of that user's password, even when running at the maximum privilege level. HTCondor provides secure password storage through the use of the *condor_store_cred* tool. Passwords managed by HTCondor are encrypted and stored in a secure location within the Windows registry. When HTCondor needs to perform an action as or on behalf of a particular user, it uses the securely stored password to do so. This implies that a password is stored for every user that will submit jobs from the Windows submit machine.

A further feature permits HTCondor to execute the job itself under the security context of its submitting user, specifying the **run_as_owner** command in the job's submit description file. With this feature, it is necessary to configure and run a centralized *condor_credd* daemon to manage the secure password storage. This makes each user's password available, via an encrypted connection to the *condor_credd*, to any execute machine that may need it.

By default, the secure password store for a submit machine when no *condor_credd* is running is managed by the *condor_schedd*. This approach works in environments where the user's password is only needed on the submit machine.

12.2.4 Executing Jobs as the Submitting User

By default, HTCondor executes jobs on Windows using dedicated run accounts that have minimal access rights and privileges, and which are recreated for each new job. As an alternative, HTCondor can be configured to allow users to run jobs using their Windows login accounts. This may be useful if jobs need access to files on a network share, or to other resources that are not available to the low-privilege run account.

This feature requires use of a *condor_credd* daemon for secure password storage and retrieval. With the *condor_credd* daemon running, the user's password must be stored, using the *condor_store_cred* tool. Then, a user that wants a job to run using their own account places into the job's submit description file

```
run_as_owner = True
```

12.2.5 The condor_credd Daemon

The *condor_credd* daemon manages secure password storage. A single running instance of the *condor_credd* within an HTCondor pool is necessary in order to provide the feature described in *Executing Jobs as the Submitting User*, where a job runs as the submitting user, instead of as a temporary user that has strictly limited access capabilities.

It is first necessary to select the single machine on which to run the *condor_credd*. Often, the machine acting as the pool's central manager is a good choice. An important restriction, however, is that the *condor_credd* host must be a machine running Windows.

All configuration settings necessary to enable the *condor_credd* are contained in the example file etc\condor_config.local.credd from the HTCondor distribution. Copy these settings into a local configuration file for the machine that will run the *condor_credd*. Run `condor_restart` for these new settings to take effect, then verify (via Task Manager) that a *condor_credd* process is running.

A second set of configuration variables specify security for the communication among HTCondor daemons. These variables must be set for all machines in the pool. The following example settings are in the comments contained in the etc\condor_config.local.credd example file. These sample settings rely on the `PASSWORD` method for authentication among daemons, including communication with the *condor_credd* daemon. The `LOCAL_CREDD` variable must be customized to point to the machine hosting the *condor_credd* and the `ALLOW_CONFIG` variable will be customized, if needed, to refer to an administrative account that exists on all HTCondor nodes.

```
CREDD_HOST = credd.cs.wisc.edu
CREDD_CACHE_LOCALLY = True

STARTER_ALLOW_RUNAS_OWNER = True

ALLOW_CONFIG = Administrator@*
SEC_CLIENT_AUTHENTICATION_METHODS = NTSSPI, PASSWORD
SEC_CONFIG_NEGOTIATION = REQUIRED
```

(continues on next page)

(continued from previous page)

```
SEC_CONFIG_AUTHENTICATION = REQUIRED
SEC_CONFIG_ENCRYPTION = REQUIRED
SEC_CONFIG_INTEGRITY = REQUIRED
```

The example above can be modified to meet the needs of your pool, providing the following conditions are met:

1. The requesting client must use an authenticated connection
2. The requesting client must have an encrypted connection
3. The requesting client must be authorized for DAEMON level access.

Using a pool password on Windows

In order for PASSWORD authenticated communication to work, a pool password must be chosen and distributed. The chosen pool password must be stored identically for each machine. The pool password first should be stored on the *condor_credd* host, then on the other machines in the pool.

To store the pool password on a Windows machine, run

```
$ condor_store_cred add -c
```

when logged in with the administrative account on that machine, and enter the password when prompted. If the administrative account is shared across all machines, that is if it is a domain account or has the same password on all machines, logging in separately to each machine in the pool can be avoided. Instead, the pool password can be securely pushed out for each Windows machine using a command of the form

```
$ condor_store_cred add -c -n exec01.cs.wisc.edu
```

Once the pool password is distributed, but before submitting jobs, all machines must reevaluate their configuration, so execute

```
$ condor_reconfig -all
```

from the central manager. This will cause each execute machine to test its ability to authenticate with the *condor_credd*. To see whether this test worked for each machine in the pool, run the command

```
$ condor_status -f "%s\t" Name -f "%s\n" ifThenElse(isUndefined(LocalCredd),\"UNDEF\",
↪LocalCredd)
```

Any rows in the output with the UNDEF string indicate machines where secure communication is not working properly. Verify that the pool password is stored correctly on these machines.

Regardless of how Condor's authentication is configured, the pool password can always be set locally by running the

```
$ condor_store_cred add -c
```

command as the local SYSTEM account. Third party tools such as PsExec can be used to accomplish this. When condor_store_cred is run as the local SYSTEM account, it bypasses the network authentication and writes the pool password to the registry itself. This allows the other condor daemons (also running under the SYSTEM account) to access the pool password when authenticating against the pool's collector. In case the pool is remote and no initial communication can be established due to strong security, the pool password may have to be set using the above method and following command:

```
$ condor_store_cred -u condor_pool@poolhost add
```

12.2.6 Executing Jobs with the User's Profile Loaded

HTCondor can be configured when using dedicated run accounts, to load the account's profile. A user's profile includes a set of personal directories and a registry hive loaded under HKEY_CURRENT_USER.

This may be useful if the job requires direct access to the user's registry entries. It also may be useful when the job requires an application, and the application requires registry access. This feature is always enabled on the *condor_startd*, but it is limited to the dedicated run account. For security reasons, the profile is cleaned before a subsequent job which uses the dedicated run account begins. This ensures that malicious jobs cannot discover what any previous job has done, nor sabotage the registry for future jobs. It also ensures the next job has a fresh registry hive.

A job that is to run with a profile uses the **load_profile** command in the job's submit description file:

```
load_profile = True
```

This feature is currently not compatible with **run_as_owner** , and will be ignored if both are specified.

12.2.7 Using Windows Scripts as Job Executables

HTCondor has added support for scripting jobs on Windows. Previously, HTCondor jobs on Windows were limited to executables or batch files. With this new support, HTCondor determines how to interpret the script using the file name's extension. Without a file name extension, the file will be treated as it has been in the past: as a Windows executable.

This feature may not require any modifications to HTCondor's configuration. An example that does not require administrative intervention are Perl scripts using *ActivePerl*.

Windows Scripting Host scripts do require configuration to work correctly. The configuration variables set values to be used in registry look up, which results in a command that invokes the correct interpreter, with the correct command line arguments for the specific scripting language. In Microsoft nomenclature, verbs are actions that can be taken upon a given a file. The familiar examples of **Open**, **Print**, and **Edit**, can be found on the context menu when a user right clicks on a file. The command lines to be used for each of these verbs are stored in the registry under the HKEY_CLASSES_ROOT hive. In general, a registry look up uses the form:

```
HKEY_CLASSES_ROOT\<FileType>\Shell\<OpenVerb>\Command
```

Within this specification, <FileType> is the name of a file type (and therefore a scripting language), and is obtained from the file name extension. <OpenVerb> identifies the verb, and is obtained from the HTCondor configuration.

The HTCondor configuration sets the selection of a verb, to aid in the registry look up. The file name extension sets the name of the HTCondor configuration variable. This variable name is of the form:

```
OPEN_VERB_FOR_<EXT>_FILES
```

<EXT> represents the file name extension. The following configuration example uses the Open2 verb for a *Windows Scripting Host* registry look up for several scripting languages:

```
OPEN_VERB_FOR_JS_FILES  = Open2
OPEN_VERB_FOR_VBS_FILES = Open2
OPEN_VERB_FOR_VBE_FILES = Open2
OPEN_VERB_FOR_JSE_FILES = Open2
OPEN_VERB_FOR_WSF_FILES = Open2
OPEN_VERB_FOR_WSH_FILES = Open2
```

In this example, HTCondor specifies the Open2 verb, instead of the default Open verb, for a script with the file name extension of wsh. The *Windows Scripting Host* 's Open2 verb allows standard input, standard output, and standard error to be redirected as needed for HTCondor jobs.

A common difficulty is encountered when a script interpreter requires access to the user's registry. Note that the user's registry is different than the root registry. If not given access to the user's registry, some scripts, such as *Windows Scripting Host* scripts, will fail. The failure error message appears as:

```
CScript Error: Loading your settings failed. (Access is denied.)
```

The fix for this error is to give explicit access to the submitting user's registry hive. This can be accomplished with the addition of the **load_profile** command in the job's submit description file:

```
load_profile = True
```

With this command, there should be no registry access errors. This command should also work for other interpreters. Note that not all interpreters will require access. For example, *ActivePerl* does not by default require access to the user's registry hive.

12.2.8 How HTCondor for Windows Starts and Stops a Job

This section provides some details on how HTCondor starts and stops jobs. This discussion is geared for the HTCondor administrator or advanced user who is already familiar with the material in the Administrator's Manual and wishes to know detailed information on what HTCondor does when starting and stopping jobs.

When HTCondor is about to start a job, the *condor_startd* on the execute machine spawns a *condor_starter* process. The *condor_starter* then creates:

1. a run account on the machine with a login name of condor-slot<X>, where <X> is the slot number of the *condor_starter*. This account is added to group `Users` by default. The default group may be changed by setting configuration variable `DYNAMIC_RUN_ACCOUNT_LOCAL_GROUP` . This step is skipped if the job is to be run using the submitting user's account, as specified in *Executing Jobs as the Submitting User*.
2. a new temporary working directory for the job on the execute machine. This directory is named `dir_XXX`, where `XXX` is the process ID of the *condor_starter*. The directory is created in the `$(EXECUTE)` directory, as specified in HTCondor's configuration file. HTCondor then grants write permission to this directory for the user account newly created for the job.
3. a new, non-visible Window Station and Desktop for the job. Permissions are set so that only the account that will run the job has access rights to this Desktop. Any windows created by this job are not seen by anyone; the job is run in the background. Setting `USE_VISIBLE_DESKTOP` to `True` will allow the job to access the default desktop instead of a newly created one.

Next, the *condor_starter* daemon contacts the *condor_shadow* daemon, which is running on the submitting machine, and the *condor_starter* pulls over the job's executable and input files. These files are placed into the temporary working directory for the job. After all files have been received, the *condor_starter* spawns the user's executable. Its current working directory set to the temporary working directory.

While the job is running, the *condor_starter* closely monitors the CPU usage and image size of all processes started by the job. Every 20 minutes the *condor_starter* sends this information, along with the total size of all files contained in the job's temporary working directory, to the *condor_shadow*. The *condor_shadow* then inserts this information into the job's ClassAd so that policy and scheduling expressions can make use of this dynamic information.

If the job exits of its own accord (that is, the job completes), the *condor_starter* first terminates any processes started by the job which could still be around if the job did not clean up after itself. The *condor_starter* examines the job's

temporary working directory for any files which have been created or modified and sends these files back to the *condor_shadow* running on the submit machine. The *condor_shadow* places these files into the **initialdir** specified in the submit description file; if no **initialdir** was specified, the files go into the directory where the user invoked *condor_submit*. Once all the output files are safely transferred back, the job is removed from the queue. If, however, the *condor_startd* forcibly kills the job before all output files could be transferred, the job is not removed from the queue but instead switches back to the Idle state.

If the *condor_startd* decides to vacate a job prematurely, the *condor_starter* sends a WM_CLOSE message to the job. If the job spawned multiple child processes, the WM_CLOSE message is only sent to the parent process. This is the one started by the *condor_starter*. The WM_CLOSE message is the preferred way to terminate a process on Windows, since this method allows the job to clean up and free any resources it may have allocated. When the job exits, the *condor_starter* cleans up any processes left behind. At this point, if **when_to_transfer_output** is set to `ON_EXIT` (the default) in the job's submit description file, the job switches states, from Running to Idle, and no files are transferred back. If **when_to_transfer_output** is set to `ON_EXIT_OR_EVICT`, then files in the job's temporary working directory which were changed or modified are first sent back to the submitting machine. If exactly which files to transfer is specified with **transfer_output_files** , then this modifies the files transferred and can affect the state of the job if the specified files do not exist. On an eviction, the *condor_shadow* places these intermediate files into a subdirectory created in the `$(SPOOL)` directory on the submitting machine. The job is then switched back to the Idle state until HTCondor finds a different machine on which to run. When the job is started again, HTCondor places into the job's temporary working directory the executable and input files as before, plus any files stored in the submit machine's `$(SPOOL)` directory for that job.

Note: A Windows console process can intercept a WM_CLOSE message via the Win32 SetConsoleCtrlHandler() function, if it needs to do special cleanup work at vacate time; a WM_CLOSE message generates a CTRL_CLOSE_EVENT. See SetConsoleCtrlHandler() in the Win32 documentation for more info.

Note: The default handler in Windows for a WM_CLOSE message is for the process to exit. Of course, the job could be coded to ignore it and not exit, but eventually the *condor_startd* will become impatient and hard-kill the job, if that is the policy desired by the administrator.

Finally, after the job has left and any files transferred back, the *condor_starter* deletes the temporary working directory, the temporary account if one was created, the Window Station and the Desktop before exiting. If the *condor_starter* should terminate abnormally, the *condor_startd* attempts the clean up. If for some reason the *condor_startd* should disappear as well (that is, if the entire machine was power-cycled hard), the *condor_startd* will clean up when HTCondor is restarted.

12.2.9 Security Considerations in HTCondor for Windows

On the execute machine (by default), the user job is run using the access token of an account dynamically created by HTCondor which has bare-bones access rights and privileges. For instance, if your machines are configured so that only Administrators have write access to C:\WINNT, then certainly no HTCondor job run on that machine would be able to write anything there. The only files the job should be able to access on the execute machine are files accessible by the Users and Everyone groups, and files in the job's temporary working directory. Of course, if the job is configured to run using the account of the submitting user (as described in *Executing Jobs as the Submitting User*), it will be able to do anything that the user is able to do on the execute machine it runs on.

On the submit machine, HTCondor impersonates the submitting user, therefore the File Transfer mechanism has the same access rights as the submitting user. For example, say only Administrators can write to C:\WINNT on the submit machine, and a user gives the following to *condor_submit* :

```
executable = mytrojan.exe
initialdir = c:\winnt
output = explorer.exe
queue
```

Unless that user is in group Administrators, HTCondor will not permit `explorer.exe` to be overwritten.

If for some reason the submitting user's account disappears between the time *condor_submit* was run and when the job runs, HTCondor is not able to check and see if the now-defunct submitting user has read/write access to a given file. In this case, HTCondor will ensure that group "Everyone" has read or write access to any file the job subsequently tries to read or write. This is in consideration for some network setups, where the user account only exists for as long as the user is logged in.

HTCondor also provides protection to the job queue. It would be bad if the integrity of the job queue is compromised, because a malicious user could remove other user's jobs or even change what executable a user's job will run. To guard against this, in HTCondor's default configuration all connections to the *condor_schedd* (the process which manages the job queue on a given machine) are authenticated using Windows' eSSPI security layer. The user is then authenticated using the same challenge-response protocol that Windows uses to authenticate users to Windows file servers. Once authenticated, the only users allowed to edit job entry in the queue are:

1. the user who originally submitted that job (i.e. HTCondor allows users to remove or edit their own jobs)
2. users listed in the `condor_config` file parameter `QUEUE_SUPER_USERS`. In the default configuration, only the "SYSTEM" (LocalSystem) account is listed here.

WARNING: Do not remove "SYSTEM" from `QUEUE_SUPER_USERS`, or HTCondor itself will not be able to access the job queue when needed. If the LocalSystem account on your machine is compromised, you have all sorts of problems!

To protect the actual job queue files themselves, the HTCondor installation program will automatically set permissions on the entire HTCondor release directory so that only Administrators have write access.

Finally, HTCondor has all the security mechanisms present in the full-blown version of HTCondor. See the *Authorization* section for complete information on how to allow/deny access to HTCondor.

12.2.10 Network files and HTCondor

HTCondor can work well with a network file server. The recommended approach to having jobs access files on network shares is to configure jobs to run using the security context of the submitting user (see *Executing Jobs as the Submitting User*). If this is done, the job will be able to access resources on the network in the same way as the user can when logged in interactively.

In some environments, running jobs as their submitting users is not a feasible option. This section outlines some possible alternatives. The heart of the difficulty in this case is that on the execute machine, HTCondor creates a temporary user that will run the job. The file server has never heard of this user before.

Choose one of these methods to make it work:

- METHOD A: access the file server as a different user via a net use command with a login and password
- METHOD B: access the file server as guest
- METHOD C: access the file server with a "NULL" descriptor
- METHOD D: create and have HTCondor use a special account

All of these methods have advantages and disadvantages.

Here are the methods in more detail:

METHOD A - access the file server as a different user via a net use command with a login and password

Example: you want to copy a file off of a server before running it. . . .

```
@echo off
net use \\myserver\someshare MYPASSWORD /USER:MYLOGIN
copy \\myserver\someshare\my-program.exe
my-program.exe
```

The idea here is to simply authenticate to the file server with a different login than the temporary HTCondor login. This is easy with the "net use" command as shown above. Of course, the obvious disadvantage is this user's password is stored and transferred as clear text.

METHOD B - access the file server as guest

Example: you want to copy a file off of a server before running it as GUEST

```
@echo off
net use \\myserver\someshare
copy \\myserver\someshare\my-program.exe
my-program.exe
```

In this example, you'd contact the server MYSERVER as the HTCondor temporary user. However, if you have the GUEST account enabled on MYSERVER, you will be authenticated to the server as user "GUEST". If your file permissions (ACLs) are setup so that either user GUEST (or group EVERYONE) has access the share "someshare" and the directories/files that live there, you can use this method. The downside of this method is you need to enable the GUEST account on your file server. WARNING: This should be done *with extreme caution* and only if your file server is well protected behind a firewall that blocks SMB traffic.

METHOD C - access the file server with a "NULL" descriptor

One more option is to use NULL Security Descriptors. In this way, you can specify which shares are accessible by NULL Descriptor by adding them to your registry. You can then use the batch file wrapper like:

```
net use z: \\myserver\someshare /USER:""
z:\my-program.exe
```

so long as 'someshare' is in the list of allowed NULL session shares. To edit this list, run regedit.exe and navigate to the key:

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
     CurrentControlSet\
       Services\
         LanmanServer\
           Parameters\
             NullSessionShares
```

and edit it. Unfortunately it is a binary value, so you'll then need to type in the hex ASCII codes to spell out your share. Each share is separated by a null (0x00) and the last in the list is terminated with two nulls.

Although a little more difficult to set up, this method of sharing is a relatively safe way to have one quasi-public share without opening the whole guest account. You can control specifically which shares can be accessed or not via the registry value mentioned above.

METHOD D - create and have HTCondor use a special account

Create a permanent account (called condor-guest in this description) under which HTCondor will run jobs. On all Windows machines, and on the file server, create the condor-guest account.

On the network file server, give the condor-guest user permissions to access files needed to run HTCondor jobs.

Securely store the password of the condor-guest user in the Windows registry using *condor_store_cred* on all Windows machines.

Tell HTCondor to use the condor-guest user as the owner of jobs, when required. Details for this are in the *Security* section.

12.2.11 Interoperability between HTCondor for Unix and HTCondor for Windows

Unix machines and Windows machines running HTCondor can happily co-exist in the same HTCondor pool without any problems. Jobs submitted on Windows can run on Windows or Unix, and jobs submitted on Unix can run on Unix or Windows. Without any specification using the **Requirements** command in the submit description file, the default behavior will be to require the execute machine to be of the same architecture and operating system as the submit machine.

There is absolutely no need to run more than one HTCondor central manager, even if there are both Unix and Windows machines in the pool. The HTCondor central manager itself can run on either Unix or Windows; there is no advantage to choosing one over the other.

12.2.12 Some differences between HTCondor for Unix -vs- HTCondor for Windows

- On Unix, we recommend the creation of a condor account when installing HTCondor. On Windows, this is not necessary, as HTCondor is designed to run as a system service as user LocalSystem.
- On Unix, HTCondor finds the `condor_config` main configuration file by looking in ~condor, in `/etc`, or via an environment variable. On Windows, the location of `condor_config` file is determined via the registry key `HKEY_LOCAL_MACHINE/Software/Condor`. Override this value by setting an environment variable named `CONDOR_CONFIG`.
- On Unix, in the vanilla universe at job vacate time, HTCondor sends the job a softkill signal defined in the submit description file, which defaults to SIGTERM. On Windows, HTCondor sends a WM_CLOSE message to the job at vacate time.
- On Unix, if one of the HTCondor daemons has a fault, a core file will be created in the `$(Log)` directory. On Windows, a core file will also be created, but instead of a memory dump of the process, it will be a very short ASCII text file which describes what fault occurred and where it happened. This information can be used by the HTCondor developers to fix the problem.

12.3 Macintosh OS X

This section provides information specific to the Macintosh OS X port of HTCondor. The Macintosh port of HTCondor is more accurately a port of HTCondor to Darwin, the BSD layer of OS X. It is not well-integrated into the Macintosh environment beyond that.

HTCondor on the Macintosh has a few shortcomings:

- Users connected to the Macintosh via *ssh* are not noticed for console activity.
- The memory size of threaded programs is reported incorrectly.
- No Macintosh-based installer is provided.
- The example start up scripts do not follow Macintosh conventions.

12.4 Windows Installer

This section includes detailed information about the options offered by the Windows Installer, including how to run it unattended for automated installations. If you're not an experienced user, you may wish to follow the quick start guide's *instructions* instead.

12.4.1 Detailed Installation Instructions Using the MSI Program

This section describes the different HTCondor Installer options in greater detail.

STEP 1: License Agreement.
The first step in installing HTCondor is a welcome screen and license agreement. You are reminded that it is best to run the installation when no other Windows programs are running. If you need to close other Windows programs, it is safe to cancel the installation and close them. You are asked to agree to the license. Answer yes or no. If you should disagree with the License, the installation will not continue.

Also fill in name and company information, or use the defaults as given.

STEP 2: HTCondor Pool Configuration.
The HTCondor configuration needs to be set based upon if this is a new pool or to join an existing one. Choose the appropriate radio button.

For a new pool, enter a chosen name for the pool. To join an existing pool, enter the host name of the central manager of the pool.

STEP 3: This Machine's Roles.
Each machine within an HTCondor pool can either submit jobs or execute submitted jobs, or both submit and execute jobs. A check box determines if this machine will be a submit point for the pool.

A set of radio buttons determines the ability and configuration of the ability to execute jobs. There are four choices:

- Do not run jobs on this machine. This machine will not execute HTCondor jobs.
- Always run jobs and never suspend them.
- Run jobs when the keyboard has been idle for 15 minutes.
- Run jobs when the keyboard has been idle for 15 minutes, and the CPU is idle.

If you are setting up HTCondor as a single installation for testing, make sure you check the box to make the machine a submit point, and also choose the second option from the list above.

For a machine that is to execute jobs and the choice is one of the last two in the list, HTCondor needs to further know what to do with the currently running jobs. There are two choices:

- Keep the job in memory and continue when the machine meets the condition chosen for when to run jobs.
- Restart the job on a different machine.

This choice involves a trade off. Restarting the job on a different machine is less intrusive on the workstation owner than leaving the job in memory for a later time. A suspended job left in memory will require swap space, which could be a scarce resource. Leaving a job in memory, however, has the benefit that accumulated run time is not lost for a partially completed job.

STEP 4: The Account Domain.
Enter the machine's accounting (or UID) domain. On this version of HTCondor for Windows, this setting is only used for user priorities (see the *User Priorities and Negotiation* section) and to form a default e-mail address for the user.

STEP 5: E-mail Settings.
Various parts of HTCondor will send e-mail to an HTCondor administrator if something goes wrong and requires human attention. Specify the e-mail address and the SMTP relay host of this administrator. Please pay close attention to this e-mail, since it will indicate problems in the HTCondor pool.

STEP 6: Java Settings.
In order to run jobs in the **java** universe, HTCondor must have the path to the jvm executable on the machine. The installer will search for and list the jvm path, if it finds one. If not, enter the path. To disable use of the **java** universe, leave the field blank.

STEP 7: Access Permission Settings.
Machines within the HTCondor pool will need various types of access permission. The three categories of permission that can be set here are read, write, and administrator. The values can be usernames, hostnames or IP address ranges, Wild cards and macros are permitted. It is recommended that you accept the defaults here and change the values later as needed by modifying the HTCondor configuration files.

Read
Read access allows a machine to obtain information about HTCondor such as the status of machines in the pool and the job queues. If all of your HTCondor machines and users are in a single DNS domain or IP Address range, setting this to *.domain an IP address range with wildcards is a good choice. See `ALLOW_READ`

Write
Write access is for submitting jobs to the Schedd. Setting this to * will allow any user that can login to the machine submit jobs. See `ALLOW_WRITE`

Administrator
Administrator access is for starting and stopping the daemons and sending administrative commands such as reconfig and drain. By default the installer will give this permission to the Windows user that runs the installer and to the Windows `Adminstrator` account. See `ALLOW_ADMINISTRATOR`

For more details on these access permissions, and others that can be manually changed in your configuration file, please see the section titled Setting Up Security in HTCondor in the *Authorization* section.

STEP 8: VM Universe Setting.
A radio button determines whether this machine will be configured to run **vm** universe jobs utilizing VMware. In addition to having the VMware Server installed, HTCondor also needs *Perl* installed. The resources available for **vm** universe jobs can be tuned with these settings, or the defaults listed can be used.

Version
Use the default value, as only one version is currently supported.

Maximum Memory
The maximum memory that each virtual machine is permitted to use on the target machine.

Maximum Number of VMs
The number of virtual machines that can be run in parallel on the target machine.

Networking Support
The VMware instances can be configured to use network support. There are four options in the pull-down menu.

- None: No networking support.
- NAT: Network address translation.
- Bridged: Bridged mode.
- NAT and Bridged: Allow both methods.

Path to Perl Executable
The path to the *Perl* executable.

STEP 9: Choose Setup Type

The next step is where the destination of the HTCondor files will be decided. We recommend that HTCondor be installed in the location shown as the default in the install choice: C:\Condor. This is due to several hard coded paths in scripts and configuration files. Clicking on the Custom choice permits changing the installation directory.

Installation on the local disk is chosen for several reasons. The HTCondor services run as local system, and within Microsoft Windows, local system has no network privileges. Therefore, for HTCondor to operate, HTCondor should be installed on a local hard drive, as opposed to a network drive (file server).

The second reason for installation on the local disk is that the Windows usage of drive letters has implications for where HTCondor is placed. The drive letter used must be not change, even when different users are logged in. Local drive letters do not change under normal operation of Windows.

While it is strongly discouraged, it may be possible to place HTCondor on a hard drive that is not local, if a dependency is added to the service control manager such that HTCondor starts after the required file services are available.

12.4.2 Unattended Installation Procedure Using the MSI Installer

This section details how to run the HTCondor for Windows installer in an unattended batch mode. This mode is one that occurs completely from the command prompt, without the GUI interface.

The HTCondor for Windows installer uses the Microsoft Installer (MSI) technology, and it can be configured for unattended installs analogous to any other ordinary MSI installer.

The following is a sample batch file that is used to set all the properties necessary for an unattended install.

```
@echo on
set ARGS=
set ARGS=NEWPOOL="N"
set ARGS=%ARGS% POOLNAME=""
set ARGS=%ARGS% RUNJOBS="C"
set ARGS=%ARGS% VACATEJOBS="Y"
set ARGS=%ARGS% SUBMITJOBS="Y"
set ARGS=%ARGS% CONDOREMAIL="you@yours.com"
set ARGS=%ARGS% SMTPSERVER="smtp.localhost"
set ARGS=%ARGS% ALLOWREAD="*"
set ARGS=%ARGS% ALLOWWRITE="*"
set ARGS=%ARGS% ALLOWADMINISTRATOR="$(IP_ADDRESS)"
set ARGS=%ARGS% INSTALLDIR="C:\Condor"
set ARGS=%ARGS% POOLHOSTNAME="$(IP_ADDRESS)"
set ARGS=%ARGS% ACCOUNTINGDOMAIN="none"
set ARGS=%ARGS% JVMLOCATION="C:\Windows\system32\java.exe"
set ARGS=%ARGS% USEVMUNIVERSE="N"
set ARGS=%ARGS% VMMEMORY="128"
set ARGS=%ARGS% VMMAXNUMBER="$(NUM_CPUS)"
set ARGS=%ARGS% VMNETWORKING="N"
REM set ARGS=%ARGS% LOCALCONFIG="http://my.example.com/condor_config.$(FULL_HOSTNAME)"

msiexec /qb /l* condor-install-log.txt /i condor-8.0.0-133173-Windows-x86.msi %ARGS%
```

Each property corresponds to answers that would have been supplied while running the interactive installer. The following is a brief explanation of each property as it applies to unattended installations; see the above explanations for more detail.

NEWPOOL = < Y | N >
: determines whether the installer will create a new pool with the target machine as the central manager.

POOLNAME
: sets the name of the pool, if a new pool is to be created. Possible values are either the name or the empty string "".

RUNJOBS = < N | A | I | C >
: determines when HTCondor will run jobs. This can be set to:

  - Never run jobs (N)
  - Always run jobs (A)
  - Only run jobs when the keyboard and mouse are Idle (I)
  - Only run jobs when the keyboard and mouse are idle and the CPU usage is low (C)

VACATEJOBS = < Y | N >
: determines what HTCondor should do when it has to stop the execution of a user job. When set to Y, HTCondor will vacate the job and start it somewhere else if possible. When set to N, HTCondor will merely suspend the job in memory and wait for the machine to become available again.

SUBMITJOBS = < Y | N >
: will cause the installer to configure the machine as a submit node when set to Y.

CONDOREMAIL
: sets the e-mail address of the HTCondor administrator. Possible values are an e-mail address or the empty string "".

ALLOWREAD
: is a list of names that are allowed to issue READ commands to HTCondor daemons. This value should be set in accordance with the `ALLOW_READ` setting in the configuration file, as described in the *Authorization* section.

ALLOWWRITE
: is a list of names that are allowed to issue WRITE commands to HTCondor daemons. This value should be set in accordance with the `ALLOW_WRITE` setting in the configuration file, as described in the *Authorization* section.

ALLOWADMINISTRATOR
: is a list of names that are allowed to issue ADMINISTRATOR commands to HTCondor daemons. This value should be set in accordance with the `ALLOW_ADMINISTRATOR` setting in the configuration file, as described in the *Authorization* section.

INSTALLDIR
: defines the path to the directory where HTCondor will be installed.

POOLHOSTNAME
: defines the host name of the pool's central manager.

ACCOUNTINGDOMAIN
: defines the accounting (or UID) domain the target machine will be in.

JVMLOCATION
: defines the path to Java virtual machine on the target machine.

SMTPSERVER
: defines the host name of the SMTP server that the target machine is to use to send e-mail.

VMMEMORY
an integer value that defines the maximum memory each VM run on the target machine.

VMMAXNUMBER
an integer value that defines the number of VMs that can be run in parallel on the target machine.

VMNETWORKING = < N | A | B | C >
determines if VM Universe can use networking. This can be set to:

- None (N)
- NAT (A)
- Bridged (B)
- NAT and Bridged (C)

USEVMUNIVERSE = < Y | N >
will cause the installer to enable VM Universe jobs on the target machine.

LOCALCONFIG
defines the location of the local configuration file. The value can be the path to a file on the local machine, or it can be a URL beginning with `http`. If the value is a URL, then the *condor_urlfetch* tool is invoked to fetch configuration whenever the configuration is read.

PERLLOCATION
defines the path to *Perl* on the target machine. This is required in order to use the **vm** universe.

After defining each of these properties for the MSI installer, the installer can be started with the *msiexec* command. The following command starts the installer in unattended mode, and it dumps a journal of the installer's progress to a log file:

```
> msiexec /qb /lxv* condor-install-log.txt /i condor-8.0.0-173133-Windows-x86.msi␣
↪[property=value] ...
```

More information on the features of *msiexec* can be found at Microsoft's website at http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/msiexec.mspx.

Manual Installation of HTCondor on Windows

If you are to install HTCondor on many different machines, you may wish to use some other mechanism to install HTCondor on additional machines rather than running the Setup program described above on each machine.

WARNING: This is for advanced users only! All others should use the Setup program described above.

Here is a brief overview of how to install HTCondor manually without using the provided GUI-based setup program:

The Service
The service that HTCondor will install is called "Condor". The Startup Type is Automatic. The service should log on as System Account, but **do not enable** "Allow Service to Interact with Desktop". The program that is run is *condor_master.exe*.

The HTCondor service can be installed and removed using the `sc.exe` tool, which is included in Windows XP and Windows 2003 Server. The tool is also available as part of the Windows 2000 Resource Kit.

Installation can be done as follows:

```
> sc create Condor binpath= c:\condor\bin\condor_master.exe
```

To remove the service, use:

```
> sc delete Condor
```

The Registry

HTCondor uses a few registry entries in its operation. The key that HTCondor uses is HKEY_LOCAL_MACHINE/Software/Condor. The values that HTCondor puts in this registry key serve two purposes.

1. The values of CONDOR_CONFIG and RELEASE_DIR are used for HTCondor to start its service.

   CONDOR_CONFIG should point to the `condor_config` file. In this version of HTCondor, it **must** reside on the local disk.

   RELEASE_DIR should point to the directory where HTCondor is installed. This is typically C:\Condor, and again, this **must** reside on the local disk.

2. The other purpose is storing the entries from the last installation so that they can be used for the next one.

The File System

The files that are needed for HTCondor to operate are identical to the Unix version of HTCondor, except that executable files end in `.exe`. For example the on Unix one of the files is *condor_master* and on HTCondor the corresponding file is `condor_master.exe`.

These files currently must reside on the local disk for a variety of reasons. Advanced Windows users might be able to put the files on remote resources. The main concern is twofold. First, the files must be there when the service is started. Second, the files must always be in the same spot (including drive letter), no matter who is logged into the machine.

Note also that when installing manually, you will need to create the directories that HTCondor will expect to be present given your configuration. This normally is simply a matter of creating the `log`, `spool`, and `execute` directories. Do not stage other files in any of these directories; any files not created by HTCondor in these directories are subject to removal.

For any installation, HTCondor services are installed and run as the Local System account. Running the HTCondor services as any other account (such as a domain user) is not supported and could be problematic.

CHAPTER

THIRTEEN

FREQUENTLY ASKED QUESTIONS (FAQ)

There are many Frequently Asked Questions maintained on the HTCondor web page, at http://htcondor-wiki.cs.wisc.edu/index.cgi/wiki and on the configuration how-to and recipes page at https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToAdminRecipes

Supported platforms are listed in the *Availability* section. There is also *Platform-Specific Information* available..

CHAPTER

FOURTEEN

VERSION HISTORY AND RELEASE NOTES

14.1 Introduction to HTCondor Versions

This chapter provides descriptions of what features have been added or bugs fixed for each release of HTCondor. The first section describes the HTCondor version numbering scheme, what the numbers mean, and what the different releases are. The rest of the sections each describe the specific releases.

14.1.1 HTCondor Version Number Scheme

We changed the version numbering scheme after the 9.1.3 release: what would have been the next 9.1.x release is now the 9.2.0 release. We made this change to give us additional flexibility in releasing small updates to address specific issues without disturbing the normal development of HTCondor. The version number will still retain the MAJOR.MINOR.PATCH form with slightly different meanings. We have borrowed ideas from Semantic Versioning.

- The MAJOR number increments for each new Long Term Support (LTS) release. A new LTS release may have backward-incompatible changes and may require updates to configuration files. If the current LTS release is 23.0.6, the next one will be 24.0.0. A new LTS release is expected about every twelve months in August. The LTS major version number matches the year of initial release.
- The MINOR number increments each feature release. This number stays at 0 for LTS releases. If the current feature release is 23.2.0, the next one will be 23.3.0. A new feature release is expected every month.
- The PATCH number increments when we have targeted fixes. For the LTS releases, a patch release is expected every month and may occur more frequently if a serious problem is discovered. For the feature releases, the frequency of patch releases depends on the demand for quick updates.

Types of Releases

- An **LTS** release is numbered **X.0.0**, and is a new long-term support release. The previous LTS release is supported for six months after a new LTS version is released. The final feature release undergoes a stabilization effort where the software is run through multiple code quality tools (such as Valgrind) to assure the best possible LTS release. The MAJOR.0.0 version is not released until the stabilization effort is complete. Paid support contracts are only available for the LTS release.
- An **LTS patch** release is numbered **X.0.Z**, and is an update to the LTS major release. The patches are reviewed to ensure correctness and compatibility with the LTS release. These releases contain bug fixes and security updates and are released when a major issue is identified, or just before the next feature release. These releases go through our entire testing process. Large code changes are not permitted in the LTS release. Enhancements are not implemented in the LTS release unless there is minimal impact with a major benefit. Ports to new platforms will appear in the LTS release. The HTCondor team guarantees that patches to the LTS release are compatible.

- A **feature** release is numbered **X.Y.0** and includes one or more new features. The software goes through our entire testing process. We use these releases in production at the Center for High Throughput Computing. These releases contain all the patches from the LTS release and all the patches from the to the feature releases. The HTCondor development team guarantees protocol compatibility between the feature releases and the LTS release. However, changes in behavior may be observed, and adjustments to configuration may be required when new features are implemented.
- A **feature patch** release is numbered **X.Y.Z** and contains targeted patches to address a specific issue with a feature release. specific issue with a feature release. If there is a specific need to be addressed before 23.3.0 is tested and ready, we would issue a 23.2.1 patch release. These releases have undergone code review and light testing. These patch releases are cumulative.

Support Life Cycle

We plan to release a new LTS version every August. The support life cycles are directly related to the release dates.

Table 1: HTCondor Support Life Cycle

Version	Release	End of Regular Support	End of Security Suport
23.x	September 29, 2023	August 2024	August 2024
23.0	September 29, 2023	August 2024	August 2025
10.x	November 10, 2022	September 29, 2023	September 29, 2023
10.0	November 10, 2022	September 29, 2023	August 2024

Repositories

These releases will be served out of three repositories.

- The LTS release and its patches (X.0.Z) are in the existing Stable channel.
- The feature releases (X.Y.0) are in the existing Current channel.
- A new Updates channel will contain quick patch releases (X.Y.Z).

Recommendations

If you are new to HTCondor or require maximum stability in your environment, use an LTS release. Updates to the latest LTS release should be seamless. A new LTS release will appear about every twelve months with clear directions on issues to address when upgrading to the new LTS release.

If you want to take advantage of the latest features, use the feature releases. This is an opportunity see our development directions early, and have some influence on the features being implemented. It is what we use in our production environment.

If you want to run the very latest release, also enable the updates repository to get the targeted fixes. However, these fixes may come frequently, and you may wish to pick and choose which updates to install.

14.2 Upgrading from an 10.0 LTS version to an 23.0 LTS version of HTCondor

Upgrading from a 10.0 LTS version of HTCondor to a 23.0 LTS version will bring new features introduced in the 10.x versions of HTCondor. These new features include the following (note that this list contains only the most significant changes; a full list of changes can be found in the version history: Version 10 Feature Releases):

- A *condor_startd* without any slot types defined will now default to a single partitionable slot rather than a number of static slots equal to the number of cores as it was in previous versions. The configuration template `use FEATURE : StaticSlots` was added for admins wanting the old behavior. (HTCONDOR-2026)
- In an HTCondor Execution Point started by root on Linux, the default for cgroups memory has changed to be enforcing. This means that jobs that use more then their provisioned memory will be put on hold with an appropriate hold message. The previous default can be restored by setting = none on the Execution points. (HTCONDOR-1974)
- Users can now define DAGMan save points to be able to save the state of a DAGs progess to a file and then re-run a DAG from that saved point of progress. (HTCONDOR-1636)
- DAGMan has much better user control of enviroment variables present in the DAGMan job propers environment via *condor_submit_dag*'s new flags (`-include_env` & `-insert_env`) and/or the new DAG file description command `ENV`. (HTCONDOR-1955) (HTCONDOR-1580)
- Added the *condor_qusers* command to monitor and control users at the Access Point. Users disabled at the Access Point are no longer allowed to submit jobs. Jobs submitted before the user was disabled are allowed to run to completion. When a user is disabled, an optional reason string can be provided. (HTCONDOR-1723) (HTCONDOR-1853)
- The *condor_negotiator* now support setting a minimum floor number of cores that any given submitter should get, regardless of their fair share. This can be set or queried via the *condor_userprio* tool, in the same way that the ceiling can be set or get. (HTCONDOR-557)
- Added a `-gpus` option to *condor_status*. With this option *condor_status* will show only machines that have GPUs provisioned; and it will show information about the GPU properties. (HTCONDOR-1958)
- The output of *condor_status* when using the `-compact` option has been improved to show a separate row for the second and subsequent slot type for machines that have multiple slot types. Also the totals now count slots that have the `BackfillSlot` attribute under the `Backfill` or `BkIdle` columns. (HTCONDOR-1957)
- Container universe jobs may now specify the *container_image* to be an image transferred via a file transfer plugin. (HTCONDOR-1820)
- Support for Enterprise Linux 9, Amazon Linux 2023, and Debian 12. (HTCONDOR-1285) (HTCONDOR-1742) (HTCONDOR-1938)
- Administrators can specify a new history file for Access Points that records information about a job for each execution attempt. If enabled then this information can be queried via *condor_history* `-epochs`. (HTCONDOR-1104)
- A single HTCondor pool can now have multiple *condor_defrag* daemons running and they will not interfere with each other so long as each has that select mutually exclusive subsets of the pool. (HTCONDOR-1903)
- Add *condor_test_token* tool to generate a short lived SciToken for testing. (HTCONDOR-1115)
- The job's executable is no longer renamed to `condor_exec.exe`. (HTCONDOR-1227)

Upgrading from a 10.0 LTS version of HTCondor to a 23.0 LTS version will also introduce changes that administrators and users of sites running from an older HTCondor version should be aware of when planning an upgrade. Here is

a list of items that administrators should be aware of. To see if any of the following items will affect an upgrade run `condor_upgrade_check`.

- HTCondor will no longer pass all environment variables to the DAGMan proper manager jobs environment. This may result in DAGMan and its various parts (primarily PRE, POST,& HOLD Scripts) to start failing or change behavior due to missing needed environment variables. To revert back to the old behavior or add the missing environment variables to the DAGMan proper job set the configuration option. (HTCONDOR-1580)
- We added the ability for the *condor_schedd* to track users over time. Once you have upgraded to HTCondor 23, you may no longer downgrade to a version before HTCondor 10.5.0 or HTCondor 10.0.4 LTS. (HTCONDOR-1432)
- Execution Points without any administrator defined slot configuration will now default to creating and utilizing one partitionable slot. This causes Startd `RANK` expressions to have no effect. To revert an Execution Point to use static slots add `use FEATURE:StaticSlots` to the Execution Point configuration. (HTCONDOR-2026)
- The configuration expression constant `CpuBusyTime` no longer represents a time delta but rather a timestamp of when the CPU became busy. The new expression constant `CpuBusyTimer` now represents the time delta of how long a CPU has been busy for. (HTCONDOR-1502)
- The configuration expression constants `ActivationTimer`, `ConsoleBusy`, `CpuBusy`, `CpuIdle`, `JustCPU`, `KeyboardBusy`, `KeyboardNotBusy`, `LastCkpt`, `MachineBusy`, and `NonCondorLoadAvg` no longer exist by default for configuration expressions. To re-enable these constants either add `use FEATURE: POLICY_EXPR_FRAGMENTS` or one of the desktop policies to the configuration. (HTCONDOR-1502)
- The job router configuration macros , , , and are deprecated and will be removed during the lifetime of the HTCondor **V23** feature series. (HTCONDOR-1968)

14.3 Version 23.0 LTS Releases

These are Long Term Support (LTS) versions of HTCondor. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 23.0.y versions. New features will be added in the 23.x.y feature versions.

Warning: The configuration macros JOB_ROUTER_DEFAULTS, JOB_ROUTER_ENTRIES, JOB_ROUTER_ENTRIES_CMD, and JOB_ROUTER_ENTRIES_FILE are deprecated and will be removed for V24 of HTCondor. New configuration syntax for the job router is defined using JOB_ROUTER_ROUTE_NAMES and JOB_ROUTER_ROUTE_<name>. Note: The removal will occur during the lifetime of the HTCondor V23 feature series. (HTCONDOR-1968)

The details of each version are described below.

14.3.1 Version 23.0.8

Release Notes:

- HTCondor version 23.0.8 released on April 11, 2024.

New Features:

- None.

Bugs Fixed:

- Fixed a bug that caused **ssh-agent** processes to be leaked when using *grid* universe remote batch job submission over SSH. (HTCONDOR-2286)

- Fixed a bug where DAGMan would crash when the provisioner node was given a parent node. (HTCONDOR-2291)
- Fixed a bug that prevented the use of `ftp:` URLs in the file transfer plugin. (HTCONDOR-2273)
- Fixed a bug where a job that's matched to an offline slot ad remains idle forever. (HTCONDOR-2304)
- Fixed a bug where the *condor_shadow* would not write a job termination event to the job log for a completed job if the *condor_shadow* failed to reconnect to the *condor_starter* prior to completing cleanup. This would result in DAGMan workflows being stuck waiting forever for jobs to finish. (HTCONDOR-2292)
- Fixed bug where the Shadow failed to write its job ad to when it failed to reconnect to the Starter. (HTCONDOR-2289)
- Fixed a bug in the Windows MSI installer that would cause installation to fail when the install path had a space in the path name, such as when installing to `C:\Program Files` (HTCONDOR-2302)
- Fixed a bug where the was allowed to create job event log information events with newlines in them, which broke the event log parser. (HTCONDOR-2305)
- Fixed `SyntaxWarning` raised by Python 3.12 in **condor_adstash**. (HTCONDOR-2312)
- Improved use of Vault for job credentials. Reject some invalid use cases and avoid redundant work with frequent job submission. (HTCONDOR-2038) (HTCONDOR-2232)
- Fixed an issue where HTCondor could not be installed on Debian or Ubuntu platforms if there was more that one `condor` user in LDAP. (HTCONDOR-2306)

14.3.2 Version 23.0.6

Release Notes:

- HTCondor version 23.0.6 released on March 14, 2024.

New Features:

- Speed up starting of daemons on Linux systems configured with very large number of file descriptors. (HTCONDOR-2270)

Bugs Fixed:

- Fixed bug in DAGMan where nodes that had retries would incorrectly set its descendants to the Futile state if the node job got removed. (HTCONDOR-2240)
- Fixed bug in the event log reader that would rarely cause DAGMan to lose track of a job, and wait forever for a job that had really finished, with DAGMan not realizing that said job had indeed finished. (HTCONDOR-2236)
- Fixed *condor_test_token* to access the SciTokens cache as the correct user when run as root. (HTCONDOR-2241)
- Fixed a bug that caused a crash if a configuration file or submit description file contained an empty multi-line value. (HTCONDOR-2249)
- Fixed a bug where a submit transform or a job router route could crash on a two argument transform statement that had missing arguments. (HTCONDOR-2280)
- Fixed error handing for the `-format` and `-autoformat` options of the *condor_qusers* tool when the argument to those options was not a valid expression. (HTCONDOR-2269)
- Fixed a bug where the **condor_collector** generated an invalid host certificate for itself on macOS. (HTCONDOR-2272)

14.3.3 Version 23.0.4

Release Notes:

- HTCondor version 23.0.4 released on February 8, 2024.

New Features:

- The **condor_starter** will now set the environment variable `NVIDIA_VISIBLE_DEVICES` either to `none` or to a list of the full uuid of each GPU device assigned to the slot. (HTCONDOR-2242)
- When the HTCondor Keyboard daemon (**condor_kbdd**) is installed, a configuration file is included to automatically enable user input monitoring. (HTCONDOR-2255)
- The **condor_starter** can now be configured to capture the stdout and stderr of file transfer plugins and write that output into the StarterLog. (HTCONDOR-1459)
- Updated **condor_upgrade_check** script for better support and maintainability. This update includes new flags/functionality and removal of old checks for upgrading between V9 and V10 of HTCondor. (HTCONDOR-2168)

Bugs Fixed:

- Fixed a bug in the HTCondor Keyboard daemon where activity detected by the X Screen Saver extension was ignored. (HTCONDOR-2255)
- Search engine timeout settings for **condor_adstash** now apply to all search engine operations, not just the initial request to the search engine. (HTCONDOR-2167)
- Ensure Perl dependencies are present for the **condor_gather_info** script. The **condor_gather_info** script now properly reports the User login name. Also, report the contents of `/etc/os-release`. (HTCONDOR-2094)
- The submit language will no longer treat `request_gpu_memory` and `request_gpus_memory` as requests for a custom resource of type `gpu_memory` or `gpus_memory` respectively. (HTCONDOR-2201)
- Fixed bug where DAG node jobs declared inline inside a DAG file would fail to set the Job ClassAd attribute `JobSubmitMethod`. (HTCONDOR-2184)
- Fixed `SyntaxWarning` raised by Python 3.12 in scripts packaged with the Python bindings. (HTCONDOR-2212)

14.3.4 Version 23.0.3

Release Notes:

- HTCondor version 23.0.3 released on January 4, 2024.
- Preliminary support for openSUSE LEAP 15. (HTCONDOR-2156)

New Features:

- Improve `htcondor job status` command to display information about a jobs goodput. (HTCONDOR-1982)
- Added `ROOT_MAX_THREADS` to default value. (HTCONDOR-2137)

Bugs Fixed:

- The file transfer plugin documents that an exit code of 0 is success, 1 is failure, and 2 is reserved for future work to handle the need to refresh credentials. The definition has now changed so that any non-zero exit codes are treated as an error putting the job on hold. (HTCONDOR-2205)
- Fixed a bug where any file I/O error (such as disk full) was ignored by the *condor_starter* when writing the ClassAd file that controlled file transfer plugins. As a result, in rare cases, file transfer plugins could be unknowingly given incomplete sets of files to transfer. (HTCONDOR-2203)

- Fixed a crash in the Python bindings when job submit fails due to any reason. A common reason might be when fails. (HTCONDOR-1931)
- There is a fixed size limit of 5120 bytes for chip commands. The starter now returns an error, and the chirp_client prints out an error when requested to send a chirp command over this limit. Previously, these were silently ignored. (HTCONDOR-2157)
- Fixed a bug where the Python-based HTChirp client had its max line length set much shorter than is allowed by the HTCondor Chirp server. The client now also throws a relevant error when this max limit is hit while sending commands to the server. (HTCONDOR-2142)
- Linux jobs with a invalid `#!` interpreter now get a better error message when the Execution Point is running as root. This was enhanced in 10.0, but a bug prevented the enhancement from fully working on a system installed Execution Point. (HTCONDOR-1698)
- Fixed a bug where the DAGMan job proper for a DAG with a final node could stay stuck in the removed job state. (HTCONDOR-2147)
- Correctly identify `GPUsAverageUsage` and `GPUsMemoryUsage` as floating point values for *condor_adstash*. (HTCONDOR-2170)
- Fixed a bug where *condor_adstash* would get wedged due to a logging failure. (HTCONDOR-2166)
- Updated the usage and man page of the *condor_drain* tool to include information about the `-reconfig-on-completion` option. (HTCONDOR-2164)

14.3.5 Version 23.0.2

Release Notes:

- HTCondor version 23.0.2 released on November 20, 2023.

New Features:

- None.

Bugs Fixed:

- Fixed a bug when Hashicorp Vault is configured to issue data transfer tokens (which is not the default), job submission could hang and then fail. Reverted a change to *condor_submit* that disconnected the output stream of to the user's console, which broke OIDC flow. (HTCONDOR-2078)
- Fixed a bug that could result in job sandboxes not being cleaned up for **batch** grid jobs submitted to a remote cluster. (HTCONDOR-2073)
- Improved cleanup of ssh-agent processes when submitting **batch** grid universe jobs to a remote cluster via ssh. (HTCONDOR-2118)
- Fixed a bug where the *condor_negotiator* could fail to contact a *condor_schedd* that's on the same private network. (HTCONDOR-2115)
- Fixed = `custom` for cgroup v2 systems. (HTCONDOR-2133)
- Implemented support for cgroup v2 systems. (HTCONDOR-2127)
- Fixed a bug in the OAuth and Vault credmons where log files would not rotate according to the configuration. (HTCONDOR-2013)
- Fixed a bug in the *condor_schedd* where it would not create a permanent User record when a queue super user submitted a job for a different owner. This bug would sometimes cause the *condor_schedd* to crash after a job for a new user was submitted. (HTCONDOR-2131)

- Fixed a bug that could cause jobs to be created incorrectly when a using `initialdir` and `max_idle` or `max_materialize` in the same submit file. (HTCONDOR-2092)
- Fixed bug in DAGMan where held jobs that were removed would cause a warning about the internal count of held job procs being incorrect. (HTCONDOR-2102)
- Fixed a bug in *condor_transfer_data* where using the `-addr` flag would automatically apply the `-all` flag to transfer all job data back making the use of `-addr` with a Job ID constraint fail. (HTCONDOR-2105)
- Fixed warnings about use of deprecated HTCondor Python binding methods in the *htcondor dag submit* command. (HTCONDOR-2104)
- Fixed several small bugs with Trust On First Use (TOFU) for SSL authentication. Added configuration parameter , which can be used to prevent tools from prompting the user about trusting the server's SSL certificate. (HTCONDOR-2080)
- Fixed bug in the *condor_userlog* tool where it would crash when reading logs with parallel universe jobs in it. (HTCONDOR-2099)

14.3.6 Version 23.0.1

Release Notes:

- HTCondor version 23.0.1 released on October 31, 2023.
- We added a HTCondor Python wheel for Python 3.12 on PyPI. (HTCONDOR-2117)
- The HTCondor tarballs now contain apptainer version 1.2.4. (HTCONDOR-2111)

New Features:

- None.

Bugs Fixed:

- Fixed a bug introduced in HTCondor 10.6.0 that prevented USE_PID_NAMESPACES from working. (HTCONDOR-2088)
- Fix a bug where HTCondor fails to install on Debian and Ubuntu platforms when the `condor` user is present and the `/var/lib/condor` directory is not. (HTCONDOR-2074)
- Fixed a bug where execution times reported for ARC CE jobs were inflated by a factor of 60. (HTCONDOR-2068)
- Fixed a bug in DAGMan where `Service` nodes that failed caused the DAGMan process to fail an assertion check and crash. (HTCONDOR-2051)
- The job attributes `CpusProvisioned`, `DiskProvisioned`, and `MemoryProvisioned` are now updated for Condor-C and Job Router jobs. (HTCONDOR-2069)
- Updated HTCondor Windows binaries that are statically linked to the curl library to use curl version 8.4.0. The update was due to a report of a vulnerability, CVE-2023-38545, which affects earlier versions of curl. (HTCONDOR-2084)
- Fixed a bug on Windows where jobs would be inappropriately put on hold with an out of memory error if they returned an exit code with high bits set (HTCONDOR-2061)
- Fixed a bug where jobs put on hold by the shadow were not writing their ad to the job epoch history file. (HTCONDOR-2060)
- Fixed a rare race condition where *condor_rm*'ing a parallel universe job would not remove the job if the rm happened after the job was matched but before it fully started (HTCONDOR-2070)

14.3.7 Version 23.0.0

Release Notes:

- HTCondor version 23.0.0 released on September 29, 2023.

New Features:

- A *condor_startd* without any slot types defined will now default to a single partitionable slot rather than a number of static slots equal to the number of cores as it was in previous versions. The configuration template `use FEATURE : StaticSlots` was added for admins wanting the old behavior. (HTCONDOR-2026)
- The `TargetType` attribute is no longer a required attribute in most Classads. It is still used for queries to the *condor_collector* and it remains in the Job ClassAd and the Machine ClassAd because of older versions of HTCondor require it to be present. (HTCONDOR-1997)
- The `-dry-run` option of *condor_submit* will now print the output of a `SEC_CREDENTIAL_STORER` script. This can be useful when developing such a script. (HTCONDOR-2014)
- Added ability to query epoch history records from the Python bindings. (HTCONDOR-2036)
- The default value of will now be visible in *condor_config_val*. The default for will inherit from this value, and thus no `READ` and `CLIENT` will no longer automatically have `CLAIMTOBE`. (HTCONDOR-2047)
- Added new tool *condor_test_token*, which will create a SciToken with configurable contents (including issuer) which will be accepted for a short period of time by the local HTCondor daemons. (HTCONDOR-1115)

Bugs Fixed:

- Fixed a bug that would cause the *condor_startd* to crash in rare cases when jobs go on hold (HTCONDOR-2016)
- Fixed a bug where if a user-level checkpoint could not be transferred from the starter to the AP, the job would go on hold. Now it will retry, or go back to idle. (HTCONDOR-2034)
- Fixed a bug where the *CommittedTime* attribute was not set correctly for Docker Universe jobs doing user level check-pointing. (HTCONDOR-2014)
- Fixed a bug where *condor_preen* was deleting files named '*OfflineAds*' in the spool directory. (HTCONDOR-2019)
- Fixed a bug where the *blahpd* would incorrectly believe that an LSF batch scheduler was not working. (HTCONDOR-2003)
- Fixed the Execution Point's detection of whether libvirt is working properly for the VM universe. (HTCONDOR-2009)
- Fixed a bug where container universe did not work for late materialization jobs submitted to the *condor_schedd* (HTCONDOR-2031)
- Fixed a bug where the *condor_startd* could crash if a new match is made at the end a drain request. (HTCONDOR-2032)

14.4 Version 10 Feature Releases

We release new features in these releases of HTCondor. The details of each version are described below.

14.4.1 Version 10.9.0

Release Notes:

- HTCondor version 10.9.0 released on September 28, 2023.
- This version includes all the updates from *Version 10.0.9*.

New Features:

- None.

Bugs Fixed:

- None.

14.4.2 Version 10.8.0

Release Notes:

- HTCondor version 10.8.0 released on September 14, 2023.
- The packaged builds (RPMs and debs) have been reorganized. We no longer wish to support the ClassAd library and it has been folded into the main `condor` package. The `condor-blahp` and `condor-procd` packages have also been folded into the `condor` package. (HTCONDOR-1981)
- On Debian based systems, the HTCondor's `libexec` directory has moved to the more standard `/usr/libexec/condor`. (HTCONDOR-1981)
- The Debian packaging has been aligned with the RPM packaging. The package names are now `condor` and `minicondor`. The `condor-kbdd` package has been split out, since many installations are server based and do not require the keyboard daemon and all of its dependencies on the X Window system. Also, the `condor-vm-gahp` package has been split out for sites that do not want to support VM Universe and the `libvirt` dependencies that come along with it. (HTCONDOR-1987)
- This version includes all the updates from *Version 10.0.8*.

New Features:

- In an HTCondor Execution Point started by root on Linux, the default for cgroups memory has changed to be enforcing. This means that jobs that use more then their provisioned memory will be put on hold with an appropriate hold message. *condor_q -hold* will show that message. The previous default can be restored by setting = none on the Execution points. (HTCONDOR-1974)
- Added a `-gpus` option to *condor_status*. With this option *condor_status* will show only machines that have GPUs provisioned; and it will show information about the GPU properties. (HTCONDOR-1958)
- The output of *condor_status* when using the `-compact` option has been improved to show a separate row for the second and subsequent slot type for machines that have multiple slot types. Also the totals now count slots that have the `BackfillSlot` attribute under the `Backfill` or `BkIdle` columns. (HTCONDOR-1957)
- Added new DAG command `ENV` for DAGMan. This command allows users to specify environment variables to be added into the DAGMan job proper's environment either by setting values explicitly or getting them from the environment the job is submitted from. (HTCONDOR-1955)

- Improved output for `htcondor dag status` command to include more information about the specified DAG. (HTCONDOR-1951)
- Updated DAGMan to utilize the `-reason` flag to add a message about why a job was removed when DAGMan removes managed jobs via *condor_rm* for some reason. (HTCONDOR-1950)
- Partitionable slots can now be directly claimed by a *condor_schedd* (i.e. the `State` of the partitionable slot changes to `Claimed`). While a slot is claimed, no other *condor_schedd* is able to create new dynamic slots to run jobs. This is controlled by the new configuration parameter and is disabled by default. (HTCONDOR-1824)
- By default, the user event logs are no longer fsync'd by the *condor_schedd*. This should improve the performance of the *condor_schedd*, especially when the user's event logs are on non-solid state disks. There is a knob to revert to the old semantics, ENABLE_USERLOG_FSYNC, which defaults to false. (HTCONDOR-1550)
- A new configuration variable was added to allow administrators to restrict job submission to users that have already been added to the *condor_schedd* using the *condor_qusers* tool. (HTCONDOR-1934)
- Updated *condor_upgrade_check* script to check and warn about known incompatibilities introduced in the feature series for HTCondor `V10` that can cause issues when upgrading to a newer version (i.e. HTCondor `V23`). (HTCONDOR-1960)
- Self-checkpointing jobs may now include the time spent generating successfully-stored checkpoints as part of their *CommittedTime* job ad attribute. (HTCONDOR-1942)

Bugs Fixed:

- Fixed a bug introduced in 10.5.0 that caused jobs to fail to start if they requested an OAuth credential whose service name included an asterisk. (HTCONDOR-1966)
- Fixed bugs in *condor_store_cred* that could cause it to crash or write incorrect data for the pool password. (HTCONDOR-1587)
- Fixed a bug with *condor_ssh_to_job* where it would fail if the Execution point was behind CCB, and the command was run immediately after the job started. (HTCONDOR-1979)
- Some support scripts for the `htcondor annex` command are now properly installed as executable. (HTCONDOR-1984)
- Fixed a bug where *condor_remote_cluster* could get stuck in a loop while installing files into an NFS directory. (HTCONDOR-2023)

14.4.3 Version 10.7.1

- HTCondor version 10.7.1 released on August 9, 2023.

New Features:

- None.

Bugs Fixed:

- Fixed inefficiency in DAGMan setting a nodes descendants to futile status which would result in DAGMan taking an extremely long time when a node fails in a very large and bushy DAG. (HTCONDOR-1945)

14.4.4 Version 10.7.0

Release Notes:

- HTCondor version 10.7.0 released on July 31, 2023.
- This version includes all the updates from *Version 10.0.7*.
- Add support for Debian 12 (bookworm). (HTCONDOR-1938)

New Features:

- A single HTCondor pool can now have multiple *condor_defrag* daemons running and they will not interfere with each other so long as each has that select mutually exclusive subsets of the pool. (HTCONDOR-1903)
- If a job does not define any of the periodic policy expressions (like periodic_hold), HTCondor no longer sets a default value (like false) in the job ad. The system knows that if these aren't set, to take the default action. This removes about 10% of the attributes in a job ad, with corresponding benefits for all consumers of the job ad. (HTCONDOR-1919)
- Added submit command **want_io_proxy**. This replaces the old command **+WantIOProxy**. (HTCONDOR-1875)
- Apptainer is now included in the tarballs. (HTCONDOR-1932)

Bugs Fixed:

- Fixed bug introduced in 10.5.0 on cgroup v1 systems where the user and system CPU time measured was low by a factor of 10,000. (HTCONDOR-1920)
- Fixed a bug introduced in `V10.5.0` of HTCondor where the `.job.ad` and `.machine.ad` failed to be written to a `local` universe jobs scratch directory because of the *condor_starter* having the wrong permissions. (HTCONDOR-1912)
- If the collector is storing offline ads via COLLECTOR_PERSISTENT_AD_LOG the *condor_preen* tool will no longer delete that file (HTCONDOR-1874)
- Fixed a bug where empty execute sandboxes failed to be cleaned up on the Execution Point when using Startd disk enforcement. (HTCONDOR-1821)
- When using Startd disk enforcement, if a *condor_starter* running a container or VM universe job is abruptly killed (like SIGABRT) then the *condor_startd* would fail to cleanup the running docker container or VM and underlying logical volume. (HTCONDOR-1895)

14.4.5 Version 10.6.0

Release Notes:

- HTCondor version 10.6.0 released on June 29, 2023.
- This version includes all the updates from *Version 10.0.6*.

New Features:

- Added the *condor_qusers* command to monitor and control users at the Access Point. Users disabled at the Access Point are no longer allowed to submit jobs. Jobs submitted before the user was disabled are allowed to run to completion. When a user is disabled, an optional reason string can be provided. The reason will be included in the error message from *condor_submit* when submission is refused because the user is disabled. (HTCONDOR-1723) (HTCONDOR-1835)
- Mitigate a memory leak in the *arc_gahp* with libcurl when it uses NSS for security. When an *arc_gahp* process has handled a certain number of commands, a new *arc_gahp* is started and old process exits. The number of commands that triggers a new process is controlled by new configuration parameter . (HTCONDOR-1778)

- Container universe jobs may now specify the *container_image* to be an image transferred via a file transfer plugin. (HTCONDOR-1820)
- Added two new functions for using ClassAd expressions. The `stringListSubsetMatch` and `stringListISubsetMatch` functions can be used to check if all of the members of a stringlist are also in a target stringlist. A single `stringListSubsetMatch` function call can replace a whole set of `stringListMember` calls once the whole pool is updated to 10.6.0. (HTCONDOR-1817)
- Added a new automatic submit file macro `$(JobId)` which expands to the full id of the submitted job. (HTCONDOR-1836)
- The job's executable is no longer renamed to *condor_exec.exe* when the job's sandbox is transferred to the Execution Point. (HTCONDOR-1227)

Bugs Fixed:

- condor_restd service in the htcondor/mini container no longer crashes on startup due to the *en_US.UTF-8* locale being unavailable. (HTCONDOR-1785)
- Fixed a bug that would very rarely cause *condor_wait* to hang forever. (HTCONDOR-1792)

14.4.6 Version 10.5.1

- HTCondor version 10.5.1 released on June 6, 2023.

New Features:

- None.

Bugs Fixed:

- For grid universe jobs of type **batch**, detecting if a Slurm system is functioning now works with older versions of Slurm. (HTCONDOR-1826)

14.4.7 Version 10.5.0

Release Notes:

- HTCondor version 10.5.0 released on June 5, 2023.
- This version includes all the updates from *Version 10.0.4*.
- Add support for Amazon Linux 2023. VOMS authentication is omitted on this platform. (HTCONDOR-1742)

New Features:

- Added new **Save File** functionality to DAGMan which allows users to specify DAG nodes as save points to record the current DAG's progress in a file similar to a rescue file. These files can then be specified with the new *condor_submit_dag* flag `load_save` to re-run the DAG from that point of progression. For more information visit *DAG Save Point Files*. (HTCONDOR-1636)
- The admin knob `SUBMIT_ALLOW_GETENV` when set to false, now allows submit files to use any value but *true* for their `getenv = ...` commands. (HTCONDOR-1671)
- Improved throughput when submitting a large number of ARC CE jobs. Previously, jobs could remain stalled for a long time in the ARC CE server waiting for their input sandbox to be transferred while other were being submitted. (HTCONDOR-1666)
- The *arc_gahp* can now issue multiple HTTPS requests in parallel in different threads. This is controlled by the new configuration parameter . (HTCONDOR-1690)

- The Execute event in the user log now prints out slot name, sandbox path and resource quantities of execution slot. (HTCONDOR-1722)
- Added new submit command `ulog_execute_attrs` for a jobs submit file. This command takes a comma-separated list of machine ClassAd attributes to be written to the user logs execute event. (HTCONDOR-1759)
- Added new DAGMan configuration macro to give a list of machine attributes that will be added to DAGMan submitted jobs for recording in the various produced job ads and userlogs. (HTCONDOR-1717)
- The *condor_transform_ads* tool can now read a configuration file containing `JOB_TRANSFORM_<name>` or `JOB_ROUTER_ROUTE_<name>` and then apply any or all of the transforms declared in that file. This makes it easier to test job transforms before deploying them. (HTCONDOR-1710)
- Linux Cgroup support has been redone in a way that doesn't depend on using the procd. There should be no user visible changes in the usual cases. (HTCONDOR-1589)

Bugs Fixed:

- Expanded default list of environment variables to include in the DAGMan proper manager jobs getenv to include `HOME`, `USER`, `LANG`, and `LC_ALL`. Thus resulting in these variables appearing in the DAGMan manager jobs environment. (HTCONDOR-1725)
- Fixed a bug on cgroup v2 systems where memory limits over 2 gigabytes would not be enforced correctly. (HTCONDOR-1775)
- HTCondor no longer puts jobs using cgroup v1 into the blkio controller. HTCondor never put limits on the i/o, and some kernel version panicked and crashed when they had active jobs in the blkio controller. (HTCONDOR-1786)
- Forced condor_ssh_to_job to never try to use a Control Master, which would break ssh_to_job. Also raised the timeout for ssh_to_job which might be needed for slow WANs. (HTCONDOR-1782)
- Fixed a bug when running with root on a Linux systems with cgroup v1 that would print a warning to the StarterLog claiming Warning: cannot chown /sys/fs/cgroup/cpu,cpuset (HTCONDOR-1672)
- Fixed a bug where *condor_history* would fail to find history files for a remote query if the various history configuration macros were specified with subsystem prefixes i.e. `SCHEDD.HISTORY = /path` (HTCONDOR-1739)
- When started on a systemd system, HTCondor will now wait for the SSSD service to start. Previously it only waited for ypbind. (HTCONDOR-1655)
- Fixed a bug in *condor_preen* that would remove any recorded job epoch history files stored in the spool directory. (HTCONDOR-1738)

14.4.8 Version 10.4.3

Release Notes:

- HTCondor version 10.4.3 released on May 9, 2023.
- Tarballs in this release contain the recent scitokens-cpp 1.0.1 library. (HTCONDOR-1779)

New Features:

- None.

Bugs Fixed:

- The ce-audit collector plug-in should no longer crash. (HTCONDOR-1774)

14.4.9 Version 10.4.2

- HTCondor version 10.4.2 released on May 2, 2023.

New Features:

- None.

Bugs Fixed:

- Fixed a bug introduced in HTCondor 10.0.3 that caused remote submission of **batch** grid universe jobs via ssh to fail when attempting to do file transfer. (HTCONDOR-1747)
- Fixed a bug where the HTCondor-CE would fail to handle any of its jobs after a restart. (HTCONDOR-1755)

14.4.10 Version 10.4.1

Release Notes:

- HTCondor version 10.4.1 released on April 12, 2023.
- Preliminary support for Ubuntu 20.04 (Focal Fossa) on PowerPC (ppc64el). (HTCONDOR-1668)

New Features:

- None.

Bugs Fixed:

- *condor_remote_cluster* now works correctly when the hardware architecture of the remote machine isn't x86_64. (HTCONDOR-1670)

14.4.11 Version 10.4.0

Release Notes:

- HTCondor version 10.4.0 released on April 6, 2023.
- This version includes all the updates from *Version 10.0.3*.
- HTCondor will no longer pass all environment variables to the DAGMan proper manager jobs environment. This may result in DAGMan and its various parts (primarily PRE, POST,& HOLD Scripts) to start failing or change behavior due to missing needed environment variables. To revert back to the old behavior or add the missing environment variables to the DAGMan proper jobs environment set the configuration option. (HTCONDOR-1580)
- The *condor_startd* will no longer advertise *CpuBusy* or *CpuBusyTime* unless the configuration template `use POLICY : DESKTOP` or `use POLICY : UWCS_DESKTOP` is used. Those templates will cause *CpuBusyTime* to be advertised as a time value and not a duration value. The policy expressions in those templates have been modified to account for this fact. If you have written policy expressions of your own that reference *CpuBusyTime* you will need to modify them to use `$(CpuBusyTimer)` from one of those templates or make the equivalent change. (HTCONDOR-1502)

New Features:

- DAGMan no longer sets `getenv = true` in the `.condor.sub` file while adding the ability to better control the environment passed to the DAGMan proper job. `getenv` will default to `CONDOR_CONFIG,_CONDOR_*,PATH, PYTHONPATH,PERL*,PEGASUS_*,TZ` in the `.condor.sub` file which can be appended to via the or the new *condor_submit_dag* flag `include_env`. Also added new *condor_submit_dag* flag `insert_env` to directly set key=value pairs of information into the `.condor.sub` environment. (HTCONDOR-1580)

- New configuration parameter `SEC_SCITOKENS_FOREIGN_TOKEN_ISSUERS` restricts which issuers' tokens will be accepted under `SEC_SCITOKENS_ALLOW_FOREIGN_TOKEN_TYPES`. Updated default values allow EGI CheckIn tokens to be accepted under the SCITOKENS authentication method. (HTCONDOR-1515)
- The *condor_startd* can now be configured to evaluate a set of expressions defined by . For each expression, the last evaluated value will be advertised as well as the time that the evaluation changed to that value. This new generic mechanism was used to add a new slot attribute *NumDynamicSlotsTime* that is the last time a dynamic slot was created or destroyed. (HTCONDOR-1502)
- Add new field `ContainerDuration` to TransferInput attribute of jobs that measure the number of seconds to transfer the Apptainer/Singularity image. (HTCONDOR-1588)
- For grid universe jobs of type **batch**, add detection of when the target batch system is unreachable or not functioning. When this is the case, HTCondor marks the resource as unavailable instead of putting the affected jobs on hold. This matches the behavior for other grid universe job types. Grid ads in the collector now contain attributes `GridResourceUnavailableTimeReason` and `GridResourceUnavailableTimeReasonCode`, which give details about why the remote scheduling system is considered unavailable. (HTCONDOR-1582)
- Added ability for DAGMan to automatically record the Node Retry attempt in that nodes job ad. This is done by setting the new configuration option . (HTCONDOR-1634)

Bugs Fixed:

- Fixed a bug where if the docker command emitted warnings to stderr, the *condor_startd* would not correctly advertise the amount of used image cache. (HTCONDOR-1645)
- Fixed a bug where *condor_history* would fail if the job history file doesn't exist. (HTCONDOR-1578)
- Fixed a bug in the view server where it would assert and exit if the view server stats file are deleted at just the wrong time. (HTCONDOR-1599)
- Fixed a bug where *condor_shadow* was unable to write the job ad to the file when located in condor owned directories such as the spool directory. (HTCONDOR-1631)
- Remove warning when installing HTCondor RPMs on Enterprise Linux 9. (HTCONDOR-1571)

14.4.12 Version 10.3.1

- HTCondor version 10.3.1 released on March 7, 2023.

New Features:

- The *condor_startd* now advertises whether there appears to be a useful /usr/sbin/sshd on the system, in order for *condor_ssh_to_job* to work. (HTCONDOR-1614)

Bugs Fixed:

- None.

14.4.13 Version 10.3.0

Release Notes:

- HTCondor version 10.3.0 released on March 6, 2023.
- This version includes all the updates from *Version 10.0.2*.
- When HTCondor is configured to use cgroups, if the system as a whole is out of memory, and the kernel kills a job with the out of memory killer, HTCondor now checks to see if the job is below the provisioned memory. If so, HTCondor now evicts the job, and marks it as idle, not held, so that it might start again on a machine with

sufficient resources. Previous, HTCondor would let this job attempt to run, hoping the next time the OOM killer fired it would pick a different process. (HTCONDOR-1512)

- This version changes the semantics of the `output_destination` submit command. It no longer sends the files named by the `output` or `error` submit commands to the output destination. Submitters may instead specify those locations with URLs directly. (HTCONDOR-1365)

New Features:

- When HTCondor has root, and is running with cgroups, the cgroup the job is in is writeable by the job. This allows the job (perhaps a glidein) to sub-divide the resource limits it has been given, and allocate subsets of those to its child processes. (HTCONDOR-1496)
- Added capabilities for per job run instance history recording. Where during the *condor_shadow* daemon's shutdown it will write the current job ad to a file designated by and/or a directory specified by . These per run instance job ad records can be read via *condor_history* using the new `-epochs` option. This behavior is not turned on by default. Setting either of the job epoch location config knobs above will turn on this behavior. (HTCONDOR-1104)
- Added new *condor_history* `-search` option that takes a filename to find all matching condor time rotated files `filename.YYYYMMDDTHHMMSS` to read from instead of using any default files. (HTCONDOR-1514)
- Added new *condor_history* `-directory` option to use a history sources alternative configured directory knob such as to search for history. (HTCONDOR-1514)
- Added ability to set a gangliad metrics lifetime (DMAX value) within the metric definition language with the new `Lifetime` keyword. (HTCONDOR-1547)
- Added configuration knob to set the minimum value for gangliads calculated metric lifetime (DMAX value) for all metrics without a specified `Lifetime`. (HTCONDOR-1547)
- Added an attribute to the *condor_schedd* classad that advertises the number of late materialization jobs that have been submitted, but have not yet materialized. The new attribute is called `JobsUnmaterialized` (HTCONDOR-1591)
- The *linux_kernel_tuning_script*, run by the *condor_master* at startup, now tries to increase the value of /proc/sys/fs/pipe-user-pages-soft to 128k, if it was below this. This improves the scalability of the *condor_schedd* when running more than 16k jobs from any one user. (HTCONDOR-1556)
- The *linux_kernel_tuning_script*, run by the *condor_master* at startup, no longer tries to mount the various cgroup filesystems. We assume that any reasonable Linux system will have done this in a manner that it deems appropriate. (HTCONDOR-1528)
- Linux worker nodes now advertise *DockerCachedImageSizeMb*, the number of megabytes that are used in the docker image cache. (HTCONDOR-1494)
- When a file-transfer plug-in aborts due to lack of progress, the message now includes the `https_proxy` (or `http_proxy`) environment variable, and the phrasing has been changed to avoid suggesting that the plug-in actually respected it. (HTCONDOR-1473)

Bugs Fixed:

- Added support for older cgroup v2 systems with missing memory.peak files in the memory controller. (HTCONDOR-1529)
- The HTCondor starter now removes any cgroup that it has created for a job when it exits. (HTCONDOR-1500)
- Fixed bug where `condor_history` would occasionally fail to display all matching user requested job ids. (HTCONDOR-1506)
- Fixed bugs in how the *condor_collector* generated its own CA and host certificate files. Configuration parameter `COLLECTOR_BOOTSTRAP_SSL_CERTIFICATE` now defaults to `True` on Unix platforms. Configuration parame-

ters `AUTH_SSL_SERVER_CERTFILE` and `AUTH_SSL_SERVER_KEYFILE` can now be a list of files. The first pair of files with valid credentials is used. (HTCONDOR-1455)

- Added missing environment variables for the SciTokens plugin. (HTCONDOR-1516)

14.4.14 Version 10.2.5

- HTCondor version 10.2.5 released on February 28, 2023.

New Features:

- None.

-Bugs Fixed:

- Fixed an issue where after a *condor_schedd* restart, the `JobsUnmaterialized` attribute in the *condor_schedd* ad may be an overcount of the number of unmaterialized jobs in rare cases. (HTCONDOR-1606)

14.4.15 Version 10.2.4

Release Notes:

- HTCondor version 10.2.4 released on February 24, 2023.

New Features:

- None.

Bugs Fixed:

- Fixed an issue where after a *condor_schedd* restart, the `JobsUnmaterialized` attribute in the *condor_schedd* ad may be an undercount of the number of unmaterialized jobs for previous submissions. (HTCONDOR-1591)

14.4.16 Version 10.2.3

- HTCondor version 10.2.3 released on February 21, 2023.

New Features:

- Added an attribute to the *condor_schedd* ClassAd that advertises the number of late materialization jobs that have been submitted, but have not yet materialized. The new attribute is called `JobsUnmaterialized`. (HTCONDOR-1591)

Bugs Fixed:

- None.

14.4.17 Version 10.2.2

Release Notes:

- HTCondor version 10.2.2 released on February 7, 2023.

New Features:

- None.

Bugs Fixed:

- Fixed bugs with configuration knob `SINGULARITY_USE_PID_NAMESPACES`. (HTCONDOR-1574)

14.4.18 Version 10.2.1

- HTCondor version 10.2.1 released on January 24, 2023.

New Features:

- Improved scalability of *condor_schedd* when running more than 1,000 jobs from the same user. (HTCONDOR-1549)
- *condor_ssh_to_job* should now work in glidein and other environments where the job or HTCondor is running as a Unix user id that doesn't have an entry in the /etc/passwd database. (HTCONDOR-1543)

Bugs Fixed:

- In the Python bindings, the attribute `ServerTime` is now included in job ads returned by `Schedd.query()`. (HTCONDOR-1531)
- Fixed issue when HTCondor could not be installed on Ubuntu 18.04 (Bionic Beaver). (HTCONDOR-1548)

14.4.19 Version 10.2.0

Release Notes:

- HTCondor version 10.2.0 released on January 5, 2023.
- This version includes all the updates from *Version 10.0.1*.
- We changed the semantics of relative paths in the `output`, `error`, and `transfer_output_remaps` submit file commands. These commands now create the directories named in relative paths if they do not exist. This could cause jobs that used to go on hold (because they couldn't write their `output` or `error` files, or a remapped output file) to instead succeed. (HTCONDOR-1325)
- HTCondor can now put a job in a Linux control (cgroup), not only if it has root privilege, but also if the administrator or some external entity has made the cgroup HTCondor is configured to use writeable by the non-rootly user a personal condor or glidein is running as. (HTCONDOR-1465)
- File-transfer plug-ins may no longer take as long as they like to finish. After seconds, the starter will terminate the transfer and report a time-out failure (with `ETIME`, 62, as the hold reason subcode). (HTCONDOR-1404)

New Features:

- Add support for Enterprise Linux 9 on x86_64 and aarch64 architectures. (HTCONDOR-1285)
- Add support to the *condor_starter* for tracking processes via cgroup v2 on Linux distributions that support cgroup v2. (HTCONDOR-1457)
- The *condor_negotiator* now support setting a minimum floor number of cores that any given submitter should get, regardless of their fair share. This can be set or queried via the *condor_userprio* tool, in the same way that the ceiling can be set or get (HTCONDOR-557)
- Improved the validity testing of the Singularity / Apptainer container runtime software at *condor_startd* startup. If this testing fails, slot attribute `HasSingularity` will be set to `false`, and attribute `SingularityOfflineReason` will contain error information. Also in the event of Singularity errors, more information is recorded into the *condor_starter* log file. (HTCONDOR-1431)
- *condor_q* default behavior of displaying the cumulative run time has changed to now display the current run time for jobs in running, transferring output, and suspended states while displaying the previous run time for jobs in idle or held state unless passed `-cumulative-time` to show the jobs cumulative run time for all runs. (HTCONDOR-1064)

- Docker Universe submit files now support *docker_pull_policy = always*, so that docker will check to see if the cached image is out of date. This increases the network activity, may cause increased throttling when pulling from docker hub, and is recommended to be used with care. (HTCONDOR-1482)
- Added configuration knob . (HTCONDOR-1431)
- *condor_history* will now stop searching history files once all requested job ads are found if passed ClusterIds or ClusterId.ProcId pairs. (HTCONDOR-1364)
- Improved *condor_history* search speeds when searching for matching jobs, matching clusters, and matching owners. (HTCONDOR-1382)
- The local issuer credmon can optionally add group authorizations to users' tokens by setting `LOCAL_CREDMON_AUTHZ_GROUP_TEMPLATE` and `LOCAL_CREDMON_AUTHZ_GROUP_MAPFILE`. (HTCONDOR-1402)
- The `JOB_INHERITS_STARTER_ENVIRONMENT` configuration variable now accepts a list of match patterns just like the submit command `getenv` does. (HTCONDOR-1339)
- Declaring either `container_image` or `docker_image` without a defined `universe` in a submit file will now automatically setup job for respective `universe` based on image type. (HTCONDOR-1401)
- Added new Scheduler ClassAd attribute `EffectiveFlockList` that represents the *condor_collector* addresses that a *condor_schedd* is actively sending flocked jobs. (HTCONDOR-1389)
- Added new DAGMan node status called *Futile* that represents a node that will never run due to the failure of a node that the *Futile* node depends on either directly or indirectly through a chain of **PARENT/CHILD** relationships. Also, added a new ClassAd attribute `DAG_NodesFutile` to count the number of *Futile* nodes in a **DAG**. (HTCONDOR-1456)
- Improved error handling in the *condor_shadow* and *condor_starter* when they have trouble talking to each other. (HTCONDOR-1360)
- Added support for plugins that can perform the mapping of a validated SciToken to an HTCondor canonical user name during security authentication. (HTCONDOR-1463)
- EGI CheckIn tokens can now be used to authenticate via the SCITOKENS authentication method. New configuration parameter `SEC_SCITOKENS_ALLOW_FOREIGN_TOKEN_TYPES` must be set to `True` to enable this usage. (HTCONDOR-1498)

Bugs Fixed:

- Fixed bug where `HasSingularity` would be advertised as true in cases where it wouldn't work. (HTCONDOR-1274)

14.4.20 Version 10.1.3

Release Notes:

- HTCondor version 10.1.3 limited release on November 22, 2022.

New Features:

- Jobs run in Singularity or Apptainer container runtimes now use the SINGULARITY_VERBOSITY flag, which controls the verbosity of the runtime logging to the job's stderr. The default value is "-s" for silent, meaning only fatal errors are logged. (HTCONDOR-1436)
- The PREPARE_JOB and PREPARE_JOB_BEFORE_TRANSFER job hooks can now return a `HookStatusCode` and a `HookStatusMessage` to give better feedback to the user. See the *Startd Cron and Schedd Cron* manual section. (HTCONDOR-1416)

- The local issuer credmon can optionally add group authorizations to users' tokens by setting `LOCAL_CREDMON_AUTHZ_GROUP_TEMPLATE` and `LOCAL_CREDMON_AUTHZ_GROUP_MAPFILE`. (HTCONDOR-1402)

Bugs Fixed:

- None.

14.4.21 Version 10.1.2

- HTCondor version 10.1.2 limited release on November 15, 2022.

New Features:

- OpenCL jobs can now run inside a Singularity container launched by HTCondor if the OpenCL drivers are present on the host in directory `/etc/OpenCL/vendors`. (HTCONDOR-1410)

Bugs Fixed:

- None.

14.4.22 Version 10.1.1

Release Notes:

- HTCondor version 10.1.1 released on November 10, 2022.

New Features:

- Improvements to job hooks, including configuration knob STARTER_DEFAULT_JOB_HOOK_KEYWORD, the new hook PREPARE_JOB_BEFORE_TRANSFER, and the ability to preserve stderr from job hooks into the StarterLog or StartdLog. See the *Hooks* manual section. (HTCONDOR-1400)

Bugs Fixed:

- Fixed bugs in the container universe that prevented apptainer-only systems from running container universe jobs with Docker repository style images (HTCONDOR-1412)

14.4.23 Version 10.1.0

Release Notes:

- HTCondor version 10.1.0 released on November 10, 2022.
- This version includes all the updates from *Version 10.0.0*.

New Features:

- None.

Bugs Fixed:

- None.

14.5 Version 10.0 LTS Releases

These are Long Term Support (LTS) versions of HTCondor. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 10.0.y versions. New features will be added in the 10.x.y feature versions.

The details of each version are described below.

14.5.1 Version 10.0.9

Release Notes:

- HTCondor version 10.0.9 released on September 28, 2023.

New Features:

- Updated *condor_upgrade_check* script to check and warn about known incompatibilities introduced in the feature series for HTCondor `V10` that can cause issues when upgrading to a newer version (i.e. HTCondor `V23`). (HTCONDOR-1960)

Bugs Fixed:

- Fixed *htcondor.htchirp* to find its configuration at *_CONDOR_CHIRP_CONFIG* instead of at *_CONDOR_SCRATCH_DIR/.chirp.config*. (HTCONDOR-2012)
- Fixed a bug that prevented deletion of stored user passwords with *condor_store_cred* on Windows. (HTCONDOR-1998)
- Fixed misaligned pointers issue for the PowerPC architecture in the configuration system. (HTCONDOR-2001)

14.5.2 Version 10.0.8

Release Notes:

- HTCondor version 10.0.8 released on September 14, 2023.

New Features:

- None.

Bugs Fixed:

- Removed cgroup v1 blkio controller support – this prevents a kernel panic in some EL8 kernels. (HTCONDOR-1985)
- Fixed a bug in DAGMan where service nodes that finish before the DAGs end would result in DAGMan crashing due to an assertion failure. (HTCONDOR-1909)
- When the file transfer queue is growing too big, HTCondor sends email to the administrator. Prior versions of HTCondor would send an arbitrarily large number of emails. Now HTCondor will only send one email per day. (HTCONDOR-1937)
- Fixed a bug where *condor_adstash* would not import the OpenSearch library properly. (HTCONDOR-1965)
- Fixed a bug that broke the version check for older versions of the Elasticsearch Python library. (HTCONDOR-1964)
- Fixed a bug in *condor_adstash* that caused a "unexpected keyword argument" error to occur when new attributes needed to be added to the index and when using version 8.0.0 or newer of the Elasticsearch Python library. (HTCONDOR-1930)

- Fixed a bug with parallel universe that would result in the *condor_startd* rejecting start attempts from the *condor_schedd* and causing the *condor_schedd* to crash. (HTCONDOR-1952)
- Preen now preserves all files in the spool directory matching **OfflineLog** so that central managers with multiple active collectors can have offline ads. (HTCONDOR-1933)
- Fixed a bug that could cause *condor_config_val* to crash when there were no configuration files. (HTCONDOR-1954)

14.5.3 Version 10.0.7

Release Notes:

- HTCondor version 10.0.7 released on July 25, 2023.

New Features:

- Improved daemon logging for IDTOKENS authentication to make useful messages more prominent. (HTCONDOR-1776)
- The `-summary` option of *condor_config_val* now works with a remote configuration query when the daemon being queried is version 10.0.7 or later. It behaves like `-dump` when the daemon is older. (HTCONDOR-1879)

Bugs Fixed:

- Fixed bug where condor cron jobs put on hold by the *condor_shadow* or *condor_starter* would never start running again and stay `IDLE` when released from the `HELD` state. (HTCONDOR-1869)
- Remove limit on certificate chain length in SSL authentication. (HTCONDOR-1904)
- Print detailed error message when *condor_remote_cluster* fails to fetch a URL. (HTCONDOR-1884)
- Fixed a bug that caused *condor_preen* to crash if configuration parameter `PREEN_COREFILE_MAX_SIZE` was set to a value larger than 2 gigabytes. (HTCONDOR-1908)
- Fixed a bug where if the $(SPOOL) directory was on a separate file system *condor_preen* would delete the special `lost+found` directory. (HTCONDOR-1906)
- If the collector is storing offline ads via COLLECTOR_PERSISTENT_AD_LOG the *condor_preen* tool will no longer delete that file (HTCONDOR-1874)
- Fixed a bug when creating the default value for where a secondary daemon such as `COLLECTOR01` would not be considered a DC daemon if the primary daemon was not in . (HTCONDOR-1900)

14.5.4 Version 10.0.6

Release Notes:

- HTCondor version 10.0.6 released on June 22, 2023.

New Features:

- Added configuration parameter , which controls whether the client checks the environment variable *X509_USER_PROXY* for the location of a credential to use during SSL authentication with a daemon. (HTCONDOR-1841)
- During SSL authentication, when the client uses a proxy certificate, the server now uses the End Entity certificate's subject as the authenticated identity to map, instead of the proxy certificate's subject. (HTCONDOR-1866)

Bugs Fixed:

- Fixed a bug in the python bindings where some attributes were omitted from accounting ads queried from the *condor_negotiator*. (HTCONDOR-1780)
- Fixed a bug in the python bindings where an incorrect version was being reported. (HTCONDOR-1813)
- The classad functions `anycompare`, `allcompare`, `sum`, `min`, `max`, `avg` and `join` no longer treat a single undefined input as forcing the result to be undefined. `sum`, `min`, `max`, `avg` and `join` will skip over undefined inputs, while `anycompare` and `allcompare` will compare them correctly. (HTCONDOR-1799)
- The submit commands **remote_initialdir**, **transfer_input**, **transfer_output**, and **transfer_error** now work properly when submitting a **batch** grid universe job to a remote system via ssh. (HTCONDOR-1560)
- Fixed bug in `condor_pool_job_report` script that broke the script and outputted error messages about invalid constraint expressions due internal use of `condor_history` specifying a file to read with `-f` flag instead of full `-file`. (HTCONDOR-1812)
- Fixed a bug where the *condor_startd* would sometimes not remove docker images that had been left behind when a *condor_starter* exited abruptly. (HTCONDOR-1814)
- *condor_store_cred* and *condor_credmon_vault* now reuses existing Vault tokens when down scoping access tokens. (HTCONDOR-1527)
- Fixed a missing library import in *condor_credmon_vault*. (HTCONDOR-1527)
- When started on a systemd system, HTCondor will now wait for the SSSD service to start. Previously it only waited for ypbind. (HTCONDOR-1829)

14.5.5 Version 10.0.5

Release Notes:

- HTCondor version 10.0.5 released on June 9, 2023.
- Renamed the `upgrade9to10checks.py` script to `condor_upgrade_check` to match standard HTCSS naming scheme. (HTCONDOR-1828)

New Features:

- None.

Bugs Fixed:

- Fix spurious warning from `condor_upgrade_check` for regular expressions that contain a space. (HTCONDOR-1840)
- `condor_upgrade_check` no longer attempts to check for problems for an HTCondor pool when requesting checks for an HTCondor-CE. (HTCONDOR-1840)

14.5.6 Version 10.0.4

Release Notes:

- HTCondor version 10.0.4 released on May 30, 2023.
- Ubuntu 18.04 (Bionic Beaver) is no longer supported, since its end of life is April 30th, 2023.
- Preliminary support for Ubuntu 20.04 (Focal Fossa) on PowerPC (ppc64le). (HTCONDOR-1668)

New Features:

- Added new script called `upgrade9to10checks.py` to help administrators check for known issues that exist and changes needed for an HTCondor system when upgrading from `V9` to `V10`. This script checks for three well known breaking changes: changing of the default value for , changing to using `PCRE2` for regular expression matching, and changes to how users request GPUs. (HTCONDOR-1658)
- Added configuration parameter , which allows the client to present an X.509 proxy certificate during SSL authentication with a daemon. (HTCONDOR-1781)
- Added `CONFIG_ROOT` configuration variable that is set to the directory of the main configuration file before the configuration files are read. (HTCONDOR-1733)
- Ensure that the SciTokens library can create its cache of token issuer credentials. (HTCONDOR-1757)

Bugs Fixed:

- Fixed a bug where certain errors during file transfer could result in file-transfer processes not being cleaned up. This would manifest as jobs completing successfully, including final file transfer, but ending up without one of their output files (the one the error occurred during). (HTCONDOR-1687)
- Fixed a bug where the *condor_schedd* falsely believed there were too many jobs in the queue and rejected new job submissions based on `MAX_JOBS_SUBMITTED`. (HTCONDOR-1688)
- Fix a bug where SSL authentication would fail when using a daemon's private network address when `PRIVATE_NETWORK_NAME` was configured. (HTCONDOR-1713)
- Fixed a bug that could cause a daemon or tool to crash when attempting SSL or SCITOKENS authentication. (HTCONDOR-1756)
- Fixed a bug where the HTCondor-CE would fail to handle any of its jobs after a restart. (HTCONDOR-1755)
- Fixed a bug where Job Ad Information events weren't always written when using the Job Router. (HTCONDOR-1642)
- Fixed a bug where the submit event wasn't written to the job event log if the job ad didn't contain a `CondorVersion` attribute. (HTCONDOR-1643)
- Fixed a bug where a *condor_schedd* was denied authorization to send reschedule command to a *condor_negotiator* with the IDToken authorization levels recommended in the documentation for setting up a condor pool. (HTCONDOR-1615)
- *condor_remote_cluster* now works correctly when the hardware architecture of the remote machine isn't x86_64. (HTCONDOR-1670)
- Fixed *condor_c-gahp* and *condor_job_router* to submit jobs in the same way as *condor_submit*. (HTCONDOR-1695)
- Fixed a bug introduced in HTCondor 10.0.3 that caused remote submission of **batch** grid universe jobs via ssh to fail when attempting to do file transfer. (HTCONDOR-1747)
- When writing a remove event in JSON, the `ToE.When` field is now seconds since the (Unix) epoch, like all other events. (HTCONDOR-1763)
- Fixed a bug where DAGMan job submission would fail when not using direct submission due to setting a custom job ClassAd attribute with the + syntax in a `VARS` command that doesn't append the variables i.e. `VARS NodeA PREPEND +customAttr="value"` (HTCONDOR-1771)
- The ce-audit collector plug-in should no longer crash. (HTCONDOR-1774)

14.5.7 Version 10.0.3

Release Notes:

- HTCondor version 10.0.3 released on April 6, 2023.
- If you set and use / to mark the beginning and end of a regular expression, the character sequence \\ in the mapfile now passes a single \ to the regular expression engine. This allows you to pass the sequence \/ to the regular expression engine (put \\\/ in the map file), which was not previously possible. If the macro above is set and you have a \\ in your map file, you will need to replace it with \\\\. (HTCONDOR-1573)
- For *condor_annex* users: Amazon Web Services is deprecating the Node.js 12.x runtime. If you ran the *condor_annex* setup command with a previous version of HTCondor, you'll need to update your setup. Go to the AWS CloudFormation console and look for the stack named `HTCondorAnnex-LambdaFunctions`. (You may have to switch regions.) Click on that stack's radio button, hit the delete button in the table header, and confirm. Wait for the delete to finish. Then run `condor_annex -aws-region region-name-N -setup` for the region. Repeat for each region of interest. (HTCONDOR-1627).

New Features:

- Allow remote submission of **batch** grid universe jobs via ssh to work with sites that were configured with the old *bosco_cluster* tool. (HTCONDOR-1632)

Bugs Fixed:

- Fixed two problems with GPU metrics. First, fixed a bug where reconfiguring a *condor_startd* caused GPU metrics to stop being reported. Second, fixed a bug where GPU (core) utilization could be wildly over-reported. (HTCONDOR-1660)
- Fix bug, introduced in HTCondor version 10.0.2, that prevented new installations of HTCondor from working on Debian or Ubuntu. (HTCONDOR-1689)
- Fixed bug where a *condor_dagman* node with `RETRY` capabilities would instantly restart that node every time it saw a job proc failure. This would result in nodes with multi-proc jobs to resubmit the entire node multiple times causing internal issues for DAGMan. (HTCONDOR-1607)
- Fixed a rare bug in the late materialization code that could cause a *condor_schedd* crash. (HTCONDOR-1581)
- Fixed bug where the *condor_shadow* would crash during job removal. (HTCONDOR-1585)
- Fixed a bug where two *condor_schedd* daemons in a High Availability configuration could be active at the same time. (HTCONDOR-1590)
- Improved the HTCondor's systemd configuration to not start HTCondor until the system attempts (and mostly likely succeeds) to mount remote filesystems. (HTCONDOR-1594)
- Fixed a bug where the *condor_master* of a glidein submitted to SLURM via HTCondor-CE would try to talk to the *condor_gridmanager* of the HTCondor-CE. (HTCONDOR-1604)
- Fixed a bug in the *condor_schedd* that could result in the `TotalSubmitProcs` attribute of a late materialization job being set to a value smaller than the correct value shortly after the *condor_schedd* was restarted. (HTCONDOR-1603)
- If a job's requested credentials are not available when the job is about to start, the job is now placed on hold. (HTCONDOR-1600)
- Fixed a bug that would cause the *condor_schedd* to hang if an invalid condor cron argument was submitted (HTCONDOR-1624)
- Fixed a bug where cron jobs put on hold due to invalid time specifications would be unable to be removed from the job queue with tools. (HTCONDOR-1629)

- Fixed how the *condor_gridmanager* handles failed ARC CE jobs. Before, it would endlessly re-query the status of jobs that failed during submission to the LRMS behind ARC CE. If ARC CE reports a job as FAILED because the job exited with a non-zero exit code, the *condor_gridmanager* now treats it as completed. (HTCONDOR-1583)
- Fixed a bug where values specified with **arc_rte** in the job's submit description weren't properly sent to the ARC CE service. (HTCONDOR-1648)
- Fixed a bug that can cause a daemon to crash during SciTokens authentication if the configuration parameter `SCITOKENS_SERVER_AUDIENCE` isn't set. (HTCONDOR-1652)

14.5.8 Version 10.0.2

Release Notes:

- HTCondor version 10.0.2 released on March 2, 2023.
- HTCondor Python wheel is now available for Python 3.11 on PyPI. (HTCONDOR-1586)
- The macOS tarball is now being built on macOS 11. (HTCONDOR-1610)

New Features:

- Added configuration option called to allow a transfer output remap to create directories in allowed places if they do not exist at transfer output time. (HTCONDOR-1480)
- Improved scalability of *condor_schedd* when running more than 1,000 jobs from the same user. (HTCONDOR-1549)
- *condor_ssh_to_job* should now work in glidein and other environments where the job or HTCondor is running as a Unix user id that doesn't have an entry in the /etc/passwd database. (HTCONDOR-1543)
- VM universe jobs are now configured to pass through the host CPU model to the VM. This change enables VMs with newer kernels (such as Enterprise Linux 9) to operate in VM Universe. (HTCONDOR-1559)
- The *condor_remote_cluster* command was updated to fetch the Alma Linux tarballs for Enterprise Linux 8 and 9. (HTCONDOR-1562)

Bugs Fixed:

- In the python bindings, the attribute `ServerTime` is now included in job ads returned by `Schedd.query()` to support Fifemon. (HTCONDOR-1531)
- Fixed issue when HTCondor could not be installed on Ubuntu 18.04 (Bionic Beaver). (HTCONDOR-1548)
- Attempting to use a file-transfer plug-in that doesn't exist is no longer silently ignored. This could happen due to different bug, also fixed, where plug-ins specified only in `transfer_output_remaps` were not automatically added to a job's requirements. (HTCONDOR-1501)
- Fixed a bug where **condor_now** could not use the resources freed by evicting a job if its procID was 1. (HTCONDOR-1519)
- Fixed a bug that caused the *condor_startd* to exit when thinpool provisioned filesystems were enabled. (HTCONDOR-1524)
- Fixed a bug causing a Python warning when installing on Ubuntu 22.04. (HTCONDOR-1534)
- Fixed a bug where the *condor_history* tool would crash when doing a remote query with a constraint expression or specified job IDs. (HTCONDOR-1564)

14.5.9 Version 10.0.1

Release Notes:

- HTCondor version 10.0.1 released on January 5, 2023.

New Features:

- Add support for Ubuntu 22.04 LTS (Jammy Jellyfish). (HTCONDOR-1304)
- HTCondor now includes a file transfer plugin that support `stash://` and `osdf://` URLs. (HTCONDOR-1332)
- The Windows installer now uses the localized name of the Users group so that it can be installed on non-English Windows platforms. (HTCONDOR-1474)
- OpenCL jobs can now run inside a Singularity container launched by HTCondor if the OpenCL drivers are present on the host in directory `/etc/OpenCL/vendors`. (HTCONDOR-1410)
- The `CompletionDate` attribute of jobs is now undefined until such time as the job completes previously it was 0. (HTCONDOR-1393)

Bugs Fixed:

- Fixed a bug where Debian, Ubuntu and other Linux platforms with swap accounting disabled in the kernel would never put a job on hold if it exceeded RequestMemory and MEMORY_LIMIT_POLICY was set to hard or soft. (HTCONDOR-1466)
- Fixed a bug where using the `-forcex` option with *condor_rm* on a scheduler universe job could cause a *condor_schedd* crash. (HTCONDOR-1472)
- Fixed bugs in the container universe that prevented apptainer-only systems from running container universe jobs with Docker repository style images. (HTCONDOR-1412)
- Docker universe and container universe job that use the docker runtime now detect when the Unix uid or gid has the high bit set, which docker does not support. (HTCONDOR-1421)
- Grid universe **batch** works again on Debian and Ubuntu. Since 9.5.0, some required files had been missing. (HTCONDOR-1475)
- Fixed bug in the curl plugin where it would crash on Enterprise Linux 8 systems when using a file:// url type. (HTCONDOR-1426)
- Fixed bug in where the multi-file curl plugin would fail to timeout due lack of upload or download progress if a large amount of bytes where transferred at some point. (HTCONDOR-1403)
- Fixed bug where the multi-file curl plugin would fail to receive a SciToken if it was in raw format rather than json. (HTCONDOR-1447)
- Fixed a bug that prevented the starter from properly mounting thinpool provisioned ephemeral scratch directories. (HTCONDOR-1419)
- Fixed a bug where SSL authentication with the *condor_collector* could fail when the provided hostname is not a DNS CNAME. (HTCONDOR-1443)
- Fixed a Vault credmon bug where tokens were being refreshed too often. (HTCONDOR-1017)
- Fixed a Vault credmon bug where the CA certificates used were not based on the HTCondor configuration. (HTCONDOR-1179)
- Fixed the *condor_gridmanager* to recognize when it has the final data for an ARC job in the FAILED status with newer versions of ARC CE. Before, the *condor_gridmanager* would leave the job marked as RUNNING and retry querying the ARC CE server endlessly. (HTCONDOR-1448)
- Fixed AES encryption failures on macOS Ventura. (HTCONDOR-1458)

- Fixed a bug that would cause tools that have the `-printformat` argument to segfault when the format file contained a `FIELDPREFIX`, `FIELDSUFFIX`, `RECORDPREFIX` or `RECORDSUFFIX`. (HTCONDOR-1464)
- Fixed a bug in the `RENAME` command of the transform language that could result in a crash of the *condor_schedd* or *condor_job_router*. (HTCONDOR-1486)
- For tarball installations, the *condor_configure* script now configures HTCondor to use user based security. (HTCONDOR-1461)

14.5.10 Version 10.0.0

Release Notes:

- HTCondor version 10.0.0 released on November 10, 2022.

New Features:

- The default for `TRUST_DOMAIN`, which is used by with IDTOKEN authentication has been changed to `$(UID_DOMAIN)`. If you have already created IDTOKENs for use in your pool, you should configure `TRUST_DOMAIN` to the issuer value of a valid token. (HTCONDOR-1381)
- The *condor_transform_ads* tool now has a `-jobtransforms` argument that reads transforms from the configuration. This provides a convenient way to test the `JOB_TRANSFORM_<NAME>` configuration variables. (HTCONDOR-1312)
- Added new automatic configuration variable `DETECTED_CPUS_LIMIT` which gets set to the minimum of `DETECTED_CPUS` from the configuration and `OMP_NUM_THREADS` and `SLURM_CPU_ON_NODES` from the environment. (HTCONDOR-1307)

Bugs Fixed:

- Fixed a bug where if a job created a symbolic link to a file, the contents of that file would be counted in the job's *DiskUsage*. Previously, symbolic links to directories were (correctly) ignored, but not symbolic links to files. (HTCONDOR-1354)
- Fixed a bug where if SINGULARITY_TARGET_DIR is set, condor_ssh_to job would start the interactive shell in the root directory of the job, not in the current working directory of the job. (HTCONDOR-1406)
- Suppressed a Singularity or Apptainer warning that would appear in a job's stderr file, warning about the inability to set the HOME environment variable if the job or the system explicitly tried to set it. (HTCONDOR-1386)
- Fixed a bug where on certain Linux kernels, the ProcLog would be filled with thousands of errors of the form "Internal cgroup error when retrieving iowait statistics". This error was harmless, but filled the ProcLog with noise. (HTCONDOR-1385)
- Fixed bug where certain **submit file** variables like `accounting_group` and `accounting_group_user` couldn't be declared specifically for DAGMan jobs because DAGMan would always write over the variables at job submission time. (HTCONDOR-1277)
- Fixed a bug where SciTokens authentication wasn't available on macOS and Python wheels distributions. (HTCONDOR-1328)
- Fixed job submission to newer ARC CE releases. (HTCONDOR-1327)
- Fixed a bug where a pre-created security session may not be used when connecting to a daemon over IPv6. The peers would do a full round of authentication and authorization, which may fail. This primarily happened with both peers had `PREFER_IPV4` set to `False`. (HTCONDOR-1341)
- The *condor_negotiator* no longer sends the admin capability attribute of machine ads to the *condor_schedd*. (HTCONDOR-1349)

- Fixed a bug in DAGMan where **Node** jobs that could not write to their **UserLog** would cause the **DAG** to get stuck indefinitely while waiting for pending **Nodes**. (HTCONDOR-1305)
- Fixed a bug where `s3://` URLs host or bucket names shorter than 14 characters caused the shadow to dump core. (HTCONDOR-1378)
- Fixed a bug in the hibernation code that caused HTCondor to ignore the active Suspend-To-Disk option. (HTCONDOR-1357)
- Fixed a bug where some administrator client tools did not properly use the remote administrator capability (configuration parameter `SEC_ENABLE_REMOTE_ADMINISTRATION`). (HTCONDOR-1371)
- When a `JOB_TRANSFORM_*` transform changes an attribute at submit time in a late materialization factory, it no longer marks that attribute as fixed for all jobs. This change makes it possible for a transform to modify rather than simply replacing an attribute that that the user wishes to vary per job. (HTCONDOR-1369)
- Fixed bug where **Collector**, **Negotiator**, and **Schedd** core files that are naturally large would be deleted by *condor_preen* because the file sizes exceeded the max file size. (HTCONDOR-1377)
- Fixed a bug that could cause a daemon or tool to crash when connecting to a daemon using a security session. This particularly affected the *condor_schedd*. (HTCONDOR-1372)
- Fixed a bug that could cause digits to be truncated reading resource usage information from the job event log via the Python or C++ APIs for reading event logs. Note this only happens for very large values of requested or allocated disk, memory. (HTCONDOR-1263)
- Fixed a bug where GPUs that were marked as OFFLINE in the **Startd** would still be available for matchmaking in the `AvailableGPUs` attribute. (HTCONDOR-1397)
- The executables within the tarball distribution now use `RPATH` to find shared libraries. Formerly, `RUNPATH` was used and tarballs became susceptible to failures when independently compiled HTCondor libraries were present in the `LD_LIBRARY_PATH`. (HTCONDOR-1405)

CHAPTER

FIFTEEN

COMMAND REFERENCE MANUAL (MAN PAGES)

HTCondor ships with many command line tools. While the number may seem overwhelming at first, they can be divided into a few groups:

Fig. 1: A map of all the tools

Commands that manage jobs:

> *condor_rm*, *condor_submit*, *condor_submit_dag*, *condor_suspend*, *condor_continue*, *condor_hold*, *condor_release*, *condor_transfer_data*, *condor_q condor_qedit*, *condor_history*

Commands for managing execution points:

> *condor_off*, *condor_on*, *condor_restart*, *condor_drain*, *condor_now*, *condor_vacate*, *condor_config_val*, *condor_reconfig*, *condor_status*

Commands for working with running jobs:

> *condor_ssh_to_job*, *condor_tail*, *condor_evicted_files*, *condor_chirp*, *condor_vacate_job*

Commands for debugging and testing:

> *classad_eval*, *condor_version*, *condor_who*, *condor_top*, *condor_fetchlog*, *condor_transform_ads*, *condor_gpu_discovery*, *condor_power_state*

Commands for managing submitters:

> *condor_userprio*, *condor_qusers*

15.1 *classad_eval*

Evaluate the given ClassAd expression(s) in the context of the given ClassAd attributes, and prints the result in ClassAd format.

15.1.1 Synopsis

classad_eval -help

classad_eval [**-[ad]-file** <*file-name*>] [**-target-file** <*file-name*>] <*ad* | *assignment* | *expression* | **-quiet**>+

15.1.2 Description

classad_eval is designed to help you understand and debug ClassAd expressions. You can supply a ClassAd on the command-line, or via a file, as context for evaluating the expression. You may also construct a ClassAd one argument at a time, with assignments.

By default, **classad_eval** will print the ClassAd context used to evaluate the expression before printing the result of the first expression, and for every expression with a new ClassAd thereafter. You may suppress this behavior with the `-quiet` flag, which replaces an ad, assignment, or expression, and quiets every expression after it on the command line.

Attributes specified on the command line, including those specified as part of a complete ad, replace attributes in the context ad, which starts empty. You can't remove attributes from the context ad, but you can set them to `undefined`.

Options, flags, and arguments may be freely intermixed, and take effect in order.

Note that **classad_eval** uses the `new` ClassAd syntax: ClassAds specified in a file must be surrounded by square brackets and attribute-value pairs must be separated by semicolons. For compability with `condor_q -long:new` and `condor_status -long:new`, *classad_eval* will use only the first ClassAd if passed a ClassAd list of them.

15.1.3 Examples

Almost every ad, assignment, or expression will require you to single quote them. There are some exceptions; for instance, the following two commands are equivalent:

```
$ classad_eval 'a = 2' 'a * 2'
$ classad_eval a=2 a*2
```

You can specify attributes for the context ad in three ways:

```
$ classad_eval '[ a = 2; b = 2 ]' 'a + b'
$ classad_eval 'a = 2; b = 2' 'a + b'
$ classad_eval 'a = 2' 'b = 2' 'a + b'
```

You need not supply an empty ad for expressions that don't reference attributes:

```
$ classad_eval 'strcat("foo", "bar")'
```

If you want to evaluate an expression in the context of the job ad, first store the job ad in a file:

```
$ condor_q -l:new 1227.2 > job.ad
$ classad_eval -quiet -file job.ad 'JobUniverse'
```

You can extract a machine ad in a similar way:

```
$ condor_status -l:new slot1@exec-17 > machine.ad
$ classad_eval -quiet -file machine.ad 'Rank'
```

You may evaluate an expression in order to check a match by using the `-target-file` option:

```
$ condor_q -l:new 1227.2 > job.ad
$ condor_status -l:new exec-17 > machine.ad
$ classad_eval -quiet -my-file job.ad -target-ad machine.ad 'MY.requirements'␣
↪'TARGET.requirements'
```

Assignments (including whole ClassAds) are all merged into the context ad:

```
$ classad_eval 'x = y' 'x' 'y = 7' 'x' '[ x = 6; z = "foo"; ]' 'x'
[ x = y ]
undefined
[ y = 7; x = y ]
7
[ z = "foo"; x = 6; y = 7 ]
6
```

You can suppress printing the context ad partway through:

```
$ classad_eval 'x = y' 'x' -quiet 'y = 7' 'x' '[ x = 6; z = "foo"; ]' 'x'
[ x = y ]
undefined
7
6
```

15.1.4 Exit Status

Returns 0 on success.

15.1.5 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.2 *ClassAds*

The ClassAd language consists of two parts: structured data (called "ClassAds"), and expressions.

HTCondor uses ClassAds to describe various things, primarily machines and jobs; it uses expressions as constraints for querying ClassAds, and for defining what it means for two ClassAds to match each other.

15.2.1 Data Syntax

A ClassAd is a list of attribute-value pairs, separated by newlines. Values may be booleans, integers, reals, strings, dictionaries, lists, or the special values UNDEFINED and ERROR. Dictionaries are marked by square brackets and lists by braces; dictionaries separate elements with semicolons, and lists separate elements with commas.

```
attribute_name  = "attribute-value"
pi              = 3.141
count           = 3
list            = { "red", "green", "blue" }
dictionary      = [ type = "complex"; real = 7.75; imaginary = -3 ]
```

(continues on next page)

(continued from previous page)

```
structured_attr = [ hostnames = { "submit-1", "submit", "submit1" };
                    ip = "127.0.0.1"; port = "9618" ]
```

For the list of ClassAd attributes generated by HTCondor, see https://htcondor.readthedocs.io/en/latest/classad-attributes/index.html.

15.2.2 Expression Syntax

An expression consists of literals (from the data syntax) and attribute references composed with operators and functions. The value of a ClassAd attribute may be an expression.

```
MY.count < 10 && regexp( ".*example.*", attribute_name )
```

Attribute References

An attribute reference always includes an attribute name. In HTCondor, when determining if two ClassAds match, an expression may specify which ad's value is used by prefixing it with `MY.` or `TARGET.`. Attribute references are case-insensitive.

```
MY.count
TARGET.machine
```

An element of a dictionary is referenced by using the subscript operator (`[]`) with an expression that evaluates to a string *or* with a dot (`.`), as follows:

```
MY.structured_attr.hostnames
MY.structured_attr["hostnames"]
```

Note that the following references the attribute named by the attribute `hostnames`, not the attribute named hostnames:

```
MY.structured_attr[hostnames]
```

List elements are referenced by an expression that evaluates to an integer, where the first element in the list is numbered 0. For example, if `colors = { [ x = "1" ], [ x = "2", y = "3" ] }`, then `MY.structure_attr.colors[0]` results in `[ x = "1" ]`.

UNDEFINED and ERROR

The ClassAd language includes two special values, UNDEFINED and ERROR. An attribute may be set to either explicitly, but these values typically result from referring to an attribute that doesn't exist, or asking for something impossible:

```
undefined_reference = MY.undefined_attribute
explicitly_undefined = UNDEFINED
one_error_value = "three" * 7
another_error_value = 1.3 / 0
```

Most expressions that refer to values that are UNDEFINED will evaluate to UNDEFINED. The same applies to ERROR.

Operators

The operators *, /, + and - operate arithmetically, on integers and reals.

The comparison operators ==, !=, <=, <, >= and > operate on booleans, integers, reals and strings. String comparison is case-insensitive. Comparing a string and a non-string value results in ERROR.

The special comparison operator =?= is like == except in the following two ways: it is case-sensitive when comparing strings; and, when comparing values to UNDEFINED, results in FALSE instead of UNDEFINED. (If comparing UNDEFINED to itself, the operator =?= results in TRUE).

The special comparison operator =!= is the negation of =?=.

The logical operators && and || operate on integers and reals; non-zero is true, and zero is false.

The ternary operator `x ? y : z` operates on expressions.

The default operator `x ?: z` returns `x` if `x` is defined and `z` otherwise.

The `IS` and `ISNT` operators are synonyms for =?= and =!=.

Functions

Function name are case-insensitive. Unless otherwise noted, if any of a function's arguments are UNDEFINED or ERROR, so is the result. If an argument's type is noted, the function will return ERROR unless the argument has that type.

integer INT(*expr*)
: If *expr* is numeric, return the closest integer. If *expr* evaluates to a string, attempt to convert the string to an integer. Return ERROR if the string is not an integer, or if *expr* is neither numeric nor a string.

boolean MEMBER(*expr*, list *l*)
: Returns TRUE if *expr* is equal, as defined by the operator ==, to any member of the list *l*, or FALSE if it isn't.

boolean REGEXP(string *pattern*, string *target*[, string *options*])
: Return TRUE if the PCRE regular expression *pattern* matches *target*, or FALSE if it doesn't. Return ERROR if *pattern* is a not a valid regular expression. If specified, *options* is a PCRE option string (one or more of `f`, `i`, `s`, `m`, and `g`). See the *Specification* section for details.

list SPLIT(string *s*[, string *tokens*])
: Separate *s* by whitespace or comma, or instead by any of the characters in *tokens*, if specified, and return the result as a list of strings.

boolean STRINGLISTIMEMBER(string *s*, string *list*[, string *tokens*])
: Equivalent to `MEMBER( *s*, SPLIT( *list*, *tokens* ))`.

string SUBSTR(string *s*, integer *offset*[, integer *length*])
: Returns the substring of *s* from *offset* to the end of the string, or instead for *length* characters, if specified. The first character in *s* is at position 0. If *offset* is negative, the substring begins *offset* characters before the end of the string. If *length* is negative, the substring ends that many characters before the end of the string. If the substring contains no characters, return the empty string. Thus, the following two calls both return the string "78":

```
substr( "0123456789", 7, 2 )
substr( "0123456789", -3, -1 )
```

All ClassAd functions are defined in the references below.

Reserved Words

The words UNDEFINED, ERROR, IS, ISNT, TRUE, FALSE, MY, TARGET, and PARENT may not be used as attribute names.

15.2.3 Testing ClassAd Expressions

Use *classad_eval* to test ClassAd expressions. For instance, if you want to test to see if a regular expression matches some fixed string, you could check in the following way (on Linux or Mac; the quoting rules are different on Windows):

```
$ classad_eval 'regexp( ".*tr.*", "string" )'
[ ]
true
```

This prints out the ClassAd used as context in the evaluation (in this case, there wasn't one, so it's the empty ad) and the result.

15.2.4 Examples

These examples assume a Linux shell environment and a working HTCondor pool.

Selecting a Slot based on Job ID

If job 288.7 is running:

```
$ condor_status -const 'JobId == "288.7"'
```

Selecting Jobs based on Execute Machine

If jobs are running on the machine example-execute-node:

```
$ condor_q -all -const 'regexp( "@example-execute-node$", RemoteHost )'
```

String Manipulation

In this example, an administrator has just added twelve new hosts to the pool – compute-296 to compute-307 – and wants to see if they've started running jobs yet.

```
$ condor_status -const␣
↪'296 <= int(substr( Machine, 8 )) && int(substr( Machine, 8 )) <= 307'
```

You could also write this as follows:

```
$ condor_status -const␣
↪'296 <= int(split(Machine, "-")[1]) && int(split(Machine, "-")[1]) <= 307'
```

Selecting Machines with a Particular File-Transfer Plugin

If you're considering using the gdrive file-transfer plugin, and you'd like to see which machines have it, select from the slot ads based on the corresponding attribute, but only print out the machine name, and then throw away the duplicates:

```
$ condor_status -af Machine \
    -const 'StringListIMember( "gdrive", HasFileTransferPluginMethods )' \
    | uniq
```

You could instead use a constraint to ignore dynamic slots for a report on the resources available to run jobs which require the gdrive plugin. Note that you can also use expressions when formatting the output. In this case, it's just to make the output prettier.

```
$ condor_status -af Machine CPUs Memory Disk \
    '(GPUs =!= undefined && GPUs >= 1) ? CUDACapability : "[no GPUs]"' \
    -const␣
↪'SlotType =!= "Dynamic" && StringListIMember( "gdrive", HasFileTransferPluginMethods )'
```

15.2.5 Specification

For use in HTCondor, including a complete list of functions, see https://htcondor.readthedocs.io/en/latest/classads/classad-mechanism.html.

For the language specification, see https://research.cs.wisc.edu/htcondor/classad/refman/.

15.2.6 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.3 *condor_adstash*

Gather schedd and/or startd job history ClassAds and push them via a search engine or file interface.

15.3.1 Synopsis

condor_adstash [**--help**]

condor_adstash [**--process_name** *NAME*] [**--standalone**] [**--sample_interval** *SECONDS*] [**--checkpoint_file** *PATH*] [**--log_file** *PATH*] [**--log_level** *LEVEL*] [**--threads** *THREADS*] [**--interface** *{null,elasticsearch,jsonfile}*] [**--collectors** *COLLECTORS*] [**--schedds** *SCHEDDS*] [**--startds** *STARTDS*] [**--schedd_history**] [**--startd_history**] [**--ad_file** *PATH*] [**--schedd_history_max_ads** *NUM_ADS*] [**--startd_history_max_ads** *NUM_ADS*] [**--schedd_history_timeout** *SECONDS*] [**--startd_history_timeout** *SECONDS*] [**--se_host** *HOST[:PORT]*] [**--se_url_prefix** *PREFIX*] [**--se_username** *USERNAME*] [**--se_use_https**] [**--se_timeout** SECONDS] [**--se_bunch_size** *NUM_DOCS*] [**--es_index_name** *INDEX_NAME*] [**--se_no_log_mappings**] [**--se_ca_certs** *PATH*] [**--json_dir** *PATH*]

15.3.2 Description

condor_adstash is a tool that assists in monitoring usage by gathering job ClassAds (typically from *condor_schedd* and/or *condor_startd* history queries) and pushing the ClassAds as documents to some target (typically Elasticsearch).

Unless run in `--standalone` mode, *condor_adstash* expects to be invoked as a daemon by a *condor_master*, i.e. *condor_adstash* should be invoked in standalone mode when run on the command-line. Whether invoked by *condor_master* or run standalone, *condor_adstash* gets its configuration, in increasing priority, from the HTCondor configuration macros beginning with `ADSTASH_` (when `--process_name` is not provided), then environment variables, and finally command-line options.

condor_adstash must be able to write its `--checkpoint_file` to a persistent location so that duplicate job ClassAds are not fetched from the daemons' histories in consecutive polls.

A named Elasticsearch index will be created if it doesn't exist, and may be modified if new fields (corresponding to ClassAd attribute names) need to be added. It is up to the administrator of the Elasticsearch instance to install rollover policies (e.g. ILM) on the named index and/or to set up the index as an alias.

15.3.3 Options

-h, --help
Display the help message and exit.

--process_name *PREFIX*
Give *condor_adstash* a different name for looking up HTCondor configuration and environment variable values (see examples).

--standalone
Run condor_adstash in standalone mode (runs once, does not attempt to contact *condor_master*)

--sample_interval *SECONDS*
Number of seconds between polling the list(s) of daemons (ignored in standalone mode)

--checkpoint_file *PATH*
Location of checkpoint file (will be created if missing)

--log_file *PATH*
Location of log file

--log_level *LEVEL*
Log level (uses Python logging library levels: CRITICAL/ERROR/WARNING/INFO/DEBUG)

--threads *THREADS*
Number of parallel threads to use when polling for job ClassAds and when pushing documents to Elasticsearch

--interface *{null,elasticsearch,opensearch,jsonfile}*
Push ads via the chosen interface

15.3.4 ClassAd source options

--schedd_history
Poll and push *condor_schedd* job histories

--startd_history
Poll and push *condor_startd* job histories

--ad_file *PATH*
Load Job ClassAds from a file instead of querying daemons (Ignores *--schedd_history* and *--startd_history*.)

15.3.5 Options for HTCondor daemon (Schedd, Startd, etc.) history sources

--collectors *COLLECTORS*
Comma-separated list of *condor_collector* addresses to contact to locate *condor_schedd* and *condor_startd* daemons

--schedds *SCHEDDS*
Comma-separated list of *condor_schedd* names to poll job histories from

--startds *STARTDS*
Comma-separated list of *condor_startd* machines to poll job histories from

--schedd_history_max_ads *NUM_ADS*
Abort after reading NUM_ADS from a *condor_schedd*

--startd_history_max_ads *NUM_ADS*
Abort after reading NUM_ADS from a *condor_startd*

--schedd_history_timeout *SECONDS*
Abort if reading from a *condor_schedd* takes more than this many seconds

--startd_history_timeout *SECONDS*
Abort if reading from a *condor_startd* takes more than this many seconds

15.3.6 Search engine (Elasticsearch, OpenSearch, etc.) interface options

--se_host *HOST[:PORT]*
Search engine host:port

--se_url_prefix *PREFIX*
Search engine URL prefix

--se_username *USERNAME*
Search engine username

--se_use_https
Use HTTPS when connecting to search engine

--se_timeout *SECONDS*
Max time to wait for search engine queries

--se_bunch_size *NUM_DOCS*
Group ads in bunches of this size to send to search engine

--se_index_name *INDEX_NAME*
Push ads to this search engine index or alias

--se_no_log_mappings
Don't write a JSON file with mappings to the log directory

--se_ca_certs ***PATH***
Path to root certificate authority file (will use certifi's CA if not set)

15.3.7 JSON file interface options

--json_dir ***PATH***
Directory to store JSON files, which are named by timestamp

15.3.8 Examples

Running *condor_adstash* in standalone mode on the command-line will result in *condor_adstash* reading its configuration from the current HTCondor configuration:

```
$ condor_adstash --standalone
```

By default, *condor_adstash* looks for HTCondor configuration variables with names are prefixed with `ADSTASH_`, e.g. `ADSTASH_READ_SCHEDDS = *`. These values can be overridden on the command-line:

```
$ condor_adstash --standalone --schedds=myschedd.localdomain
```

condor_adstash configuration variables can be also be named using custom prefixes, with the prefix passed in using `-\-process_name=PREFIX`. For example, if the HTCondor configuration contained `FOO_SCHEDD_HISTORY = False` and `FOO_STARTD_HISTORY = True`, *condor_adstash* can be invoked to read these instead of `ADSTASH_SCHEDD_HISTORY` and `ADSTASH_STARTD_HISTORY`:

```
$ condor_adstash --standalone --process_name=FOO
```

Providing a `PREFIX` to `--process_name` that does not match any HTCondor configuration variables will cause *condor_adstash* to fallback to a default set of configuration values, which may be useful in debugging.

The configuration values that *condor_adstash* reads from the current HTCondor configuration can be previewed by printing the help message. The values will be listed as the default values for each command-line option:

```
$ condor_adstash --help
$ condor_adstash --process_name=FOO --help
```

15.4 *condor_advertise*

Send a ClassAd to the *condor_collector* daemon

15.4.1 Synopsis

condor_advertise [**-help | -version**]

condor_advertise [**-pool** *centralmanagerhostname[:portname]*] [**-debug**] [**-tcp**] [**-udp**] [**-multiple**] [*update-command [classad-filename]*]

15.4.2 Description

condor_advertise sends one or more ClassAds to the *condor_collector* daemon on the central manager machine. The optional argument *update-command* says what daemon type's ClassAd is to be updated; if it is absent, it assumed to be the update command corresponding to the type of the (first) ClassAd. The optional argument *classad-filename* is the file from which the ClassAd(s) should be read. If *classad-filename* is omitted or is the dash character ('-'), then the ClassAd(s) are read from standard input. You must specify *update-command* if you do not want to read from standard input.

When **-multiple** is specified, multiple ClassAds may be published. Publishing many ClassAds in a single invocation of *condor_advertise* is more efficient than invoking *condor_advertise* once per ClassAd. The ClassAds are expected to be separated by one or more blank lines. When **-multiple** is not specified, blank lines are ignored (for backward compatibility). It is best not to rely on blank lines being ignored, as this may change in the future.

The *update-command* may be one of the following strings:

> UPDATE_STARTD_AD UPDATE_SCHEDD_AD UPDATE_MASTER_AD UPDATE_GATEWAY_AD UPDATE_CKPT_SRVR_AD UPDATE_NEGOTIATOR_AD UPDATE_HAD_AD UPDATE_AD_GENERIC UPDATE_SUBMITTOR_AD UPDATE_COLLECTOR_AD UPDATE_LICENSE_AD UPDATE_STORAGE_AD

condor_advertise can also be used to invalidate and delete ClassAds currently held by the *condor_collector* daemon. In this case the *update-command* will be one of the following strings:

> INVALIDATE_STARTD_ADS INVALIDATE_SCHEDD_ADS INVALIDATE_MASTER_ADS INVALIDATE_GATEWAY_ADS INVALIDATE_CKPT_SRVR_ADS INVALIDATE_NEGOTIATOR_ADS INVALIDATE_HAD_ADS INVALIDATE_ADS_GENERIC INVALIDATE_SUBMITTOR_ADS INVALIDATE_COLLECTOR_ADS INVALIDATE_LICENSE_ADS INVALIDATE_STORAGE_ADS

For any of these INVALIDATE commands, the ClassAd in the required file will look like the following:

```
MyType = "Query"
TargetType = "Machine"
Name = "condor.example.com"
Requirements = Name == "condor.example.com"
```

The definition for `MyType` is always `Query`. `TargetType` is set to the `MyType` of the ad to be deleted. This `MyType` is `DaemonMaster` for the *condor_master* ClassAd, `Machine` for the *condor_startd* ClassAd, `Scheduler` for the *condor_schedd* ClassAd, and `Negotiator` for the *condor_negotiator* ClassAd.

`Requirements` is an expression evaluated within the context of ads of `TargetType`. When `Requirements` evaluates to `True`, the matching ad is invalidated. A full example is given below.

15.4.3 Options

-help
Display usage information

-version
Display version information

-debug
Print debugging information as the command executes.

-multiple
Send more than one ClassAd, where the boundary between ClassAds is one or more blank lines.

-pool ***centralmanagerhostname[:portname]***
Specify a pool by giving the central manager's host name and an optional port number. The default is the `COLLECTOR_HOST` specified in the configuration file.

-tcp
Use TCP for communication. Used by default if `UPDATE_COLLECTOR_WITH_TCP` is true.

-udp
Use UDP for communication.

15.4.4 General Remarks

The job and machine ClassAds are regularly updated. Therefore, the result of *condor_advertise* is likely to be overwritten in a very short time. It is unlikely that either HTCondor users (those who submit jobs) or administrators will ever have a use for this command. If it is desired to update or set a ClassAd attribute, the *condor_config_val* command is the proper command to use.

Attributes are defined in Appendix A of the HTCondor manual.

For those administrators who do need *condor_advertise*, the following attributes may be included:

`DaemonStartTime` `UpdateSequenceNumber`

If both of the above are included, the *condor_collector* will automatically include the following attributes:

`UpdatesTotal` `UpdatesLost` `UpdatesSequenced` `UpdatesHistory`

Affected by `COLLECTOR_DAEMON_HISTORY_SIZE` .

15.4.5 Examples

Assume that a machine called condor.example.com is turned off, yet its *condor_startd* ClassAd does not expire for another 20 minutes. To avoid this machine being matched, an administrator chooses to delete the machine's *condor_startd* ClassAd. Create a file (called `remove_file` in this example) with the three required attributes:

```
MyType = "Query"
TargetType = "Machine"
Name = "condor.example.com"
Requirements = Name == "condor.example.com"
```

This file is used with the command:

```
$ condor_advertise INVALIDATE_STARTD_ADS remove_file
```

15.4.6 Exit Status

condor_advertise will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure. Success means that all ClassAds were successfully sent to all *condor_collector* daemons. When there are multiple ClassAds or multiple *condor_collector* daemons, it is possible that some but not all publications succeed; in this case, the exit status is 1, indicating failure.

15.5 *condor_annex*

Add cloud resources to the pool.

15.5.1 Synopsis

condor_annex -help

condor_annex [**-aws-region** <*region*>] **-setup** *[FROM INSTANCE|[/full/path/to/access/key/file [/full/path/to/secret/key/file]]]*

condor_annex [**-aws-on-demand**] **-annex-name** <*name of the annex*> **-count** <*integer number of instances*> [**-aws-on-demand-***] [**common options**]

condor_annex [**-aws-spot-fleet**] **-annex-name** <*name of the annex*> **-slots** <*integer weight*> [**-aws-spot-fleet-***] [**common options**]

condor_annex -annex-name <*name of the annex*> **-duration** *hours*

condor_annex [**-annex-name** <*name of the annex*>] **-status** [**-classad**]

condor_annex -check-setup

condor_annex <condor_annex options> **status** <condor_status options>

15.5.2 Description

condor_annex adds clouds resources to the pool. ("The pool" is determined in the usual manner for HTCondor daemons and tools.) This version supports only Amazon Web Services ('AWS'). To add "on-demand" instances, use the third form listed above; to add "spot" instances, use the fourth. For an explanation of terms, consult either the HTCondor manual in the *Cloud Computing* chapter or the AWS documentation.

Using *condor_annex* with AWS requires a one-time setup procedure performed by invoking *condor_annex* with the **-setup** flag (the second form listed above). You may check if this procedure has been performed with the **-check-setup** flag (the seventh form listed above). If you use the setup flag on an instance whose role gives it sufficient privileges, you may, instead of specifying your API keys, pass `FROM INSTANCE` to **-setup** to ask *condor_annex* to use the instance's role credentials.

To reset the lease on an existing annex, invoke *condor_annex* with only the **-annex-name** option and **-duration** flag (the fifth form listed above).

To determine which of the instances previously requested for a particular annex are not currently in the pool, invoke *condor_annex* with the **-status** flag and the **-annex-name** option (the sixth form listed above). The output of this command is intended to be human-readable; specifying the **-classad** flag will produce the same information in ClassAd format. If you omit **-annex-name**, information for all annexes will be returned.

Starting in 8.7.3, you may instead invoke *condor_annex* with **status** as a command argument (the eighth form listed above). This will cause *condor_annex* to use *condor_status* to present annex instance data. Arguments and options on the command line after **status** will be passed unmodified to *condor_status*, but not all arguments and options will

behave as expected. (See below.) *condor_annex* will construct an ad for each annex instance and pass that information to *condor_status*; *condor_status* will (unless you specify otherwise using its command line) query the collector for more information about the instances. Information from the collector will be presented as usual; instances which did not have ads in the collector will be presented last, in their own table. These instances can not be presented in the usual way because the annex instance ads generated by *condor_annex* do not (and can not) have the same information in them as ads generated by a *condor_startd* running in the instance. See the *condor_status* manual page for details about the "merge" mode of *condor_status* used by this command argument. Note that both *condor_annex* and *condor_status* have **-annex-name** options; if you're interested in a particular annex, put this flag on the command line before the **status** command argument to avoid confusing results.

Common options are listed first, followed by options specific to AWS, followed by options specific to AWS' on-demand instances, followed by options specific to AWS' spot instances, followed by options intended for use by experts.

15.5.3 Options

-help
Print a usage reminder.

-setup ***[/full/path/to/access/key/file/full/path/to/secret/key/file]***
Do the first-time setup.

-duration ***hours***
Set the maximum lease duration in decimal *hours*. After this amount of time, all instances will terminated, regardless of their idleness. Defaults to 50 minutes.

-idle ***hours***
Set the maximum idle duration in decimal *hours*. An instance idle for longer than this duration will terminate itself. Defaults to 15 minutes.

-yes
Start the annex automatically without a yes/no confirmation prompt.

-tag ***name value***
Add a tag named *name* with value *value* to each instance in the requested annex. Only works at annex creation. This option may be specified more than once.

-config-dir ***/full/path/to/directory***
Copy the contents of */full/path/to/directory* to each instance's configuration directory.

-owner ***owner[, owner]****
Configure the annex so that only *owner* may start jobs there. By default, configure the annex so that only the user running *condor_annex* may start jobs there.

-no-owner
Configure the annex so that anyone in the pool may use the annex.

-aws-region ***region***
Specify the region in which to create the annex.

-aws-user-data ***user-data***
Set the instance user data to *user-data*.

-aws-user-data-file ***/full/path/to/file***
Set the instance user data to the contents of the file */full/path/to/file*.

-aws-default-user-data ***user-data***
Set the instance user data to *user-data*, if it's not already set. Only applies to spot fleet requests.

-aws-default-user-data-file ***/full/path/to/file***
Set the instance user data to the contents of the file */full/path/to/file*, if it's not already set. Only applies to spot fleet requests.

-aws-on-demand-instance-type ***instance-type***
This annex will requests instances of type *instance-type*. The default for v8.7.1 is 'm4.large'.

-aws-on-demand-ami-id ***ami-id***
This annex will start instances of the AMI *ami-id*. The default for v8.7.1 is 'ami-35b13223', a GPU-compatible Amazon Linux image with HTCondor pre-installed.

-aws-on-demand-security-group-ids ***group-id[,group-id]***
This annex will start instances with the listed security group IDs. The default is the security group created by **-setup**.

-aws-on-demand-key-name ***key-name***
This annex will start instances with the key pair named *key-name*. The default is the key pair created by **-setup**.

-aws-spot-fleet-config-file ***/full/path/to/file***
Use the JSON blob in */full/path/to/file* for the spot fleet request.

-aws-access-key-file ***/full/path/to/access-key-file***
Experts only.

-aws-secret-key-file ***/full/path/to/secret-key-file***
Experts only.

-aws-ec2-url ***https://ec2.<region>.amazonaws.com***
Experts only.

-aws-events-url ***https://events.<region>.amazonaws.com***
Experts only.

-aws-lambda-url ***https://lambda.<region>.amazonaws.com***
Experts only.

-aws-s3-url ***https://s3.<region>.amazonaws.com***
Experts only.

-aws-spot-fleet-lease-function-arn ***sfr-lease-function-arn***
Developers only.

-aws-on-demand-lease-function-arn ***odi-lease-function-arn***
Developers only.

-aws-on-demand-instance-profile-arn ***instance-profile-arn***
Developers only.

15.5.4 General Remarks

Currently, only AWS is supported. The AMI configured by setup runs HTCondor v8.6.10 on Amazon Linux 2016.09, and the default instance type is "m4.large". The default AMI has the appropriate drivers for AWS' GPU instance types.

15.5.5 Examples

To start an on-demand annex named 'MyFirstAnnex' with one core, using the default AMI and instance type, run

```
$ condor_annex -count 1 -annex-name MyFirstAnnex
```

You will be asked to confirm that the defaults are what you want.

As of 2017-04-17, the following example will cost a minimum of $90.

To start an on-demand annex with 100 GPUs that job owners 'big' and 'little' may use (be sure to include yourself!), run

```
$ condor_annex -count 100 -annex-name MySecondAnnex \
  -aws-on-demand-instance-type p2.xlarge -owner "big, little"
```

15.5.6 Exit Status

condor_annex will exit with a status value of 0 (zero) on success.

15.6 *condor_check_password*

Examine HTCondor key files, looking for keys that prior version of HTCondor will not fully read.

15.6.1 Synopsis

condor_check_password <-h | --help>

condor_check_password [**--truncate**] [*key*]

15.6.2 Description

Versions of HTCondor before 8.9.12 contained contained a bug in the code used to read the pool password (hence the name of the tool): in some cases the read would be truncated before end of the file. Because the same code is used to read IDTOKENS signing keys, this bug affects the IDTOKENS authorization method, as well.

There was no backwards-compatible fix: versions 8.9.12 and later may read the same file differently than earlier versions, meaning that tokens issued before 8.9.12 may not be recognized by later versions.

This tool detects key files which will not be fully read by earlier versions of HTCondor. IDTOKENS generated by such a key will not be accepted by later versions (which read the whole key file). If you choose to truncate these files on disk, later version of HTCondor will read only the same bits as earlier versions, allowing them to accept tokens issued by earlier versions, at the cost of weakening your pool's resistance to brute-force attacks.

By default, this tool checks all the key files that will be found by the current HTCondor configuration; you may specify a specific *key* or *key*s to check, instead.

15.6.3 Options

-h, --help
Print a usage reminder.

--truncate
When a potentially insecure key is encountered, truncate it to match the behavior prior to version 8.9.12.

15.6.4 Exit Status

Exits with code 0 if there were no signing keys to check or if all of the checked keys were OK. Exits with code 1 if at least one checked key was not OK. Exits non-zero if a problem was encountered along the way.

15.7 *condor_check_userlogs*

Check job event log files for errors

15.7.1 Synopsis

condor_check_userlogs *UserLogFile1* [*UserLogFile2 . . . UserLogFileN*]

15.7.2 Description

condor_check_userlogs is a program for checking a job event log or a set of job event logs for errors. Output includes an indication that no errors were found within a log file, or a list of errors such as an execute or terminate event without a corresponding submit event, or multiple terminated events for the same job.

condor_check_userlogs is especially useful for debugging *condor_dagman* problems. If *condor_dagman* reports an error it is often useful to run *condor_check_userlogs* on the relevant log files.

15.7.3 Exit Status

condor_check_userlogs will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.8 *condor_chirp*

Access files or job ClassAd from an executing job

15.8.1 Synopsis

condor_chirp <Chirp-Command>

15.8.2 Description

condor_chirp is not intended for use as a command-line tool. It is most often invoked by an HTCondor job, while the job is executing. It accesses files or job ClassAd attributes on the access point. Files can be read, written or removed. Job attributes can be read, and most attributes can be updated.

When invoked by an HTCondor job, the command-line arguments describe the operation to be performed. Each of these arguments is described below within the section on Chirp Commands. Descriptions using the terms local and remote are given from the point of view of the executing job.

If the input file name for **put** or **write** is a dash, *condor_chirp* uses standard input as the source. If the output file name for **fetch** is a dash, *condor_chirp* writes to standard output instead of a local file.

Jobs that use *condor_chirp* must have the attribute `WantIOProxy` set to `True` in the job ClassAd. To do this, place

```
want_io_proxy = true
```

in the submit description file of the job.

condor_chirp only works for jobs run in the vanilla, parallel and java universes.

15.8.3 Chirp Commands

fetch *RemoteFileName LocalFileName*
Copy the *RemoteFileName* from the access point to the execute machine, naming it *LocalFileName*.

put [-mode *mode*] [-perm *UnixPerm*] *LocalFileName RemoteFileName*
Copy the *LocalFileName* from the execute machine to the submit machine, naming it *RemoteFileName*. The optional **-perm** *UnixPerm* argument describes the file access permissions in a Unix format; 660 is an example Unix format.

The optional **-mode** *mode* argument is one or more of the following characters describing the *RemoteFileName* file: `w`, open for writing; `a`, force all writes to append; `t`, truncate before use; `c`, create the file, if it does not exist; `x`, fail if `c` is given and the file already exists.

remove *RemoteFileName*
Remove the *RemoteFileName* file from the access point.

get_job_attr *JobAttributeName*
Prints the named job ClassAd attribute to standard output.

set_job_attr *JobAttributeName AttributeValue*
Sets the named job ClassAd attribute with the given attribute value.

get_job_attr_delayed *JobAttributeName*
Prints the named job ClassAd attribute to standard output, potentially reading the cached value from a recent set_job_attr_delayed.

set_job_attr_delayed *JobAttributeName AttributeValue*
Sets the named job ClassAd attribute with the given attribute value, but does not immediately synchronize the value with the submit side. It can take 15 minutes before the synchronization occurs. This has much less overhead than the non delayed version. With this option, jobs do not need ClassAd attribute `WantIOProxy` set. With this option, job attribute names are restricted to begin with the case sensitive substring `Chirp`.

ulog ***Message***
Appends *Message* to the job event log.

read [-offset ***offset*****] [-stride** ***length skip*****]** ***RemoteFileName Length***
Read *Length* bytes from *RemoteFileName*. Optionally, implement a stride by starting the read at *offset* and reading *length* bytes with a stride of *skip* bytes.

write [-offset ***offset*****] [-stride** ***length skip*****]** ***RemoteFileName LocalFileName*** **[*numbytes***
] Write the contents of *LocalFileName* to *RemoteFileName*. Optionally, start writing to the remote file at *offset* and write *length* bytes with a stride of *skip* bytes. If the optional *numbytes* follows *LocalFileName*, then the write will halt after *numbytes* input bytes have been written. Otherwise, the entire contents of *LocalFileName* will be written.

rmdir [-r] ***RemotePath***
Delete the directory specified by *RemotePath*. If the optional **-r** is specified, recursively delete the entire directory.

getdir [-l] ***RemotePath***
List the contents of the directory specified by *RemotePath*. If *-l* is specified, list all metadata as well.

whoami
Get the user's current identity.

whoareyou ***RemoteHost***
Get the identity of *RemoteHost*.

link [-s] ***OldRemotePath NewRemotePath***
Create a hard link from *OldRemotePath* to *NewRemotePath*. If the optional *-s* is specified, create a symbolic link instead.

readlink ***RemoteFileName***
Read the contents of the file defined by the symbolic link *RemoteFileName*.

stat ***RemotePath***
Get metadata for *RemotePath*. Examines the target, if it is a symbolic link.

lstat ***RemotePath***
Get metadata for *RemotePath*. Examines the file, if it is a symbolic link.

statfs ***RemotePath***
Get file system metadata for *RemotePath*.

access ***RemotePath Mode***
Check access permissions for *RemotePath*. *Mode* is one or more of the characters `r`, `w`, `x`, or `f`, representing read, write, execute, and existence, respectively.

chmod ***RemotePath UnixPerm***
Change the permissions of *RemotePath* to *UnixPerm*. *UnixPerm* describes the file access permissions in a Unix format; 660 is an example Unix format.

chown ***RemotePath UID GID***
Change the ownership of *RemotePath* to *UID* and *GID*. Changes the target of *RemotePath*, if it is a symbolic link.

lchown ***RemotePath UID GID***
Change the ownership of *RemotePath* to *UID* and *GID*. Changes the link, if *RemotePath* is a symbolic link.

truncate ***RemoteFileName Length***
Truncates *RemoteFileName* to *Length* bytes.

utime *RemotePath AccessTime ModifyTime*
Change the access to *AccessTime* and modification time to *ModifyTime* of *RemotePath*.

15.8.4 Examples

To copy a file from the access point to the execute machine while the user job is running, run

```
$ condor_chirp fetch remotefile localfile
```

To print to standard output the value of the `Requirements` expression from within a running job, run

```
$ condor_chirp get_job_attr Requirements
```

Note that the remote (submit-side) directory path is relative to the submit directory, and the local (execute-side) directory is relative to the current directory of the running program.

To append the word "foo" to a file called `RemoteFile` on the submit machine, run

```
$ echo foo | condor_chirp put -mode wa - RemoteFile
```

To append the message "Hello World" to the job event log, run

```
$ condor_chirp ulog "Hello World"
```

15.8.5 Exit Status

condor_chirp will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.9 *condor_configure*

Configure or install HTCondor

15.9.1 Synopsis

condor_configure or **condor_install** [–**help**] [–**usage**]

condor_configure or **condor_install** [--**install[=<path/to/release>]**] [--**install-dir=<path>**] [--**prefix=<path>**] [--**local-dir=<path>**] [--**make-personal-condor**] [--**bosco**] [--**type = < submit, execute, manager >**] [--**central-manager = < hostname>**] [--**owner = < ownername >**] [--**maybe-daemon-owner**] [--**install-log = < file >**] [--**overwrite**] [--**ignore-missing-libs**] [--**force**] [--**no-env-scripts**] [--**env-scripts-dir = < directory >**] [--**backup**] [--**credd**] [--**verbose**]

15.9.2 Description

condor_configure and *condor_install* refer to a single script that installs and/or configures HTCondor on Unix machines. As the names imply, *condor_install* is intended to perform a HTCondor installation, and *condor_configure* is intended to configure (or reconfigure) an existing installation. Both will run with Perl 5.6.0 or more recent versions.

condor_configure (and *condor_install*) are designed to be run more than one time where required. It can install HTCondor when invoked with a correct configuration via

```
$ condor_install
```

or

```
$ condor_configure --install
```

or, it can change the configuration files when invoked via

```
$ condor_configure
```

Note that changes in the configuration files do not result in changes while HTCondor is running. To effect changes while HTCondor is running, it is necessary to further use the *condor_reconfig* or *condor_restart* command. *condor_reconfig* is required where the currently executing daemons need to be informed of configuration changes. *condor_restart* is required where the options –**make-personal-condor** or –**type** are used, since these affect which daemons are running.

Running *condor_configure* or *condor_install* with no options results in a usage screen being printed. The –**help** option can be used to display a full help screen.

Within the options given below, the phrase release directories is the list of directories that are released with HTCondor. This list includes: `bin`, `etc`, `examples`, `include`, `lib`, `libexec`, `man`, `sbin`, `sql` and `src`.

15.9.3 Options

-help
: Print help screen and exit

-usage
: Print short usage and exit

-install[=<path/to/release>]
: Perform installation, assuming that the current working directory contains the release directory, if the optional =<`path/to/release`> is not specified. Without further options, the configuration is that of a Personal HTCondor, a complete one-machine pool. If used as an upgrade within an existing installation directory, existing configuration files and local directory are preserved. This is the default behavior of *condor_install*.

-install-dir=<path>
: Specifies the path where HTCondor should be installed or the path where it already is installed. The default is the current working directory.

-prefix=<path>
: This is an alias for **-install-dir**.

-local-dir=<path>
: Specifies the location of the local directory, which is the directory that generally contains the local (machine-specific) configuration file as well as the directories where HTCondor daemons write their run-time information (`spool`, `log`, `execute`). This location is indicated by the `LOCAL_DIR` variable in the configuration file. When installing (that is, if **-install** is specified), *condor_configure* will

properly create the local directory in the location specified. If none is specified, the default value is given by the evaluation of `$(RELEASE_DIR)/local.$(HOSTNAME)`.

During subsequent invocations of *condor_configure* (that is, without the -install option), if the -local-dir option is specified, the new directory will be created and the `log`, `spool` and `execute` directories will be moved there from their current location.

-make-personal-condor
Installs and configures for Personal HTCondor, a fully-functional, one-machine pool.

-bosco
Installs and configures Bosco, a personal HTCondor that submits jobs to remote batch systems.

-type= < submit, execute, manager >
One or more of the types may be listed. This determines the roles that a machine may play in a pool. In general, any machine can be a submit and/or execute machine, and there is one central manager per pool. In the case of a Personal HTCondor, the machine fulfills all three of these roles.

-central-manager=<hostname>
Instructs the current HTCondor installation to use the specified machine as the central manager. This modifies the configuration variable `COLLECTOR_HOST` to point to the given host name. The central manager machine's HTCondor configuration needs to be independently configured to act as a manager using the option **-type=manager**.

-owner=<ownername>
Set configuration such that HTCondor daemons will be executed as the given owner. This modifies the ownership on the `log`, `spool` and `execute` directories and sets the `CONDOR_IDS` value in the configuration file, to ensure that HTCondor daemons start up as the specified effective user. The section on security within the HTCondor manual discusses UIDs in HTCondor. This is only applicable when *condor_configure* is run by root. If not run as root, the owner is the user running the *condor_configure* command.

-maybe-daemon-owner
If **-owner** is not specified and no appropriate user can be found to run Condor, then this option will allow the daemon user to be selected. This option is rarely needed by users but can be useful for scripts that invoke condor_configure to install Condor.

-install-log=<file>
Save information about the installation in the specified file. This is normally only needed when condor_configure is called by a higher-level script, not when invoked by a person.

-overwrite
Always overwrite the contents of the `sbin` directory in the installation directory. By default, *condor_install* will not install if it finds an existing `sbin` directory with HTCondor programs in it. In this case, *condor_install* will exit with an error message. Specify **-overwrite** or **-backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an `sbin` directory out of the way that it should not move. This is particularly useful when trying to install HTCondor in a location used by other things (`/usr`, `/usr/local`, etc.) For example: *condor_install* **-prefix=/usr** will not move `/usr/sbin` out of the way unless you specify the **-backup** option.

The **-backup** behavior is used to prevent *condor_install* from overwriting running daemons - Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

-backup
Always backup the `sbin` directory in the installation directory. By default, *condor_install* will not install if it finds an existing `sbin` directory with HTCondor programs in it. In this case, *condor_install* with exit with an error message. You must specify **-overwrite** or **-backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an `sbin` directory out of the way that it should not move. This is particularly useful if you're trying to install HTCondor in a location used by other things (`/usr`, `/usr/local`, etc.) For example: *condor_install* **-prefix=/usr** will not move `/usr/sbin` out of the way unless you specify the **-backup** option.

The **-backup** behavior is used to prevent *condor_install* from overwriting running daemons - Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

-ignore-missing-libs
Ignore missing shared libraries that are detected by *condor_install*. By default, *condor_install* will detect missing shared libraries such as `libstdc++.so.5` on Linux; it will print messages and exit if missing libraries are detected. The **-ignore-missing-libs** will cause *condor_install* to not exit, and to proceed with the installation if missing libraries are detected.

-force
This is equivalent to enabling both the **-overwrite** and **-ignore-missing-libs** command line options.

-no-env-scripts
By default, *condor_configure* writes simple sh and csh shell scripts which can be sourced by their respective shells to set the user's `PATH` and `CONDOR_CONFIG` environment variables. This option prevents *condor_configure* from generating these scripts.

-env-scripts-dir=<directory>
By default, the simple *sh* and *csh* shell scripts (see **-no-env-scripts** for details) are created in the root directory of the HTCondor installation. This option causes *condor_configure* to generate these scripts in the specified directory.

-credd
Configure the the *condor_credd* daemon (credential manager daemon).

-verbose
Print information about changes to configuration variables as they occur.

15.9.4 Exit Status

condor_configure will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

15.9.5 Examples

Install HTCondor on the machine (machine1@cs.wisc.edu) to be the pool's central manager. On machine1, within the directory that contains the unzipped HTCondor distribution directories:

```
$ condor_install --type=submit,execute,manager
```

This will allow the machine to submit and execute HTCondor jobs, in addition to being the central manager of the pool.

To change the configuration such that machine2@cs.wisc.edu is an execute-only machine (that is, a dedicated computing node) within a pool with central manager on machine1@cs.wisc.edu, issue the command on that machine2@cs.wisc.edu from within the directory where HTCondor is installed:

```
$ condor_configure --central-manager=machine1@cs.wisc.edu --type=execute
```

To change the location of the `LOCAL_DIR` directory in the configuration file, do (from the directory where HTCondor is installed):

```
$ condor_configure --local-dir=/path/to/new/local/directory
```

This will move the `log`,`spool`,`execute` directories to `/path/to/new/local/directory` from the current local directory.

15.10 *condor_config_val*

Query or set a given HTCondor configuration variable

15.10.1 Synopsis

condor_config_val <*help option*>

condor_config_val [**<location options>**] <*edit option*>

condor_config_val [**<location options>**] [**<view options>**] *vars*

condor_config_val use *category* [*:template_name*] [**-expand**]

15.10.2 Description

condor_config_val can be used to quickly see what the current HTCondor configuration is on any given machine. Given a space separated set of configuration variables with the *vars* argument, *condor_config_val* will report what each of these variables is currently set to. If a given variable is not defined, *condor_config_val* will halt on that variable, and report that it is not defined. By default, *condor_config_val* looks in the local machine's configuration files in order to evaluate the variables. Variables and values may instead be queried from a daemon specified using a **location option**.

Raw output of *condor_config_val* displays the string used to define the configuration variable. This is what is on the right hand side of the equals sign (=) in a configuration file for a variable. The default output is an expanded one. Expanded output recursively replaces any macros within the raw definition of a variable with the macro's raw definition.

Each daemon remembers settings made by a successful invocation of *condor_config_val*. The configuration file is not modified.

condor_config_val can be used to persistently set or unset configuration variables for a specific daemon on a given machine using a *-set* or *-unset* **edit option**. Persistent settings remain when the daemon is restarted. Configuration variables for a specific daemon on a given machine may be set or unset for the time period that the daemon continues to run using a *-rset* or *-runset* **edit option**. These runtime settings will override persistent settings until the daemon is restarted. Any changes made will not take effect until *condor_reconfig* is invoked.

In general, modifying a host's configuration with *condor_config_val* requires the `CONFIG` access level, which is disabled on all hosts by default. Administrators have more fine-grained control over which access levels can modify which settings. See the *Security* section for more details on security settings. Further, security considerations require proper settings of configuration variables `SETTABLE_ATTRS_<PERMISSION-LEVEL>` (see *DaemonCore Configuration File Entries*), `ENABLE_PERSISTENT_CONFIG` (see *DaemonCore Configuration File Entries*) and `ALLOW...` (see *DaemonCore Configuration File Entries*) in order to use *condor_config_val* to change any configuration variable.

It is generally wise to test a new configuration on a single machine to ensure that no syntax or other errors in the configuration have been made before the reconfiguration of many machines. Having bad syntax or invalid configuration settings is a fatal error for HTCondor daemons, and they will exit. It is far better to discover such a problem on a single machine than to cause all the HTCondor daemons in the pool to exit. *condor_config_val* can help with this type of testing.

15.10.3 Options

-help
(help option) Print usage information and exit.

-version
(help option) Print the HTCondor version information and exit.

-set *"var = value"*
(edit option) Sets one or more persistent configuration file variables. The new value remains if the daemon is restarted. One or more variables can be set; the syntax requires double quote marks to identify the pairing of variable name to value, and to permit spaces.

-unset *var*
(edit option) Each of the persistent configuration variables listed reverts to its previous value.

-rset *"var = value"*
(edit option) Sets one or more configuration file variables. The new value remains as long as the daemon continues running. One or more variables can be set; the syntax requires double quote marks to identify the pairing of variable name to value, and to permit spaces.

-runset *var*
(edit option) Each of the configuration variables listed reverts to its previous value as long as the daemon continues running.

-summary[:detected]
(view option) For all configuration variables that differ from default value, print out the name and value. The values are grouped by the file that last set the variable, and in the order that they were set in that file. If the `detected` option is added, then variables such as `$(OPSYSANDVER)` that are detected at runtime are included in the ouput.

-dump
(view option) For all configuration variables that match *vars*, display the variables and their values. If no *vars* are listed, then display all configuration variables and their values. The values will be raw unless **-expand**, **-default**, or **-evaluate** are used.

-default
(view option) Default values are displayed.

-expand
(view option) Expanded values are displayed. This is the default unless -dump is used.

-raw
(view option) Raw values are displayed.

-verbose
(view option) Display configuration file name and line number where the variable is set, along with the raw, expanded, and default values of the variable.

-debug[:<opts>]
(view option) Send output to `stderr`, overriding a set value of `TOOL_DEBUG`.

-evaluate
(view option) Applied only when a **location option** specifies a daemon. The value of the requested parameter will be evaluated with respect to the ClassAd of that daemon.

-used
(view option) Applied only when a **location option** specifies a daemon. Modifies which variables are displayed to only those used by the specified daemon.

-unused
(view option) Applied only when a **location option** specifies a daemon. Modifies which variables are displayed to only those not used by the specified daemon.

-config
(view option) Applied only when the configuration is read from files (the default), and not when applied to a specific daemon. Display the current configuration file that set the variable.

-writeconfig[:upgrade] ***filename***
(view option) For the configuration read from files (the default), write to file *filename* all configuration variables. Values that are the same as internal, compile-time defaults will be preceded by the comment character. If the **:upgrade** *o* ption is specified, then values that are the same as the internal, compile-time defaults are omitted. Variables are in the same order as the they were read from the original configuration files.

-macro[:path]
(view option) Macro expand the text in *vars* as the configuration language would. You can use expansion functions such as `$F(<var>)`. If the **:path** *o* ption is specified, treat the result as a path and return the canonical form.

-mixedcase
(view option) Applied only when the configuration is read from files (the default), and not when applied to a specific daemon. Print variable names with the same letter case used in the variable's definition.

-local-name <*name*>
(view option) Applied only when the configuration is read from files (the default), and not when applied to a specific daemon. Inspect the values of attributes that use local names, which is useful to distinguish which daemon when there is more than one of the particular daemon running.

-subsystem <*daemon*>
(view option) Applied only when the configuration is read from files (the default), and not when applied to a specific daemon. Specifies the subsystem or daemon name to query, with a default value of the `TOOL` subsystem.

-address <*ip:port*>
(location option) Connect to the given IP address and port number.

-pool ***centralmanagerhostname[:portnumber]***
(location option) Use the given central manager and an optional port number to find daemons.

-name <*machine_name*>
(location option) Query the specified machine's *condor_master* daemon for its configuration. Does not function together with any of the options: **-dump**, **-config**, or **-verbose**.

-master | -schedd | -startd | -collector | -negotiator
(location option) The specific daemon to query.

use ***category*** **[*:set name*] [-expand]**
Display information about configuration templates (see *Configuration Templates*). Specifying only a *category* will list the *template_names* available for that category. Specifying a *category* and a *template_name* will display the definition of that configuration template. Adding the **-expand** option will display the expanded definition (with macro substitutions). (**-expand** has no effect if a *template_name* is not specified.) Note that there is no dash before **use** and that spaces are not allowed next to the colon character separating *category* and *template_name*.

15.10.4 Exit Status

condor_config_val will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.10.5 Examples

Here is a set of examples to show a sequence of operations using *condor_config_val*. To request the *condor_schedd* daemon on host perdita to display the value of the `MAX_JOBS_RUNNING` configuration variable:

```
$ condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

To request the *condor_schedd* daemon on host perdita to set the value of the `MAX_JOBS_RUNNING` configuration variable to the value 10.

```
$ condor_config_val -name perdita -schedd -set "MAX_JOBS_RUNNING = 10"
Successfully set configuration "MAX_JOBS_RUNNING = 10" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
$ condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects the change implemented:

```
$ condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
10
```

To set the configuration variable `MAX_JOBS_RUNNING` back to what it was before the command to set it to 10:

```
$ condor_config_val -name perdita -schedd -unset MAX_JOBS_RUNNING
Successfully unset configuration "MAX_JOBS_RUNNING" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
$ condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects that variable has gone back to is value before initial set of the variable:

```
$ condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

Getting a list of template_names for the **role** configuration template category:

```
$ condor_config_val use role
use ROLE accepts
  CentralManager
  Execute
  Personal
  Submit
```

Getting the definition of **role:personal** configuration template:

```
$ condor_config_val use role:personal
use ROLE:Personal is
    CONDOR_HOST=127.0.0.1
COLLECTOR_HOST=$(CONDOR_HOST):0
DAEMON_LIST=MASTER COLLECTOR NEGOTIATOR STARTD SCHEDD
RunBenchmarks=0
```

15.11 *condor_continue*

continue suspended jobs from the HTCondor queue

15.11.1 Synopsis

condor_continue [**-help | -version**]

condor_continue [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] **

15.11.2 Description

condor_continue continues one or more suspended jobs from the HTCondor job queue. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The job(s) to be continued are identified by one of the job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can continue the job.

15.11.3 Options

-help
: Display usage information

-version
: Display version information

-pool ***centralmanagerhostname[:portnumber]***
: Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname***
: Send the command to a machine identified by *scheddname*

-addr ***"<a.b.c.d:port>"***
: Send the command to a machine located at *"<a.b.c.d:port>"*

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

cluster
: Continue all jobs in the specified cluster

cluster.process
: Continue the specific job in the cluster

user
Continue jobs belonging to specified user

-constraint ***expression***
Continue all jobs which match the job ClassAd expression constraint

-all
Continue all the jobs in the queue

15.11.4 Exit Status

condor_continue will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.11.5 Examples

To continue all jobs except for a specific user:

```
$ condor_continue -constraint 'Owner =!= "foo"'
```

15.12 *condor_dagman*

meta scheduler of the jobs submitted as the nodes of a DAG or DAGs

15.12.1 Synopsis

condor_dagman *-f -t -l .* **-help**

condor_dagman -version

condor_dagman *-f -l .* **-csdversion** *version_string* [**-debug** *level*] [**-dryrun**] [**-maxidle** *numberOfProcs*] [**-maxjobs** *numberOfJobs*] [**-maxpre** *NumberOfPreScripts*] [**-maxpost** *NumberOfPostScripts*] [**-maxhold** *NumberOfHoldScripts*] [**-usedagdir**] **-lockfile** *filename* [**-waitfordebug**] [**-autorescue** *0|1*] [**-dorescuefrom** *number*] [**-load_save** *filename*] [**-allowversionmismatch**] [**-DumpRescue**] [**-verbose**] [**-force**] [**-notification** *value*] [**-suppress_notification**] [**-dont_suppress_notification**] [**-dagman** *DagmanExecutable*] [**-outfile_dir** *directory*] [**-update_submit**] [**-import_env**] [**-include_env** *Variables*] [**-insert_env** *Key=Value*] [**-priority** *number*] [**-DontAlwaysRunPost**] [**-AlwaysRunPost**] [**-DoRecovery**] [**-dot**] **-dag** *dag_file* [**-dag** *dag_file_2 ...* **-dag** *dag_file_n*]

15.12.2 Description

condor_dagman is a meta scheduler for the HTCondor jobs within a DAG (directed acyclic graph) (or multiple DAGs). In typical usage, a submitter of jobs that are organized into a DAG submits the DAG using *condor_submit_dag*. *condor_submit_dag* does error checking on aspects of the DAG and then submits *condor_dagman* as an HTCondor job. *condor_dagman* uses log files to coordinate the further submission of the jobs within the DAG.

All command line arguments to the *DaemonCore* library functions work for *condor_dagman*. When invoked from the command line, *condor_dagman* requires the arguments *-f -l .* to appear first on the command line, to be processed by *DaemonCore*. The **csdversion** must also be specified; at start up, *condor_dagman* checks for a version mismatch with the *condor_submit_dag* version in this argument. The *-t* argument must also be present for the **-help** option, such that output is sent to the terminal.

Arguments to *condor_dagman* are either automatically set by *condor_submit_dag* or they are specified as command-line arguments to *condor_submit_dag* and passed on to *condor_dagman*. The method by which the arguments are set is given in their description below.

condor_dagman can run multiple, independent DAGs. This is done by specifying multiple **-dag** *a* rguments. Pass multiple DAG input files as command-line arguments to *condor_submit_dag*.

Debugging output may be obtained by using the **-debug** *level* option. Level values and what they produce is described as

- level = 0; never produce output, except for usage info
- level = 1; very quiet, output severe errors
- level = 2; normal output, errors and warnings
- level = 3; output errors, as well as all warnings
- level = 4; internal debugging output
- level = 5; internal debugging output; outer loop debugging
- level = 6; internal debugging output; inner loop debugging; output DAG input file lines as they are parsed
- level = 7; internal debugging output; rarely used; output DAG input file lines as they are parsed

15.12.3 Options

-help
Display usage information and exit.

-version
Display version information and exit.

-csdversion ***VersionString***
Sets the version of *condor_submit_dag* command used to submit the DAGMan workflow. Used to help identify version mismatching.

-debug ***level***
An integer level of debugging output. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. This command-line option to *condor_submit_dag* is passed to *condor_dagman* or defaults to the value 3.

-dryrun
Inform DAGMan to do a dry run. Where the DAG is ran but node jobs are not actually submitted.

-maxidle ***NumberOfProcs***
Sets the maximum number of idle procs allowed before *condor_dagman* stops submitting more node jobs. If this option is omitted then the number of idle procs is limited by the configuration variable which defaults to 1000. To disable this limit, set *NumberOfProcs* to 0. The *NumberOfProcs* can be exceeded if a nodes job has a queue command with more than one proc to queue. i.e. `queue 500` will submit all procs even if *NumberOfProcs* is `250`. In this case DAGMan will wait for for the number of idle procs to fall below 250 before submitting more jobs to the **condor_schedd**.

-maxjobs ***NumberOfClusters***
Sets the maximum number of clusters within the DAG that will be submitted to HTCondor at one time. Each cluster is associated with one node job no matter how many individual procs are in the cluster. *NumberOfClusters* is a non-negative integer. If this option is omitted then the number of clusters is limited by the configuration variable which defaults to 0 (unlimited).

-maxpre *NumberOfPreScripts*
Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPreScripts* is a non-negative integer. If this option is omitted, the number of PRE scripts is limited by the configuration variable which defaults to 20.

-maxpost *NumberOfPostScripts*
Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPostScripts* is a non-negative integer. If this option is omitted, the number of POST scripts is limited by the configuration variable which defaults to 20.

-maxhold *NumberOfHoldScripts*
Sets the maximum number of HOLD scripts within the DAG that may be running at one time. *NumberOfHoldscripts* is a non-negative integer. If this option is omitted, the number of HOLD scripts is limited by the configuration variable , which defaults to 0 (unlimited).

-usedagdir
This optional argument causes *condor_dagman* to run each specified DAG as if the directory containing that DAG file was the current working directory. This option is most useful when running multiple DAGs in a single *condor_dagman*.

-lockfile *filename*
Names the file created and used as a lock file. The lock file prevents execution of two of the same DAG, as defined by a DAG input file. A default lock file ending with the suffix `.dag.lock` is passed to *condor_dagman* by *condor_submit_dag*.

-waitfordebug
This optional argument causes *condor_dagman* to wait at startup until someone attaches to the process with a debugger and sets the wait_for_debug variable in main_init() to false.

-autorescue *0|1*
Whether to automatically run the newest rescue DAG for the given DAG file, if one exists (0 = `false`, 1 = `true`).

-dorescuefrom *number*
Forces *condor_dagman* to run the specified rescue DAG number for the given DAG. A value of 0 is the same as not specifying this option. Specifying a nonexistent rescue DAG is a fatal error.

-load_save *filename*
Specify a file with saved DAG progress to re-run the DAG from. If given a path DAGMan will attempt to read that file following that path. Otherwise, DAGMan will check for the file in the DAG's `save_files` sub-directory.

-allowversionmismatch
This optional argument causes *condor_dagman* to allow a version mismatch between *condor_dagman* itself and the `.condor.sub` file produced by *condor_submit_dag* (or, in other words, between *condor_submit_dag* and *condor_dagman*). WARNING! This option should be used only if absolutely necessary. Allowing version mismatches can cause subtle problems when running DAGs.

-DumpRescue
This optional argument causes *condor_dagman* to immediately dump a Rescue DAG and then exit, as opposed to actually running the DAG. This feature is mainly intended for testing. The Rescue DAG file is produced whether or not there are parse errors reading the original DAG input file. The name of the file differs if there was a parse error.

-verbose
(This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Cause *condor_submit_dag* to give verbose error messages.

-force
(This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation

is used for nested DAGs.) Require *condor_submit_dag* to overwrite the files that it produces, if the files already exist. Note that `dagman.out` will be appended to, not overwritten. If new-style rescue DAG mode is in effect, and any new-style rescue DAGs exist, the **-force** flag will cause them to be renamed, and the original DAG will be run. If old-style rescue DAG mode is in effect, any existing old-style rescue DAGs will be deleted, and the original DAG will be run. See the HTCondor manual section on Rescue DAGs for more information.

-notification ***value***
This argument is only included to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs. Sets the e-mail notification for DAGMan itself. This information will be used within the HTCondor submit description file for DAGMan. This file is produced by *condor_submit_dag*. The **notification** option is described in the *condor_submit* manual page.

-suppress_notification
Causes jobs submitted by *condor_dagman* to not send email notification for events. The same effect can be achieved by setting the configuration variable to `True`. This command line option is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself. This flag is generally superfluous, as `DAGMAN_SUPPRESS_NOTIFICATION` defaults to `True`.

-dont_suppress_notification
Causes jobs submitted by *condor_dagman* to defer to content within the submit description file when deciding to send email notification for events. The same effect can be achieved by setting the configuration variable to `False`. This command line flag is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself. If both **-dont_suppress_notification** and **-suppress_notification** are specified within the same command line, the last argument is used.

-dagman ***DagmanExecutable***
(This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Allows the specification of an alternate *condor_dagman* executable to be used instead of the one found in the user's path. This must be a fully qualified path.

-outfile_dir ***directory***
(This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Specifies the directory in which the `.dagman.out` file will be written. The *directory* may be specified relative to the current working directory as *condor_submit_dag* is executed, or specified with an absolute path. Without this option, the `.dagman.out` file is placed in the same directory as the first DAG input file listed on the command line.

-update_submit
(This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) This optional argument causes an existing `.condor.sub` file to not be treated as an error; rather, the `.condor.sub` file will be overwritten, but the existing values of **-maxjobs**, **-maxidle**, **-maxpre**, and **-maxpost** will be preserved.

-import_env
(This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) This optional argument causes *condor_submit_dag* to import the current environment into the **environment** command of the `.condor.sub` file it generates.

-include_env ***Variables***
This optional argument takes a comma separated list of enviroment variables to add to `.condor.sub` `getenv` environment filter which causes found matching environment variables to be added to the DAGMan manager jobs **environment**.

-insert_env ***Key=Value***
This optional argument takes a delimited string of *Key=Value* pairs to explicitly set into the `.condor.`

`sub` files `environment` macro. The base delimiter is a semicolon that can be overriden by setting the first character in the string to a valid delimiting character. If multiple **-insert_env** flags contain the same *Key* then the last occurances *Value* will be set in the DAGMan jobs **environment**.

-priority *number*
Sets the minimum job priority of node jobs submitted and running under this *condor_dagman* job.

-DontAlwaysRunPost
This option causes *condor_dagman* to not run the POST script of a node if the PRE script fails.

-AlwaysRunPost
This option causes *condor_dagman* to always run the POST script of a node, even if the PRE script fails.

-DoRecovery
Causes *condor_dagman* to start in recovery mode. This means that it reads the relevant job user log(s) and catches up to the given DAG's previous state before submitting any new jobs.

-dot
Run *condor_dagman* up until the point when a **DOT** file is produced.

-dag *filename*
filename is the name of the DAG input file that is set as an argument to *condor_submit_dag*, and passed to *condor_dagman*.

15.12.4 Exit Status

condor_dagman will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.12.5 Examples

condor_dagman is normally not run directly, but submitted as an HTCondor job by running condor_submit_dag. See the *condor_submit_dag* manual page for examples.

15.13 *condor_drain*

Control draining of an execute machine

15.13.1 Synopsis

condor_drain [**-help**]

condor_drain [**-debug**] [**-pool** *pool-name*] [**-graceful | -quick | -fast**] [**-reason** *reason-text*] [**-resume-on-completion | -restart-on-completion | -reconfig-on-completion | -exit-on-completion**] [**-check** *expr*] [**-start** *expr*] *machine-name*

condor_drain [**-debug**] [**-pool** *pool-name*] **-cancel** [**-request-id** *id*] *machine-name*

15.13.2 Description

condor_drain is an administrative command used to control the draining of all slots on an execute machine. When a machine is draining, it will not accept any new jobs unless the **-start** expression specifies otherwise. Which machine to drain is specified by the argument *machine-name*, and will be the same as the machine ClassAd attribute `Machine`.

How currently running jobs are treated depends on the draining schedule that is chosen with a command-line option:

-graceful
: Initiate a graceful eviction of the job. This means all promises that have been made to the job are honored, including `MaxJobRetirementTime`. The eviction of jobs is coordinated to reduce idle time. This means that if one slot has a job with a long retirement time and the other slots have jobs with shorter retirement times, the effective retirement time for all of the jobs is the longer one. If no draining schedule is specified, **-graceful** is chosen by default.

-quick
: `MaxJobRetirementTime` is not honored. Eviction of jobs is immediately initiated. Jobs are given time to shut down according to the usual policy, that is, given by `MachineMaxVacateTime`.

-fast
: Jobs are immediately hard-killed, with no chance to gracefully shut down.

If you specify **-graceful**, you may also specify **-start**. On a gracefully-draining machine, some jobs may finish retiring before others. By default, the resources used by the newly-retired jobs do not become available for use by other jobs until the machine exits the draining state (see below). The **-start** expression you supply replaces the draining machine's normal `START` expression for the duration of the draining state, potentially making those resources available. See the *condor_startd Policy Configuration* section for more information.

Once draining is complete, the machine will enter the Drained/Idle state. To resume normal operation (negotiation) at that time or any previous time during draining, the **-cancel** option may be used. The **-resume-on-completion** option results in automatic resumption of normal operation once draining has completed, and may be used when initiating draining. This is useful for forcing a machine with a partitionable slots to join all of the resources back together into one machine, facilitating de-fragmentation and whole machine negotiation.

15.13.3 Options

-help
: Display brief usage information and exit.

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-pool ***pool-name***
: Specify an alternate HTCondor pool, if the default one is not desired.

-graceful
: (the default) Honor the maximum vacate and retirement time policy.

-quick
: Honor the maximum vacate time, but not the retirement time policy.

-fast
: Honor neither the maximum vacate time policy nor the retirement time policy.

-reason ***reason-text***
: Set the drain reason to *reason-text*. While the *condor_startd* is draining it will advertise the given reason. If this option is not used the reason defaults to the name of the user that started the drain.

-resume-on-completion
When done draining, resume normal operation, such that potentially the whole machine could be claimed.

-restart-on-completion
When done draining, restart the *condor_startd* daemon so that configuration changes will take effect.

-reconfig-on-completion
When done draining, reconfig and then resume normal operation. A reconfig will not change the resources assigned to slots, but most other configuration changes will be applied, including changes to the expression and to offline GPUs and universes.

-exit-on-completion
When done draining, shut down the *condor_startd* daemon and tell the *condor_master* not to restart it automatically.

-check *expr*
Abort draining, if `expr` is not true for all slots to be drained.

-start *expr*
The `START` expression to use while the machine is draining. You can't reference the machine's existing `START` expression.

-cancel
Cancel a prior draining request, to permit the *condor_negotiator* to use the machine again.

-request-id *id*
Specify a specific draining request to cancel, where *id* is given by the `DrainingRequestId` machine ClassAd attribute.

15.13.4 Exit Status

condor_drain will exit with a non-zero status value if it fails and zero status if it succeeds.

15.14 *condor_evicted_files*

Inspect the file(s) that HTCondor is holding on to as a result of a job being evicted when `when_to_transfer_output = ON_EXIT_OR_EVICT`, or checkpointing when `CheckpointExitCode` is set.

15.14.1 Synopsis

condor_evicted_files [COMMAND] <clusterID>.<procID>[<clusterID.<procID>]*

15.14.2 Description

Print the directory or directories HTCondor is using to store files for the specified job or jobs. COMMAND may be one of *dir*, *list*, or *get*:

- *dir*: Print the directory (for each job) in which the file(s) are stored.
- *list*: List the contents of the directory (for each job).
- *get*: Copy the contents of the directory to a subdirectory named after each job's ID.

15.14.3 General Remarks

The tool presently has a number of limitations:

- It must be run the same machine as the job's schedd.
- The schedd must NOT have ALTERNATE_JOB_SPOOL set
- You can't name the destination directory for the *get* command.
- The tool can't distinguish between an invalid job ID and a job for which HTCondor never held any files.

15.14.4 Exit Status

Returns 0 on success.

15.14.5 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.15 *condor_fetchlog*

Retrieve a daemon's log file that is located on another computer

15.15.1 Synopsis

condor_fetchlog [**-help | -version**]

condor_fetchlog [**-pool** *centralmanagerhostname[:portnumber]*] [**-master | -startd | -schedd | -collector | -negotiator | -kbdd**] *machine-name subsystem[.extension]*

15.15.2 Description

condor_fetchlog contacts HTCondor running on the machine specified by *machine-name*, and asks it to return a log file from that machine. Which log file is determined from the *subsystem[.extension]* argument. The log file is printed to standard output. This command eliminates the need to remotely log in to a machine in order to retrieve a daemon's log file.

For security purposes of authentication and authorization, this command requires `ADMINISTRATOR` level of access.

The *subsystem[.extension]* argument is utilized to construct the log file's name. Without an optional *.extension*, the value of the configuration variable named *subsystem* _LOG defines the log file's name. When specified, the *.extension* is appended to this value.

The *subsystem* argument is any value `$(SUBSYSTEM)` that has a defined configuration variable of `$(SUBSYSTEM)_LOG`, or any of

- `NEGOTIATOR_MATCH`
- `HISTORY`
- `STARTD_HISTORY`

A value for the optional *.extension* to the *subsystem* argument is typically one of the three strings:

1. `.old`
2. `.slot<X>`
3. `.slot<X>.old`

Within these strings, <X> is substituted with the slot number.

A *subsystem* argument of `STARTD_HISTORY` fetches all *condor_startd* history by concatenating all instances of log files resulting from rotation.

15.15.3 Options

-help
Display usage information

-version
Display version information

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number

-master
Send the command to the *condor_master* daemon (default)

-startd
Send the command to the *condor_startd* daemon

-schedd
Send the command to the *condor_schedd* daemon

-collector
Send the command to the *condor_collector* daemon

-kbdd
Send the command to the *condor_kbdd* daemon

15.15.4 Examples

To get the *condor_negotiator* daemon's log from a host named `head.example.com` from within the current pool:

```
$ condor_fetchlog head.example.com NEGOTIATOR
```

To get the *condor_startd* daemon's log from a host named `execute.example.com` from within the current pool:

```
$ condor_fetchlog execute.example.com STARTD
```

This command requested the *condor_startd* daemon's log from the *condor_master*. If the *condor_master* has crashed or is unresponsive, ask another daemon running on that computer to return the log. For example, ask the *condor_startd* daemon to return the *condor_master* 's log:

```
$ condor_fetchlog -startd execute.example.com MASTER
```

15.15.5 Exit Status

condor_fetchlog will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.16 *condor_findhost*

find machine(s) in the pool that can be used with minimal impact on currently running HTCondor jobs and best meet any specified constraints

15.16.1 Synopsis

condor_findhost [**-help**] [**-m**] [**-n** *num*] [**-c** *c_expr*] [**-r** *r_expr*] [**-p** *centralmanagerhostname*]

15.16.2 Description

condor_findhost searches an HTCondor pool of machines for the best machine or machines that will have the minimum impact on running HTCondor jobs if the machine or machines are taken out of the pool. The search may be limited to the machine or machines that match a set of constraints and rank expression.

condor_findhost returns a fully-qualified domain name for each machine. The search is limited (constrained) to a specific set of machines using the *-c* option. The search can use the *-r* option for rank, the criterion used for selecting a machine or machines from the constrained list.

15.16.3 Options

-help
: Display usage information and exit

-m
: Only search for entire machines. Slots within an entire machine are not considered.

-n *num*
: Find and list up to *num* machines that fulfill the specification. *num* is an integer greater than zero.

-c *c_expr*
: Constrain the search to only consider machines that result from the evaluation of *c_expr*. *c_expr* is a ClassAd expression.

-r *r_expr*
: *r_expr* is the rank expression evaluated to use as a basis for machine selection. *r_expr* is a ClassAd expression.

-p *centralmanagerhostname*
: Specify the pool to be searched by giving the central manager's host name. Without this option, the current pool is searched.

15.16.4 General Remarks

condor_findhost is used to locate a machine within a pool that can be taken out of the pool with the least disturbance of the pool.

An administrator should set preemption requirements for the HTCondor pool. The expression

```
(Interactive =?= TRUE )
```

will let *condor_findhost* know that it can claim a machine even if HTCondor would not normally preempt a job running on that machine.

15.16.5 Exit Status

The exit status of *condor_findhost* is zero on success. If not able to identify as many machines as requested, it returns one more than the number of machines identified. For example, if 8 machines are requested, and *condor_findhost* only locates 6, the exit status will be 7. If not able to locate any machines, or an error is encountered, *condor_findhost* will return the value 1.

15.16.6 Examples

To find and list four machines, preferring those with the highest mips (on Drystone benchmark) rating:

```
$ condor_findhost -n 4 -r "mips"
```

To find and list 24 machines, considering only those where the `kflops` attribute is not defined:

```
$ condor_findhost -n 24 -c "kflops=?=undefined"
```

15.17 *condor_gather_info*

Gather information about an HTCondor installation and a queued job

15.17.1 Synopsis

condor_gather_info [–**jobid** *ClusterId.ProcId*] [–**scratch** */path/to/directory*]

15.17.2 Description

condor_gather_info is a Linux-only tool that will collect and output information about the machine it is run upon, about the HTCondor installation local to the machine, and optionally about a specified HTCondor job. The information gathered by this tool is most often used as a debugging aid for the developers of HTCondor.

Without the –**jobid** option, information about the local machine and its HTCondor installation is gathered and placed into the file called `condor-profile.txt`, in the current working directory. The information gathered is under the category of Identity.

With the –**jobid** option, additional information is gathered about the job given in the command line argument and identified by its `ClusterId` and `ProcId` ClassAd attributes. The information includes both categories, Identity and Job information. As the quantity of information can be extensive, this information is placed into a compressed tar file. The file is placed into the current working directory, and it is named using the format

```
cgi-<username>-jid<ClusterId>.<ProcId>-<year>-<month>-<day>-<hour>_<minute>_<second>-<TZ>
↪.tar.gz
```

All values within <> are substituted with current values. The building of this potentially large tar file can require a fair amount of temporary space. If the –**scratch** option is specified, it identifies a directory in which to build the tar file. If the –**scratch** option is not specified, then the directory will be `/tmp/cgi-<PID>`, where the process ID is that of the *condor_gather_info* executable.

The information gathered by this tool:

1. Identity
    - User name who generated the report
    - Script location and machine name
    - Date of report creation
    - `uname -a`
    - Contents of `/etc/issue`
    - Contents of `/etc/redhat-release`
    - Contents of `/etc/debian_version`
    - Contents of `$(LOG)/MasterLog`
    - Contents of `$(LOG)/ShadowLog`
    - Contents of `$(LOG)/SchedLog`
    - Output of `ps -auxww -forest`
    - Output of `df -h`
    - Output of `iptables -L`
    - Output of `ls 'condor_config_val LOG'`
    - Output of `ldd 'condor_config_val SBIN'/condor_schedd`
    - Contents of `/etc/hosts`
    - Contents of `/etc/nsswitch.conf`

- Output of `ulimit -a`
- Output of `uptime`
- Output of `free`
- Network interface configuration (`ifconfig`)
- HTCondor version
- Location of HTCondor configuration files
- HTCondor configuration variables
  - All variables and values
  - Definition locations for each configuration variable

2. Job Information
   - Output of `condor_q jobid`
   - Output of `condor_q -l jobid`
   - Output of `condor_q -analyze jobid`
   - Job event log, if it exists
     - Only events pertaining to the job ID
   - If *condor_gather_info* has the proper permissions, it runs *condor_fetchlog* on the machine where the job most recently ran, and includes the contents of the logs from the *condor_master*, *condor_startd*, and *condor_starter*.

15.17.3 Options

-jobid <*ClusterId.ProcId*>
Data mine information about this HTCondor job from the local HTCondor installation and *condor_schedd*.

-scratch */path/to/directory*
A path to temporary space needed when building the output tar file. Defaults to `/tmp/cgi-<PID>`, where `<PID>` is replaced by the process ID of *condor_gather_info*.

15.17.4 Files

- `condor-profile.txt` The Identity portion of the information gathered when *condor_gather_info* is run without arguments.
- `cgi-<username>-jid<cluster>.<proc>-<year>-<month>-<day>-<hour>_<minute>_<second>-<TZ>.tar.gz` The output file which contains all of the information produced by this tool.

15.17.5 Exit Status

condor_gather_info will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.18 *condor_gpu_discovery*

Output GPU-related ClassAd attributes

15.18.1 Synopsis

condor_gpu_discovery -help

condor_gpu_discovery [**<options>**]

15.18.2 Description

condor_gpu_discovery outputs ClassAd attributes corresponding to a host's GPU capabilities. It can presently report CUDA and OpenCL devices; which type(s) of device(s) it reports is determined by which libraries, if any, it can find when it runs; this reflects what GPU jobs will find on that host when they run. (Note that some HTCondor configuration settings may cause the environment to differ between jobs and the HTCondor daemons in ways that change library discovery.)

If `CUDA_VISIBLE_DEVICES` or `GPU_DEVICE_ORDINAL` is set in the environment when *condor_gpu_discovery* is run, it will report only devices present in the those lists.

This tool is not available for MAC OS platforms.

With no command line options, the single ClassAd attribute `DetectedGPUs` is printed. If the value is 0, no GPUs were detected. If one or more GPUS were detected, the value is a string, presented as a comma and space separated list of the GPUs discovered, where each is given a name further used as the *prefix string* in other attribute names. Where there is more than one GPU of a particular type, the *prefix string* includes an GPU id value identifying the device; these can be integer values that monotonically increase from 0 when the `-by-index` option is used or globally unique identifiers when the `-short-uuid` or `-uuid` argument is used.

For example, a discovery of two GPUs with `-by-index` may output

```
DetectedGPUs="CUDA0, CUDA1"
```

Further command line options use `"CUDA"` either with or without one of the integer values 0 or 1 as the name of the device properties ad for `-nested` properties, or as the *prefix string* in attribute names when `-not-nested` properties are chosen.

For machines with more than one or two NVIDIA devices, it is recommended that you also use the `-short-uuid` or `-uuid` option. The uuid value assigned by NVIDA to each GPU is unique, so using this option provides stable device identifiers for your devices. The `-short-uuid` option uses only part of the uuid, but it is highly likely to still be unique for devices on a single machine. As of HTCondor 9.0 `-short-uuid` is the default. When `-short-uuid` is used, discovery of two GPUs may look like this

```
DetectedGPUs="GPU-ddc1c098, GPU-9dc7c6d6"
```

Any NVIDIA runtime library later than 9.0 will accept the above identifiers in the `CUDA_VISIBLE_DEVICES` environment variable.

If the NVML libary is available, and a multi-instance GPU (MIG) -capable device is present, has MIG enabled, and has created compute instances for each MIG instance, *condor_gpu_discovery* will report those instance as distinct devices. Their names will be in the long UUID form unless the `-short-uuid` option is used, because they can not be enumerated via CUDA. MIG instances don't have some of the properties reported by the `-properties`, `-extra`, and `-dynamic` options; these properties will be omitted. If MIG is enabled on any GPU in the system, some properties become unavailable for every GPU in the system; *condor_gpu_discovery* will report what it can.

15.18.3 Options

-help
Print usage information and exit.

-properties
In addition to the `DetectedGPUs` attribute, display some of the attributes of the GPUs. Each of these attributes will be in a nested ClassAd (`-nested`) or have a *prefix string* at the beginning of its name (`-not-nested`). The displayed CUDA attributes are `Capability`, `DeviceName`, `DriverVersion`, `ECCEnabled`, `GlobalMemoryMb`, and `RuntimeVersion`. The displayed Open CL attributes are `DeviceName`, `ECCEnabled`, `OpenCLVersion`, and `GlobalMemoryMb`.

-nested

Default. Display properties that are common to all GPUs in a `Common` nested ClassAd,
and properties that are not common to all in a nested ClassAd using the GPUid as the ClassAd name. Use the `-not-nested` argument to disable nested ClassAds and return to the older behavior of using a *prefix string* for individual property attributes.

-not-nested

Display properties that are common to all GPUs using a `CUDA` or `OCL` as
the attribute prefix, and properties that are not common to all using a GPUid prefix. Versions of *condor_gpu_discovery* prior to 9.11.0 support only this mode.

-extra
Display more attributes of the GPUs. Each of these attributes will be added to a nested property ClassAd (`-nested`) or have a *prefix string* at the beginning of its name (`-not-nested`). The additional CUDA attributes are `ClockMhz`, `ComputeUnits`, and `CoresPerCU`. The additional Open CL attributes are `ClockMhz` and `ComputeUnits`.

-dynamic
Display attributes of NVIDIA devices that change values as the GPU is working. Each of these attributes will be added to the the nested property ClassAd (`-nested`) or have a *prefix string* at the beginning of its name (`-not-nested`). These are `FanSpeedPct`, `BoardTempC`, `DieTempC`, `EccErrorsSingleBit`, and `EccErrorsDoubleBit`.

-mixed
When displaying attribute values, assume that the machine has a heterogeneous set of GPUs, so always include the integer value in the *prefix string*.

-device <*N*>
Display properties only for GPU device <*N*>, where <*N*> is the integer value defined for the *prefix string*. This option may be specified more than once; additional <*N*> are listed along with the first. This option adds to the devices(s) specified by the environment variables `CUDA_VISIBLE_DEVICES` and `GPU_DEVICE_ORDINAL`, if any.

-tag *string*
Set the resource tag portion of the intended machine ClassAd attribute `Detected<ResourceTag>` to be *string*. If this option is not specified, the resource tag is `"GPUs"`, resulting in attribute name `DetectedGPUs`.

-prefix *str*
When naming `-not-nested` attributes, use *str* as the *prefix string*. When this option is not specified, the *prefix string* is either `CUDA` or `OCL` unless `-uuid` or `-short-uuid` is also used.

-by-index
Use the prefix and device index as the device identifier.

-short-uuid
Use the first 8 characters of the NVIDIA uuid as the device identifier. When this option is used, devices will be shown as `GPU-<xxxxxxxx>` where `<xxxxxxxx>` is the first 8 hex digits of the NVIDIA device uuid. Unlike device indices, the uuid of a device will not change of other devices are taken offline or drained.

-uuid
Use the full NVIDIA uuid as the device identifier rather than the device index.

-simulate:[D,N[,D2,...]]
For testing purposes, assume that N devices of type D were detected, And N2 devices of type D2, etc. No discovery software is invoked. D can be a value from 0 to 6 which selects a simulated a GPU from the following table.

Table 1: Simulated GPUs

	DeviceName	Capability	GlobalMemoryMB
0	GeForce GT 330	1.2	1024
1	GeForce GTX 480	2.0	1536
2	Tesla V100-PCIE-16GB	7.0	24220
3	TITAN RTX	7.5	24220
4	A100-SXM4-40GB	8.0	40536
5	NVIDIA A100-SXM4-40GB MIG 3g.20gb	8.0	20096
6	NVIDIA A100-SXM4-40GB MIG 1g.5gb	8.0	4864

-opencl
Prefer detection via OpenCL rather than CUDA. Without this option, CUDA detection software is invoked first, and no further Open CL software is invoked if CUDA devices are detected.

-cuda
Do only CUDA detection.

-nvcuda
For Windows platforms only, use a CUDA driver rather than the CUDA run time.

-config
Output in the syntax of HTCondor configuration, instead of ClassAd language. An additional attribute is produced `NUM_DETECTED_GPUs` which is set to the number of GPUs detected.

-repeat [*N*]
Repeat listed GPUs *N* (default 2) times. This results in a list that looks like `CUDA0, CUDA1, CUDA0, CUDA1`.

If used with **-divide**, the last one on the command-line wins, but you must specify `2` if you want it; the default value only applies to the first flag.

-divide [*N*]
Like **-repeat**, except also divide the attribute `GlobalMemoryMb` by *N*. This may help you avoid overcommitting your GPU's memory.

If used with **-repeat**, the last one on the command-line wins, but you must specify `2` if you want it; the default value only applies to the first flag.

-packed
When repeating GPUs, repeat each GPU *N* times, not the whole list. This results in a list that looks like `CUDA0, CUDA0, CUDA1, CUDA1`.

-cron
This option suppresses the `DetectedGpus` attribute so that the output is suitable for use with *condor_startd* cron. Combine this option with the **-dynamic** option to periodically refresh the dynamic Gpu information such as temperature. For example, to refresh GPU temperatures every 5 minutes

```
use FEATURE : StartdCronPeriodic(DYNGPUS, 5*60, $(LIBEXEC)/condor_gpu_
↪discovery, -dynamic -cron)
```

-verbose
For interactive use of the tool, output extra information to show detection while in progress.

-diagnostic
Show diagnostic information, to aid in tool development.

15.18.4 Exit Status

condor_gpu_discovery will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.19 *condor_history*

View log of HTCondor jobs completed to date

15.19.1 Synopsis

condor_history [**-help**]

condor_history [**-name** *name*] [**-pool** *centralmanagerhostname[:portnumber]*] [**-backwards**] [**-forwards**] [**-constraint** *expr*] [**-file** *filename*] [**-userlog** *filename*] [**-search** *path*] [**-dir | -directory**] [**-local**] [**-startd**] [**-epochs**] [**-format** *formatString AttributeName*] [**-autoformat[:jlhVr,tng]** *attr1 [attr2 ...]*] [**-l | -long | -xml | -json | -jsonl**] [**-match | -limit** *number*] [**-attributes** *attr1[,attr2...]*] [**-print-format** *file*] [**-wide**] [**-since** *time_or_jobid*] [**-completedsince** *time_expr*] [**-scanlimit** *number*] [**cluster | cluster.process | owner**]

15.19.2 Description

condor_history displays a summary of all HTCondor jobs listed in the specified history files. If no history files are specified with the **-file** option, the local history file as specified in HTCondor's configuration file (`$(SPOOL)`/history by default) is read. The default listing summarizes in reverse chronological order each job on a single line, and contains the following items:

ID
The cluster/process id of the job.

OWNER
The owner of the job.

SUBMITTED
The month, day, hour, and minute the job was submitted to the queue.

RUN_TIME
Remote wall clock time accumulated by the job to date in days, hours, minutes, and seconds, given as the job ClassAd attribute `RemoteWallClockTime`.

ST
Completion status of the job (C = completed and X = removed).

COMPLETED
The time the job was completed.

CMD
The name of the executable.

If a job ID (in the form of *cluster_id* or *cluster_id.proc_id*) or an *owner* is provided, output will be restricted to jobs with the specified IDs and/or submitted by the specified owner. The *-constraint* option can be used to display jobs that satisfy a specified boolean expression.

15.19.3 Options

-help
Display usage information and exit.

-name *name*
Query the named *condor_schedd* daemon. If used with **-startd**, query the named *condor_startd* daemon

-pool *centralmanagerhostname[:portnumber]*
Use the *centralmanagerhostname* as the central manager to locate *condor_schedd* daemons. The default is the `COLLECTOR_HOST`, as specified in the configuration.

-backwards
List jobs in reverse chronological order. The job most recently added to the history file is first. This is the default ordering.

-forwards
List jobs in chronological order. The job most recently added to the history file is last. At least 4 characters must be given to distinguish this option from the **-file** and **-format** options.

-constraint *expr*
Display jobs that satisfy the expression.

-since *jobid or expr*
Stop scanning when the given jobid is found or when the expression becomes true.

-completedsince *time_expr*
Scan until the first job that completed on or before the given unix timestamp. The argument can be any expression that evaluates to a unix timestamp. This option is equivalent to **-since** *'CompletionDate<=time_expr'*.

-scanlimit *Number*
Stop scanning when the given number of ads have been read.

-limit *Number*
Limit the number of jobs displayed to *Number*. Same option as **-match**.

-match *Number*
Limit the number of jobs displayed to *Number*. Same option as **-limit**.

-local
Read from local history files even if there is a SCHEDD_HOST configured.

-startd
Read from Startd history files rather than Schedd history files. If used with the *-name* option, query is sent as a command to the given Startd which must be version 9.0 or later.

-epochs[:d]
Read per job run instance recording also known as job epochs instead of default history file. The **-epochs** option may be followed by a colon character for extra functionality:

d Delete job epoch files after finished reading. This option only deletes epoch files store within , and can not be used with **-match**, **-limit**, or **-scanlimit**.

-file ***filename***
Use the specified file instead of the default history file.

-userlog ***filename***
Display jobs, with job information coming from a job event log, instead of from the default history file. A job event log does not contain all of the job information, so some fields in the normal output of *condor_history* will be blank.

-search ***path***
Use the specified path to filename and all matching condor time rotated files `filename.YYYYMMDDTHHMMSS` instead of the default history file. If used with **-dir** option then *condor_history* will use the provided path as the directory to search for specific pattern matching history files.

-dir or -directory
Search for files in a sources alternate directory configuration knob to read from instead of default history file. Note: only applies to **-epochs**.

-format ***formatString*** **AttributeName**
Display jobs with a custom format. See the *condor_q* man page **-format** option for details.

-autoformat[:jlhVr,tng] ***attr1 [attr2 ...]*** **or -af[:jlhVr,tng]** ***attr1 [attr2 ...]***
(output option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

j print the job ID as the first field,

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may not be used together. The **l** and **h** characters may not be used together.

-print-format ***file***
Read output formatting information from the given custom print format file. see *Print Formats* for more information about custom print format files.

-l or -long
Display job ClassAds in long format.

-attributes ***attrs***
Display only the given attributes when the **-long** *o* ption is used.

-xml
Display job ClassAds in XML format. The XML format is fully defined in the reference manual, obtained from the ClassAds web page, with a link at http://htcondor.org/classad/classad.html.

-json
Display job ClassAds in JSON format.

-jsonl
Display job ClassAds in JSON-Lines format: one job ad per line.

-wide[:number]
Restrict output to the given column width. Default width is 80 columns, if **-wide** is used without the optional *number* argument, the width of the output is not restricted.

15.19.4 Exit Status

condor_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.20 *condor_hold*

put jobs in the queue into the hold state

15.20.1 Synopsis

condor_hold [**-help | -version**]

condor_hold [**-debug**] [**-reason** *reasonstring*] [**-subcode** *number*] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] *cluster...* | *cluster.process...* | *user...* | **-constraint** *expression ...*

condor_hold [**-debug**] [**-reason** *reasonstring*] [**-subcode** *number*] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] **-all**

15.20.2 Description

condor_hold places jobs from the HTCondor job queue in the hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be held are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can place the job on hold.

A job in the hold state remains in the job queue, but the job will not run until released with *condor_release*.

A currently running job that is placed in the hold state by *condor_hold* is sent a hard kill signal.

15.20.3 Options

-help
Display usage information

-version
Display version information

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname***
Send the command to a machine identified by *scheddname*

-addr ***"<a.b.c.d:port>"***
Send the command to a machine located at *"<a.b.c.d:port>"*

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-reason ***reasonstring***
Sets the job ClassAd attribute `HoldReason` to the value given by *reasonstring*. *reasonstring* will be delimited by double quote marks on the command line, if it contains space characters.

-subcode ***number***
Sets the job ClassAd attribute `HoldReasonSubCode` to the integer value given by *number*.

cluster
Hold all jobs in the specified cluster

cluster.process
Hold the specific job in the cluster

user
Hold all jobs belonging to specified user

-constraint ***expression***
Hold all jobs which match the job ClassAd expression constraint (within quotation marks). Note that quotation marks must be escaped with the backslash characters for most shells.

-all
Hold all the jobs in the queue

15.20.4 See Also

condor_release

15.20.5 Examples

To place on hold all jobs (of the user that issued the *condor_hold* command) that are not currently running:

```
$ condor_hold -constraint "JobStatus!=2"
```

Multiple options within the same command cause the union of all jobs that meet either (or both) of the options to be placed in the hold state. Therefore, the command

```
$ condor_hold Mary -constraint "JobStatus!=2"
```

places all of Mary's queued jobs into the hold state, and the constraint holds all queued jobs not currently running. It also sends a hard kill signal to any of Mary's jobs that are currently running. Note that the jobs specified by the constraint will also be Mary's jobs, if it is Mary that issues this example *condor_hold* command.

15.20.6 Exit Status

condor_hold will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.21 *condor_install*

Configure or install HTCondor

15.21.1 Synopsis

condor_configure or **condor_install** [–**help**] [–**usage**]

condor_configure or **condor_install** [--**install[=<path/to/release>]**] [--**install-dir=<path>**] [--**prefix=<path>**] [--**local-dir=<path>**] [--**make-personal-condor**] [--**bosco**] [--**type = < submit, execute, manager >**] [--**central-manager = < hostname>**] [--**owner = < ownername >**] [--**maybe-daemon-owner**] [--**install-log = < file >**] [--**overwrite**] [--**ignore-missing-libs**] [--**force**] [--**no-env-scripts**] [--**env-scripts-dir = < directory >**] [--**backup**] [--**credd**] [--**verbose**]

15.21.2 Description

condor_configure and *condor_install* refer to a single script that installs and/or configures HTCondor on Unix machines. As the names imply, *condor_install* is intended to perform a HTCondor installation, and *condor_configure* is intended to configure (or reconfigure) an existing installation. Both will run with Perl 5.6.0 or more recent versions.

condor_configure (and *condor_install*) are designed to be run more than one time where required. It can install HTCondor when invoked with a correct configuration via

```
$ condor_install
```

or

```
$ condor_configure --install
```

or, it can change the configuration files when invoked via

```
$ condor_configure
```

Note that changes in the configuration files do not result in changes while HTCondor is running. To effect changes while HTCondor is running, it is necessary to further use the *condor_reconfig* or *condor_restart* command. *condor_reconfig* is required where the currently executing daemons need to be informed of configuration changes. *condor_restart* is required where the options –**make-personal-condor** or –**type** are used, since these affect which daemons are running.

Running *condor_configure* or *condor_install* with no options results in a usage screen being printed. The –**help** option can be used to display a full help screen.

Within the options given below, the phrase release directories is the list of directories that are released with HTCondor. This list includes: `bin`, `etc`, `examples`, `include`, `lib`, `libexec`, `man`, `sbin`, `sql` and `src`.

15.21.3 Options

-help
Print help screen and exit

-usage
Print short usage and exit

-install
Perform installation, assuming that the current working directory contains the release directories. Without further options, the configuration is that of a Personal HTCondor, a complete one-machine pool. If used as an upgrade within an existing installation directory, existing configuration files and local directory are preserved. This is the default behavior of *condor_install*.

-install-dir=<path>
Specifies the path where HTCondor should be installed or the path where it already is installed. The default is the current working directory.

-prefix=<path>
This is an alias for **-install-dir**.

-local-dir=<path>
Specifies the location of the local directory, which is the directory that generally contains the local (machine-specific) configuration file as well as the directories where HTCondor daemons write their run-time information (`spool`, `log`, `execute`). This location is indicated by the `LOCAL_DIR` variable in the configuration file. When installing (that is, if **-install** is specified), *condor_configure* will properly create the local directory in the location specified. If none is specified, the default value is given by the evaluation of `$(RELEASE_DIR)/local.$(HOSTNAME)`.

During subsequent invocations of *condor_configure* (that is, without the -install option), if the -local-dir option is specified, the new directory will be created and the `log`, `spool` and `execute` directories will be moved there from their current location.

-make-personal-condor
Installs and configures for Personal HTCondor, a fully-functional, one-machine pool.

-bosco
Installs and configures Bosco, a personal HTCondor that submits jobs to remote batch systems.

-type= < submit, execute, manager >
One or more of the types may be listed. This determines the roles that a machine may play in a pool.

In general, any machine can be a submit and/or execute machine, and there is one central manager per pool. In the case of a Personal HTCondor, the machine fulfills all three of these roles.

-central-manager=<hostname>
Instructs the current HTCondor installation to use the specified machine as the central manager. This modifies the configuration variable `COLLECTOR_HOST` to point to the given host name. The central manager machine's HTCondor configuration needs to be independently configured to act as a manager using the option **-type=manager**.

-owner=<ownername>
Set configuration such that HTCondor daemons will be executed as the given owner. This modifies the ownership on the `log`, `spool` and `execute` directories and sets the `CONDOR_IDS` value in the configuration file, to ensure that HTCondor daemons start up as the specified effective user. This is only applicable when *condor_configure* is run by root. If not run as root, the owner is the user running the *condor_configure* command.

-maybe-daemon-owner
If **-owner** is not specified and no appropriate user can be found to run Condor, then this option will allow the daemon user to be selected. This option is rarely needed by users but can be useful for scripts that invoke condor_configure to install Condor.

-install-log=<file>
Save information about the installation in the specified file. This is normally only needed when condor_configure is called by a higher-level script, not when invoked by a person.

-overwrite
Always overwrite the contents of the `sbin` directory in the installation directory. By default, *condor_install* will not install if it finds an existing `sbin` directory with HTCondor programs in it. In this case, *condor_install* will exit with an error message. Specify **-overwrite** or **-backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an `sbin` directory out of the way that it should not move. This is particularly useful when trying to install HTCondor in a location used by other things (`/usr`, `/usr/local`, etc.) For example: *condor_install* **-prefix=/usr** will not move `/usr/sbin` out of the way unless you specify the **-backup** option.

The **-backup** behavior is used to prevent *condor_install* from overwriting running daemons - Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

-backup
Always backup the `sbin` directory in the installation directory. By default, *condor_install* will not install if it finds an existing `sbin` directory with HTCondor programs in it. In this case, *condor_install* with exit with an error message. You must specify **-overwrite** or **-backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an `sbin` directory out of the way that it should not move. This is particularly useful if you're trying to install HTCondor in a location used by other things (`/usr`, `/usr/local`, etc.) For example: *condor_install* **-prefix=/usr** will not move `/usr/sbin` out of the way unless you specify the **-backup** option.

The **-backup** behavior is used to prevent *condor_install* from overwriting running daemons - Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

-ignore-missing-libs
Ignore missing shared libraries that are detected by *condor_install*. By default, *condor_install* will detect missing shared libraries such as `libstdc++.so.5` on Linux; it will print messages and exit if missing libraries are detected. The **-ignore-missing-libs** will cause *condor_install* to not exit, and to proceed with the installation if missing libraries are detected.

-force
This is equivalent to enabling both the **-overwrite** and **-ignore-missing-libs** command line options.

-no-env-scripts
By default, *condor_configure* writes simple sh and csh shell scripts which can be sourced by their respective shells to set the user's `PATH` and `CONDOR_CONFIG` environment variables. This option prevents *condor_configure* from generating these scripts.

-env-scripts-dir=<directory>
By default, the simple *sh* and *csh* shell scripts (see **-no-env-scripts** for details) are created in the root directory of the HTCondor installation. This option causes *condor_configure* to generate these scripts in the specified directory.

-credd
Configure the the *condor_credd* daemon (credential manager daemon).

-verbose
Print information about changes to configuration variables as they occur.

15.21.4 Exit Status

condor_configure will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

15.21.5 Examples

Install HTCondor on the machine (machine1@cs.wisc.edu) to be the pool's central manager. On machine1, within the directory that contains the unzipped HTCondor distribution directories:

```
$ condor_install --type=submit,execute,manager
```

This will allow the machine to submit and execute HTCondor jobs, in addition to being the central manager of the pool.

To change the configuration such that machine2@cs.wisc.edu is an execute-only machine (that is, a dedicated computing node) within a pool with central manager on machine1@cs.wisc.edu, issue the command on that machine2@cs.wisc.edu from within the directory where HTCondor is installed:

```
$ condor_configure --central-manager=machine1@cs.wisc.edu --type=execute
```

To change the location of the `LOCAL_DIR` directory in the configuration file, do (from the directory where HTCondor is installed):

```
$ condor_configure --local-dir=/path/to/new/local/directory
```

This will move the `log`,`spool`,`execute` directories to `/path/to/new/local/directory` from the current local directory.

15.22 *condor_job_router_info*

Discover and display information related to job routing

15.22.1 Synopsis

condor_job_router_info [**-help | -version**]

condor_job_router_info -config

condor_job_router_info -match-jobs -jobads inputfile [**-ignore-prior-routing**]

condor_job_router_info -route-jobs outputfile -jobads inputfile [**-ignore-prior-routing**] [**-log-steps**]

15.22.2 Description

condor_job_router_info displays information about job routing. The information will be either the available, configured routes or the routes for specified jobs. *condor_job_router_info* can also be used to simulate routing by supplying a job classad in a file. This can be used to test the router configuration offline.

15.22.3 Options

-help
Display usage information and exit.

-version
Display HTCondor version information and exit.

-config
Display configured routes.

-match-jobs
For each job listed in the file specified by the **-jobads** option, display the first route found.

-route-jobs *filename*
For each job listed in the file specified by the **-jobads** option, apply the first route found and print the routed jobs to the specified output file. if *filename* is `-` the routed jobs are printed to `stdout`.

-log-steps
When used with the **-route-jobs** option, print each transform step as the job transforms are applied.

-ignore-prior-routing
For each job, remove any existing routing ClassAd attributes, and set attribute `JobStatus` to the Idle state before finding the first route.

-jobads *filename*
Read job ClassAds from file *filename*. If *filename* is `-`, then read from `stdin`.

15.22.4 Exit Status

condor_job_router_info will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.23 *condor_master*

The master HTCondor Daemon

15.23.1 Synopsis

condor_master

15.23.2 Description

This daemon is responsible for keeping all the rest of the HTCondor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the *condor_master* will restart the affected daemons. In addition, if any daemon crashes, the *condor_master* will send e-mail to the HTCondor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your HTCondor pool, regardless of what functions each machine are performing. Additionally, on Linux platforms, if you start the *condor_master* as root, it will tune (but never decrease) certain kernel parameters important to HTCondor's performance.

The `DAEMON_LIST` configuration macro is used by the *condor_master* to provide a per-machine list of daemons that should be started and kept running. For daemons that are specified in the `DC_DAEMON_LIST` configuration macro, the *condor_master* daemon will spawn them automatically appending a *-f* argument. For those listed in `DAEMON_LIST`, but not in `DC_DAEMON_LIST`, there will be no *-f* argument.

The *condor_master* creates certain directories necessary for its proper functioning on start-up if they don't already exist, using the values of the configuration settings `EXECUTE`, `LOCAL_DIR`, `LOCAL_DISK_LOCK_DIR`, `LOCAL_UNIV_EXECUTE`, `LOCK`, `LOG`, `RUN`, `SEC_CREDENTIAL_DIRECTORY_KRB`, `SEC_CREDENTIAL_DIRECTORY_OAUTH`, `SEC_PASSWORD_DIRECTORY`, `SEC_TOKEN_SYSTEM_DIRECTORY`, and `SPOOL`.

15.23.3 Options

-n ***name***
: Provides an alternate name for the *condor_master* to override that given by the `MASTER_NAME` configuration variable.

15.24 *condor_now*

Start a job now.

15.24.1 Synopsis

condor_now -help

condor_now [**-name]** [-debug**] *now-job vacate-job* [*vacate-job*+]

15.24.2 Description

condor_now tries to run the *now-job* now. The *vacate-job* is immediately vacated; after it terminates, if the schedd still has the claim to the vacated job's slot - and it usually will - the schedd will immediately start the now-job on that slot.

If you specify multiple *vacate-job* s, each will be immediately vacated; after they all terminate, the schedd will try to coalesce their slots into a single, larger, slot and then use that slot to run the now-job.

You must specify each job using both the cluster and proc IDs.

15.24.3 Options

-help
: Print a usage reminder.

-debug
: Print debugging output. Control the verbosity with the environment variables _CONDOR_TOOL_DEBUG, as usual.

-name **
: Specify the scheduler(‘s name) and (optionally) the pool to find it in.

15.24.4 General Remarks

The now-job and the vacated-job must have the same owner; if you are not the queue super-user, you must own both jobs. The jobs must be on the same schedd, and both jobs must be in the vanilla universe. The now-job must be idle and the vacated-job must be running.

15.24.5 Examples

To begin running job 17.3 as soon as possible using job 4.2's slot:

```
$ condor_now 17.3 4.2
```

To try to figure out why that doesn't work for the ‘magic’ scheduler in the ‘gandalf’ pool, set the environment variable _CONDOR_TOOL_DEBUG to ‘D_FULLDEBUG’ and then:

```
$ condor_now -debug -schedd magic -pool gandalf 17.3 4.2
```

15.24.6 Exit Status

condor_now will exit with a status value of 0 (zero) if the schedd accepts its request to vacate the vacate-job and start the now-job in its place. It does not wait for the now-job to have started running.

15.25 *condor_off*

Shutdown HTCondor daemons

15.25.1 Synopsis

condor_off [**-help | -version**]

condor_off [**-graceful | -fast | -peaceful | -force-graceful | -drain**] [**-annex** *name*] [**-debug[:opts]**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname* | **-master**] [**-exec** *name*] [**-reason** *"reason-string"*] [**-request-id** *id*] [**-check** *expr*] [**-start** *expr*]

15.25.2 Description

condor_off shuts down a set of the HTCondor daemons running on a set of one or more machines. By default, it does this cleanly, so that jobs have time to shut down.

The command *condor_off* without any arguments will shut down all daemons except *condor_master*, unless **-annex** *name* is specified. The *condor_master* can then handle both local and remote requests to restart the other HTCondor daemons if need be. To restart HTCondor running on a machine, see the *condor_on* command.

When the **-drain** option is chosen, draining options can be specified by using the optional **-reason**, **-request-id**, **-check**, and **-start** arguments.

With the **-daemon** *master* option, *condor_off* will shut down all daemons including the *condor_master*. Specification using the **-daemon** option will shut down only the specified daemon.

When shutting down all daemons including the *condor_master*, the **-exec** argument can be used to tell the master to run a configured script before it exits.

For security reasons of authentication and authorization, this command requires `ADMINISTRATOR` level of access.

15.25.3 Options

-help
: Display usage information

-version
: Display version information

-graceful
: The default. If jobs are running, wait for up to the configured grace period for them to finish, then exit

-fast
: Quickly shutdown daemons, immediately evicting any running jobs. A minimum of the first two characters of this option must be specified, to distinguish it from the **-force-graceful** command.

-peaceful
Wait indefinitely for jobs to finish

-force-graceful
Force a graceful shutdown, even after issuing a **-peaceful** command. A minimum of the first two characters of this option must be specified, to distinguish it from the **-fast** command.

-drain
Send a *condor_drain* command with the *-exit-on-completion* option to all *condor_startd* daemons that are managed by this master. Then wait for all *condor_startd* daemons to exit before before shutting down other daemons.

-reason *"reason-string"*
Use with **-drain** to set a **-reason** *"reason-string"* value for the *condor_drain* command.

-request-id *id*
Use with **-drain** to set a **-request-id** *id* value for the *condor_drain* command.

-check *expr*
Use with **-drain** to set a **-check** *expr* value for the *condor_drain* command.

-start *expr*
Use with **-drain** to set a **-start** *expr* value for the *condor_drain* command.

-annex *name*
Turn off master daemons in the specified annex. By default this will result in the corresponding instances shutting down.

-debug[:opts]
Causes debugging information to be sent to `stderr`. The debug level can be set by specifying an optional *opts* value. Otherwise, the configuration variable `TOOL_DEBUG` sets the debug level.

-pool *centralmanagerhostname[:portnumber]*
Specify a pool by giving the central manager's host name and an optional port number

-name *hostname*
Send the command to a machine identified by *hostname*

hostname
Send the command to a machine identified by *hostname*

-addr *"<a.b.c.d:port>"*
Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>"
Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint *expression*
Apply this command only to machines matching the given ClassAd *expression*

-all
Send the command to all machines in the pool

-master
Shutdown the *condor_master* after shutting down all other daemons.

-exec *name*
When used with **-master**, the *condor_master* will run the program configured as after shutting down all other daemons.

-daemon *daemonname*
Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

15.25.4 Graceful vs. Peaceful vs Fast

A "fast" shutdown will cause the requested daemon to exit. Jobs running under a startd that is shutdown fast will be evicted. Jobs running on a schedd that is shutdown fast will be left running for their job lease duration (default of 20 minutes). (That is, assuming the corresponding startd is not also being shut down). If that schedd restarts before the job lease expires, it will reconnect to these running jobs and continue to run them, as long as the schedd and startd are running.

A "graceful" shutdown of a schedd is functionally the same as a "fast" shutdown of a schedd.

A "graceful" shutdown of a startd that has jobs running under it causes the startd to wait for the jobs to exit of their own accord, up to the MaxJobRetirementTime. After the MaxJobRetirementTime, the startd will evict any remaining running jobs and exit.

A "peaceful" shutdown of a startd or schedd will cause that daemon to wait indefinitely for all existing jobs to exit before shutting down. During this time, no new jobs will start.

15.25.5 Exit Status

condor_off will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.25.6 Examples

To shut down all daemons (other than *condor_master*) on the local host:

```
$ condor_off
```

To shut down only the *condor_collector* on three named machines:

```
$ condor_off  cinnamon cloves vanilla -daemon collector
```

To shut down daemons within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command shuts down all daemons except the *condor_master* on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
$ condor_off  -pool condor.cae.wisc.edu -name cae17
```

15.26 *condor_on*

Start up HTCondor daemons

15.26.1 Synopsis

condor_on [**-help | -version**]

condor_on [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname*]

15.26.2 Description

condor_on starts up a set of the HTCondor daemons on a set of machines. This command assumes that the *condor_master* is already running on the machine. If this is not the case, *condor_on* will fail complaining that it cannot find the address of the master. The command *condor_on* with no arguments or with the **-daemon** *master* option will tell the *condor_master* to start up the HTCondor daemons specified in the configuration variable `DAEMON_LIST`. If a daemon other than the *condor_master* is specified with the **-daemon** option, *condor_on* starts up only that daemon.

This command cannot be used to start up the *condor_master* daemon.

For security reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

15.26.3 Options

-help
: Display usage information

-version
: Display version information

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-pool ***centralmanagerhostname[:portnumber]***
: Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname***
: Send the command to a machine identified by *hostname*

hostname
: Send the command to a machine identified by *hostname*

-addr ***"<a.b.c.d:port>"***
: Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>"
: Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression***
: Apply this command only to machines matching the given ClassAd *expression*

-all
: Send the command to all machines in the pool

-daemon ***daemonname***
: Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

15.26.4 Exit Status

condor_on will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.26.5 Examples

To begin running all daemons (other than *condor_master*) given in the configuration variable `DAEMON_LIST` on the local host:

```
$ condor_on
```

To start up only the *condor_negotiator* on two named machines:

```
$ condor_on  robin cardinal -daemon negotiator
```

To start up only a daemon within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command starts up only the *condor_schedd* daemon on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
$ condor_on -pool condor.cae.wisc.edu -name cae17 -daemon schedd
```

15.27 *condor_ping*

Attempt a security negotiation to determine if it succeeds

15.27.1 Synopsis

condor_ping [**-help | -version**]

condor_ping [**-debug**] [**-address** <*a.b.c.d:port*>] [**-pool** *host name*] [**-name** *daemon name*] [**-type** *subsystem*] [**-config** *filename*] [**-quiet | -table | -verbose**] *token* [*token [. . .]*]

15.27.2 Description

condor_ping attempts a security negotiation to discover whether the configuration is set such that the negotiation succeeds. The target of the negotiation is defined by one or a combination of the **address**, **pool**, **name**, or **type** options. If no target is specified, the default target is the *condor_schedd* daemon on the local machine.

One or more *token* s may be listed, thereby specifying one or more authorization level to impersonate in security negotiation. A token is the value `ALL`, an authorization level, a command name, or the integer value of a command. The many command names and their associated integer values will more likely be used by experts, and they are defined in the file `condor_includes/condor_commands.h`.

An authorization level may be one of the following strings. If `ALL` is listed, then negotiation is attempted for each of these possible authorization levels. Note that OWNER is no longer used in HTCondor, but is kept here for use when talking to older daemons (prior to 9.9.0).

> READ WRITE ADMINISTRATOR SOAP CONFIG OWNER DAEMON NEGOTIATOR ADVERTISE_MASTER ADVERTISE_STARTD ADVERTISE_SCHEDD CLIENT

15.27.3 Options

-help
Display usage information

-version
Display version information

-debug
Print extra debugging information as the command executes.

-config ***filename***
Attempt the negotiation based on the contents of the configuration file contents in file *filename*.

-address <*a.b.c.d:port*>
Target the given IP address with the negotiation attempt.

-pool ***hostname***
Target the given *host* with the negotiation attempt. May be combined with specifications defined by **name** and **type** options.

-name ***daemonname***
Target the daemon given by *daemonname* with the negotiation attempt.

-type ***subsystem***
Target the daemon identified by *subsystem*, one of the values of the predefined `$(SUBSYSTEM)` macro.

-quiet
Set exit status only; no output displayed.

-table
Output is displayed with one result per line, in a table format.

-verbose
Display all available output.

15.27.4 Examples

The example Unix command

```
$ condor_ping  -address "<127.0.0.1:9618>" -table READ WRITE DAEMON
```

places double quote marks around the sinful string to prevent the less than and the greater than characters from causing redirect of input and output. The given IP address is targeted with 3 attempts to negotiate: one at the `READ` authorization level, one at the `WRITE` authorization level, and one at the `DAEMON` authorization level.

15.27.5 Exit Status

condor_ping will exit with the status value of the negotiation it attempted, where 0 (zero) indicates success, and 1 (one) indicates failure. If multiple security negotiations were attempted, the exit status will be the logical OR of all values.

15.28 *condor_pool_job_report*

generate report about all jobs that have run in the last 24 hours on all execute hosts

15.28.1 Synopsis

condor_pool_job_report

15.28.2 Description

condor_pool_job_report is a Linux-only tool that is designed to be run nightly using *cron*. It is intended to be run on the central manager, or another machine that has administrative permissions, and is able to fetch the *condor_startd* history logs from all of the *condor_startd* daemons in the pool. After fetching these logs, *condor_pool_job_report* then generates a report about job run times and mails it to administrators, as defined by configuration variable `CONDOR_ADMIN` .

15.28.3 Exit Status

condor_pool_job_report will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.29 *condor_power*

send packet intended to wake a machine from a low power state

15.29.1 Synopsis

condor_power [**-h**]

condor_power [**-d**] [**-i**] [**-m** *MACaddress*] [**-s** *subnet*] [*ClassAdFile*]

15.29.2 Description

condor_power sends one UDP Wake on LAN (WOL) packet to a machine specified either by command line arguments or by the contents of a machine ClassAd. The machine ClassAd may be in a file, where the file name specified by the optional argument *ClassAdFile* is given on the command line. With no command line arguments to specify the machine, and no file specified, *condor_power* quietly presumes that standard input is the file source which will specify the machine ClassAd that includes the public IP address and subnet of the machine.

condor_power needs a complete specification of the machine to be successful. If a MAC address is provided on the command line, but no subnet is given, then the default value for the subnet is used. If a subnet is provided on the command line, but no MAC address is given, then *condor_power* falls back to taking its information in the form of the machine ClassAd as provided in a file or on standard input. Note that this case implies that the command line specification of the subnet is ignored.

condor_power relies on the router receiving the WOL packet to correctly broadcast the request. Since routers are often configured to ignore requests to broadcast messages on a different subnet than the sender, the send of a WOL packet to a machine on a different subnet may fail.

15.29.3 Options

-h
Print usage information and exit.

-d
Enable debugging messages.

-i
Read a ClassAd that is piped in through standard input.

-m ***MACaddress***
Specify the MAC address in the standard format of six groups of two hexadecimal digits separated by colons.

-s ***subnet***
Specify the subnet in the standard form of a mask for an IPv4 address. Without this option, a global broadcast will be sent.

15.29.4 Exit Status

condor_power will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.30 *condor_preen*

remove extraneous files from HTCondor directories

15.30.1 Synopsis

condor_preen [**-mail**] [**-remove**] [**-verbose**] [**-debug**] [**-log <filename>**]

15.30.2 Description

condor_preen examines the directories belonging to HTCondor, and removes extraneous files and directories which may be left over from HTCondor processes which terminated abnormally either due to internal errors or a system crash. The directories checked are the `LOG`, `EXECUTE`, and `SPOOL` directories as defined in the HTCondor configuration files. *condor_preen* is intended to be run as user root or user condor periodically as a backup method to ensure reasonable file system cleanliness in the face of errors. This is done automatically by default by the *condor_master* daemon. It may also be explicitly invoked on an as needed basis.

When *condor_preen* cleans the `SPOOL` directory, it always leaves behind the files specified in the configuration variables `VALID_SPOOL_FILES` and `SYSTEM_VALID_SPOOL_FILES` , as given by the configuration. For the `LOG` directory, the only files removed or reported are those listed within the configuration variable `INVALID_LOG_FILES` list. The reason for this difference is that, in general, the files in the `LOG` directory ought to be left alone, with few exceptions. An example of exceptions are core files. As there are new log files introduced regularly, it is less effort to specify those that ought to be removed than those that are not to be removed.

15.30.3 Options

-mail
Send mail to the user defined in the `PREEN_ADMIN` configuration variable, instead of writing to the standard output.

-remove
Remove the offending files and directories rather than reporting on them.

-verbose
List all files or directories found in the Condor directories and considered for deletion, even those which are not extraneous. This option also modifies the output produced by the **-debug** and **-log** options

-debug
Print extra debugging information to stderr as the command executes.

-log <filename>
Write extra debugging information to <filename> as the command executes.

15.30.4 Exit Status

condor_preen will exit with a status value of 0 (zero) upon success, and it will exit with a non-zero value upon failure. An exit status of 2 indicates that *condor_preen* attempted to send email about deleted files but was unable to. This usually indicates an error in the configuration for sending email. An exit status of 1 indicates a general failure.

15.31 *condor_prio*

change priority of jobs in the HTCondor queue

15.31.1 Synopsis

condor_prio *-p priority* | *+value* | *-value* [*-n schedd_name*] [**username** | **ClusterId**]

15.31.2 Description

condor_prio changes the priority of one or more jobs in the HTCondor queue. If the job identification is given by *cluster.process*, *condor_prio* attempts to change the priority of the single job with job ClassAd attributes `ClusterId` and `ProcId`. If described by *cluster*, *condor_prio* attempts to change the priority of all processes with the given `ClusterId` job ClassAd attribute. If *username* is specified, *condor_prio* attempts to change priority of all jobs belonging to that user. For **-a**, *condor_prio* attempts to change priority of all jobs in the queue.

The user must set a new priority with the **-p** option, or specify a priority adjustment.

The priority of a job can be any integer, with higher numbers corresponding to greater priority. For adjustment of the current priority, *+value* increases the priority by the amount given with *value*. *-value* decreases the priority by the amount given with *value*.

Only the owner of a job or the super user can change the priority.

The priority changed by *condor_prio* is only used when comparing to the priority jobs owned by the same user and submitted from the same machine.

15.31.3 Options

-a
Change priority of all jobs in the queue

-n ***schedd_name***
Change priority of jobs queued at the specified *condor_schedd* in the local pool.

-pool ***pool_name*** **-n** ***schedd_name***
Change priority of jobs queued at the specified *condor_schedd* in the specified pool.

15.31.4 Exit Status

condor_prio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.32 *condor_procd*

Track and manage process families

15.32.1 Synopsis

condor_procd -h

condor_procd -A *address-file* [**options**]

15.32.2 Description

condor_procd tracks and manages process families on behalf of the HTCondor daemons. It may track families of PIDs via relationships such as: direct parent/child, environment variables, UID, and supplementary group IDs. Management of the PID families include

- registering new families or new members of existing families
- getting usage information
- signaling families for operations such as suspension, continuing, or killing the family
- getting a snapshot of the tree of families

In a regular HTCondor installation, this program is not intended to be used or executed by any human.

The required argument, **-A** *address-file*, is the path and file name of the address file which is the named pipe that clients must use to speak with the *condor_procd*.

15.32.3 Options

-h
Print out usage information and exit.

-D
Wait for the debugger. Initially sleep 30 seconds before beginning normal function.

-C *principal*
The *principal* is the UID of the owner of the named pipe that clients must use to speak to the *condor_procd*.

-L *log-file*
A file the *condor_procd* will use to write logging information.

-E
When specified, another tool such as the *procd_ctl* tool must allocate the GID associated with a process. When this option is not specified, the *condor_procd* will allocate the GID itself.

-P *PID*
If not specified, the *condor_procd* will use the *condor_procd* 's parent, which may not be PID 1 on Unix, as the parent of the *condor_procd* and the root of the tracking family. When not specified, if the *condor_procd* 's parent PID dies, the *condor_procd* exits. When specified, the *condor_procd* will track this *PID* family in question and not also exit if the PID exits.

-S *seconds*
The maximum number of seconds the *condor_procd* will wait between taking snapshots of the tree of families. Different clients to the *condor_procd* can specify different snapshot times. The quickest snapshot time is the one performed by the *condor_procd*. When this option is not specified, a default value of 60 seconds is used.

-G *min-gid max-gid*
If the **-E** option is not specified, then track process families using a self-allocated, free GID out of the inclusive range specified by *min-gid* and *max-gid*. This means that if a new process shows up using a previously known GID, the new process will automatically associate into the process family assigned that GID. If the **-E** option is specified, then instead of self-allocating the GID, the *procd_ctl* tool must be used to associate the GID with the PID root of the family. The associated GID must still be in the range specified. This is a Linux-only feature.

-K *windows-softkill-binary*
This is the path and executable name of the *condor_softkill.exe* binary. It is used to send softkill signals to process families. This is a Windows-only feature.

15.32.4 Dealing with Short Reads

For unknown reasons, on Linux, attemps to read the list of PIDs from the /proc filesystem do not always return all of the PIDs on the system. The *condor_procd* attempts to detect when this occurs, using two methods.

If the list of PIDs does not include PID 1, the *condor_procd*'s own PID, or the PID of the *condor_procd*'s parent (which may be PID 1), then the list must be incomplete, and the *condor_procd* immediately retries the read.

Additionally, the *condor_procd* compares the number of PIDs it just read to the number of PIDs from the last time it (successfully) checked. If the number is too much smaller, it immediately retries. The default threshold is 0.90, meaning that if the current read has 90% or fewer of the last read's PIDs, it's considered invalid. In our testing, this threshold was met by roughly 1 in 4000 reads, but successfully detected all real short reads. If you need to adjust the threshold, you may do so by setting the environment variable _CONDOR_PROCAPI_RETRY_FRACTION. (In the normal case, simply have it in the environment when the *condor_master* starts up.)

If a retried read is incomplete (according to either method), the *condor_procd* returns the results of the previous read.

15.32.5 General Remarks

This program may be used in a stand alone mode, independent of HTCondor, to track process families. The programs *procd_ctl* and *gidd_alloc* are used with the *condor_procd* in stand alone mode to interact with the daemon and to inquire about certain state of running processes on the machine, respectively.

15.32.6 Exit Status

condor_procd will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.33 *condor_q*

Display information about jobs in queue

15.33.1 Synopsis

condor_q [**-help [Universe | State]**]

condor_q [**-debug**] [*general options*] [*restriction list*] [*output options*] [*analyze options*]

15.33.2 Description

condor_q displays information about jobs in the HTCondor job queue. By default, *condor_q* queries the local job queue, but this behavior may be modified by specifying one of the general options.

As of version 8.5.2, condor_q defaults to querying only the current user's jobs. This default is overridden when the restriction list has usernames and/or job ids, when the -submitter or -allusers arguments are specified, or when the current user is a queue superuser. It can also be overridden by setting the CONDOR_Q_ONLY_MY_JOBS configuration macro to False.

As of version 8.5.6, condor_q defaults to batch-mode output (see -batch in the Options section below). The old behavior can be obtained by specifying -nobatch on the command line. To change the default back to its pre-8.5.6 value, set the new configuration variable CONDOR_Q_DASH_BATCH_IS_DEFAULT to False.

15.33.3 Batches of jobs

As of version 8.5.6, *condor_q* defaults to displaying information about batches of jobs, rather than individual jobs. The intention is that this will be a more useful, and user-friendly, format for users with large numbers of jobs in the queue. Ideally, users will specify meaningful batch names for their jobs, to make it easier to keep track of related jobs.

(For information about specifying batch names for your jobs, see the *condor_submit* and *condor_submit_dag* manual pages.)

A batch of jobs is defined as follows:

- An entire workflow (a DAG or hierarchy of nested DAGs) (note that *condor_dagman* now specifies a default batch name for all jobs in a given workflow)
- All jobs in a single cluster

- All jobs submitted by a single user that have the same executable specified in their submit file (unless submitted with different batch names)
- All jobs submitted by a single user that have the same batch name specified in their submit file or on the *condor_submit* or *condor_submit_dag* command line.

15.33.4 Output

There are many output options that modify the output generated by *condor_q*. The effects of these options, and the meanings of the various output data, are described below.

Output options

If the **-long** option is specified, *condor_q* displays a long description of the queried jobs by printing the entire job ClassAd for all jobs matching the restrictions, if any. Individual attributes of the job ClassAd can be displayed by means of the **-format** option, which displays attributes with a printf(3) format, or with the **-autoformat** option. Multiple **-format** options may be specified in the option list to display several attributes of the job.

For most output options (except as specified), the last line of *condor_q* output contains a summary of the queue: the total number of jobs, and the number of jobs in the completed, removed, idle, running, held and suspended states.

If no output options are specified, *condor_q* now defaults to batch mode, and displays the following columns of information, with one line of output per batch of jobs:

```
OWNER, BATCH_NAME, SUBMITTED, DONE, RUN, IDLE, [HOLD,] TOTAL, JOB_IDS
```

Note that the HOLD column is only shown if there are held jobs in the output or if there are no jobs in the output.

If the **-nobatch** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, SUBMITTED, RUN_TIME, ST, PRI, SIZE, CMD
```

If the **-dag** option is specified (in conjunction with **-nobatch**), *condor_q* displays the following columns of information, with one line of output per job; the owner is shown only for top-level jobs, and for all other jobs (including sub-DAGs) the node name is shown:

```
ID, OWNER/NODENAME, SUBMITTED, RUN_TIME, ST, PRI, SIZE, CMD
```

If the **-run** option is specified (in conjunction with **-nobatch**), *condor_q* displays the following columns of information, with one line of output per running job:

```
ID, OWNER, SUBMITTED, RUN_TIME, HOST(S)
```

Also note that the **-run** option disables output of the totals line.

If the **-grid** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, STATUS, GRID->MANAGER, HOST, GRID_JOB_ID
```

If the **-grid:ec2** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, STATUS, INSTANCE ID, CMD
```

If the **-goodput** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, SUBMITTED, RUN_TIME, GOODPUT, CPU_UTIL, Mb/s
```

If the **-io** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, RUNS, ST, INPUT, OUTPUT, RATE, MISC
```

If the **-cputime** option is specified (in conjunction with **-nobatch**), *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, SUBMITTED, CPU_TIME, ST, PRI, SIZE, CMD
```

If the **-hold** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, HELD_SINCE, HOLD_REASON
```

If the **-totals** option is specified, *condor_q* displays only one line of output no matter how many jobs and batches of jobs are in the queue. That line of output contains the total number of jobs, and the number of jobs in the completed, removed, idle, running, held and suspended states.

Output data

The available output data are as follows:

ID
(Non-batch mode only) The cluster/process id of the HTCondor job.

OWNER
The owner of the job or batch of jobs.

OWNER/NODENAME
(**-dag** only) The owner of a job or the DAG node name of the job.

BATCH_NAME
(Batch mode only) The batch name of the job or batch of jobs.

SUBMITTED
The month, day, hour, and minute the job was submitted to the queue.

DONE
(Batch mode only) The number of job procs that are done, but still in the queue.

RUN
(Batch mode only) The number of job procs that are running.

IDLE
(Batch mode only) The number of job procs that are in the queue but idle.

HOLD
(Batch mode only) The number of job procs that are in the queue but held.

TOTAL
(Batch mode only) The total number of job procs in the queue, unless the batch is a DAG, in which case this is the total number of clusters in the queue. Note: for non-DAG batches, the TOTAL column contains correct values only in version 8.5.7 and later.

JOB_IDS
(Batch mode only) The range of job IDs belonging to the batch.

RUN_TIME
(Non-batch mode only) Wall-clock time accumulated by the job currently running in days, hours, minutes, and seconds. When the job is idle or held the jobs previous accumulated time will be displayed.

ST
(Non-batch mode only) Current status of the job, which varies somewhat according to the job universe and the timing of updates. H = on hold, R = running, I = idle (waiting for a machine to execute on), C = completed, X = removed, S = suspended (execution of a running job temporarily suspended on execute node), < = transferring input (or queued to do so), and > = transferring output (or queued to do so).

PRI
(Non-batch mode only) User specified priority of the job, displayed as an integer, with higher numbers corresponding to better priority.

SIZE
(Non-batch mode only) The peak amount of memory in Mbytes consumed by the job; note this value is only refreshed periodically. The actual value reported is taken from the job ClassAd attribute `MemoryUsage` if this attribute is defined, and from job attribute `ImageSize` otherwise.

CMD
(Non-batch mode only) The name of the executable. For EC2 jobs, this field is arbitrary.

HOST(S)
(**-run** only) The host where the job is running.

STATUS
(**-grid** only) The state that HTCondor believes the job is in. Possible values are grid-type specific, but include:

PENDING
The job is waiting for resources to become available in order to run.

ACTIVE
The job has received resources, and the application is executing.

FAILED
The job terminated before completion because of an error, user-triggered cancel, or system-triggered cancel.

DONE
The job completed successfully.

SUSPENDED
The job has been suspended. Resources which were allocated for this job may have been released due to a scheduler-specific reason.

STAGE_IN
The job manager is staging in files, in order to run the job.

STAGE_OUT
The job manager is staging out files generated by the job.

UNKNOWN
Unknown

GRID->MANAGER
(**-grid** only) A guess at what remote batch system is running the job. It is a guess, because HTCondor

looks at the jobmanager contact string to attempt identification. If the value is fork, the job is running on the remote host without a jobmanager. Values may also be condor, lsf, or pbs.

HOST
(**-grid** only) The host to which the job was submitted.

GRID_JOB_ID
(**-grid** only) (More information needed here.)

INSTANCE ID
(**-grid:ec2** only) Usually EC2 instance ID; may be blank or the client token, depending on job progress.

GOODPUT
(**-goodput** only) The percentage of RUN_TIME for this job which has been saved in a checkpoint. A low GOODPUT value indicates that the job is failing to checkpoint. If a job has not yet attempted a checkpoint, this column contains `[?????]`.

CPU_UTIL
(**-goodput** only) The ratio of CPU_TIME to RUN_TIME for checkpointed work. A low CPU_UTIL indicates that the job is not running efficiently, perhaps because it is I/O bound or because the job requires more memory than available on the remote workstations. If the job has not (yet) checkpointed, this column contains `[??????]`.

Mb/s
(**-goodput** only) The network usage of this job, in Megabits per second of run-time. READ The total number of bytes the application has read from files and sockets. WRITE The total number of bytes the application has written to files and sockets. SEEK The total number of seek operations the application has performed on files. XPUT The effective throughput (average bytes read and written per second) from the application's point of view. BUFSIZE The maximum number of bytes to be buffered per file. BLOCKSIZE The desired block size for large data transfers. These fields are updated when a job produces a checkpoint or completes. If a job has not yet produced a checkpoint, this information is not available.

INPUT
(**-io** only) BytesRecvd.

OUTPUT
(**-io** only) BytesSent.

RATE
(**-io** only) BytesRecvd+BytesSent.

MISC
(**-io** only) JobUniverse.

CPU_TIME
(**-cputime** only) The remote CPU time accumulated by the job to date (which has been stored in a checkpoint) in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is not shown. If the job has not produced a checkpoint, this column contains 0+00:00:00.)

HELD_SINCE
(**-hold** only) Month, day, hour and minute at which the job was held.

HOLD_REASON
(**-hold** only) The hold reason for the job.

Analyze

The **-analyze** or **-better-analyze** options can be used to determine why certain jobs are not running by performing an analysis on a per machine basis for each machine in the pool. The reasons can vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the `PREEMPTION_REQUIREMENTS` expression. If the analyze option **-verbose** is specified along with the **-analyze** option, the reason for failure is displayed on a per machine basis. **-better-analyze** differs from **-analyze** in that it will do matchmaking analysis on jobs even if they are currently running, or if the reason they are not running is not due to matchmaking. **-better-analyze** also produces more thorough analysis of complex Requirements and shows the values of relevant job ClassAd attributes. When only a single machine is being analyzed via **-machine** or **-mconstraint**, the values of relevant attributes of the machine ClassAd are also displayed.

15.33.5 Restrictions

To restrict the display to jobs of interest, a list of zero or more restriction options may be supplied. Each restriction may be one of:

- **cluster.process**, which matches jobs which belong to the specified cluster and have the specified process number;
- **cluster** (without a *process*), which matches all jobs belonging to the specified cluster;
- **owner**, which matches all jobs owned by the specified owner;
- **-constraint expression**, which matches all jobs that satisfy the specified ClassAd expression;
- **-unmatchable expression**, which matches all jobs that do not match any slot that would be considered by **-better-analyze** *;*
- **-allusers**, which overrides the default restriction of only matching jobs submitted by the current user.

If *cluster* or *cluster.process* is specified, and the job matching that restriction is a *condor_dagman* job, information for all jobs of that DAG is displayed in batch mode (in non-batch mode, only the *condor_dagman* job itself is displayed).

If no *owner* restrictions are present, the job matches the restriction list if it matches at least one restriction in the list. If *owner* restrictions are present, the job matches the list if it matches one of the *owner* restrictions and at least one non-*owner* restriction.

15.33.6 Options

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-batch
(output option) Show a single line of progress information for a batch of jobs, where a batch is defined as follows:

- An entire workflow (a DAG or hierarchy of nested DAGs)
- All jobs in a single cluster
- All jobs submitted by a single user that have the same executable specified in their submit file
- All jobs submitted by a single user that have the same batch name specified in their submit file or on the *condor_submit* or *condor_submit_dag* command line.

Also change the output columns as noted above.

Note that, as of version 8.5.6, **-batch** is the default, unless the `CONDOR_Q_DASH_BATCH_IS_DEFAULT` configuration variable is set to `False`.

-nobatch
(output option) Show a line for each job (turn off the **-batch** option).

-global
(general option) Queries all job queues in the pool.

-submitter ***submitter***
(general option) List jobs of a specific submitter in the entire pool, not just for a single *condor_schedd*.

-name ***name***
(general option) Query only the job queue of the named *condor_schedd* daemon.

-pool ***centralmanagerhostname[:portnumber]***
(general option) Use the *centralmanagerhostname* as the central manager to locate *condor_schedd* daemons. The default is the `COLLECTOR_HOST`, as specified in the configuration.

-jobads ***file***
(general option) Display jobs from a list of ClassAds from a file, instead of the real ClassAds from the *condor_schedd* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_q* **-long** is used with the header stripped out.

-userlog ***file***
(general option) Display jobs, with job information coming from a job event log, instead of from the real ClassAds from the *condor_schedd* daemon. This is most useful for automated testing of the status of jobs known to be in the given job event log, because it reduces the load on the *condor_schedd*. A job event log does not contain all of the job information, so some fields in the normal output of *condor_q* will be blank.

-factory
(output option) Display information about late materialization job factories in the *condor_shedd*.

-autocluster
(output option) Output *condor_schedd* daemon auto cluster information. For each auto cluster, output the unique ID of the auto cluster along with the number of jobs in that auto cluster. This option is intended to be used together with the **-long** option to output the ClassAds representing auto clusters. The ClassAds can then be used to identify or classify the demand for sets of machine resources, which will be useful in the on-demand creation of execute nodes for glidein services.

-cputime
(output option) Instead of wall-clock allocation time (RUN_TIME), display remote CPU time accumulated by the job to date in days, hours, minutes, and seconds. If the job is currently running, time accumulated during the current run is not shown. Note that this option has no effect unless used in conjunction with **-nobatch**.

-currentrun
(output option) If this option is specified, RUN_TIME displays the time accumulated so far on this current run unless the job is in IDLE or HELD state then RUN_TIME will display the previous runs time. Note that this is the base behavior and is not required, and this option cannot be used in conjunction with **-cumulative-time**.

-cumulative-time
(output option) Normally, RUN_TIME contains the current or previous runs accumulated wall-clock time. If this option is specified, RUN_TIME displays the accumulated time for the current run plus all previous runs. Note that this option cannot be used in conjunction with **-currentrun**.

-dag
(output option) Display DAG node jobs under their DAGMan instance. Child nodes are listed using indentation to show the structure of the DAG. Note that this option has no effect unless used in conjunction with **-nobatch**.

-expert
(output option) Display shorter error messages.

-grid
(output option) Get information only about jobs submitted to grid resources.

-grid:ec2
(output option) Get information only about jobs submitted to grid resources and display it in a format better-suited for EC2 than the default.

-goodput
(output option) Display job goodput statistics.

-help [Universe | State]
(output option) Print usage info, and, optionally, additionally print job universes or job states.

-hold
(output option) Get information about jobs in the hold state. Also displays the time the job was placed into the hold state and the reason why the job was placed in the hold state.

-limit ***Number***
(output option) Limit the number of items output to *Number*.

-io
(output option) Display job input/output summaries.

-long
(output option) Display entire job ClassAds in long format (one attribute per line).

-idle
(output option) Get information about idle jobs. Note that this option implies **-nobatch**.

-run
(output option) Get information about running jobs. Note that this option implies **-nobatch**.

-stream-results
(output option) Display results as jobs are fetched from the job queue rather than storing results in memory until all jobs have been fetched. This can reduce memory consumption when fetching large numbers of jobs, but if *condor_q* is paused while displaying results, this could result in a timeout in communication with *condor_schedd*.

-totals
(output option) Display only the totals.

-version
(output option) Print the HTCondor version and exit.

-wide
(output option) If this option is specified, and the command portion of the output would cause the output to extend beyond 80 columns, display beyond the 80 columns.

-xml
(output option) Display entire job ClassAds in XML format. The XML format is fully defined in the reference manual, obtained from the ClassAds web page, with a link at http://htcondor.org/classad/classad.html.

-json
(output option) Display entire job ClassAds in JSON format.

-attributes ***Attr1[,Attr2 . . .]***
(output option) Explicitly list the attributes, by name in a comma separated list, which should be

displayed when using the **-xml**, **-json** or **-long** options. Limiting the number of attributes increases the efficiency of the query.

-format *fmt attr*
(output option) Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)`-style conversion specifier. Attributes must be from the job ClassAd. Expressions are ClassAd expressions and may refer to attributes in the job ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. %r prints the unevaluated, or raw values. The conversion specifier must match the type of the attribute or expression. %s is suitable for strings such as `Owner`, %d for integers such as `ClusterId`, and %f for floating point numbers such as `RemoteWallClockTime`. %v identifies the type of the attribute, and then prints the value in an appropriate format. %V identifies the type of the attribute, and then prints the value in an appropriate format as it would appear in the **-long** format. As an example, strings used with %V will have quote marks. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)` style formats, one may include other text that will be reproduced directly. A format without any conversion specifiers may be specified, but an attribute is still required. Include a backslash followed by an 'n' to specify a line break.

-autoformat[:jlhVr,tng] *attr1 [attr2 ...]* **or -af[:jlhVr,tng]** *attr1 [attr2 ...]*
(output option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings. This output option does not work in conjunction with any of the options **-run**, **-currentrun**, **-hold**, **-grid**, **-goodput**, or **-io**.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

j print the job ID as the first field,

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may not be used together. The **l** and **h** characters may not be used together.

-print-format *file*
Read output formatting information from the given custom print format file. see *Print Formats* for more information about custom print format files.

-analyze[:<qual>]
(analyze option) Perform a matchmaking analysis on why the requested jobs are not running. First a simple analysis determines if the job is not running due to not being in a runnable state. If the job is in a runnable state, then this option is equivalent to **-better-analyze**. **<qual>** is a comma separated list containing one or more of

priority to consider user priority during the analysis

summary to show a one line summary for each job or machine

reverse to analyze machines, rather than jobs

-better-analyze[:<qual>]
(analyze option) Perform a more detailed matchmaking analysis to determine how many resources are available to run the requested jobs. This option is never meaningful for Scheduler universe jobs and only meaningful for grid universe jobs doing matchmaking. When this option is used in conjunction with the **-unmatchable** option, The output will be a list of job ids that don't match any of the available slots. **<qual>** is a comma separated list containing one or more of

priority to consider user priority during the analysis

summary to show a one line summary for each job or machine

reverse to analyze machines, rather than jobs

-machine *name*
(analyze option) When doing matchmaking analysis, analyze only machine ClassAds that have slot or machine names that match the given name.

-mconstraint *expression*
(analyze option) When doing matchmaking analysis, match only machine ClassAds which match the ClassAd expression constraint.

-slotads *file*
(analyze option) When doing matchmaking analysis, use the machine ClassAds from the file instead of the ones from the *condor_collector* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_status* **-long** is used.

-userprios *file*
(analyze option) When doing matchmaking analysis with priority, read user priorities from the file rather than the ones from the *condor_negotiator* daemon. This is most useful for debugging purposes or to speed up analysis in situations where the *condor_negotiator* daemon is slow to respond to *condor_userprio* requests. The file should be in the format produced by *condor_userprio* **-long**.

-nouserprios
(analyze option) Do not consider user priority during the analysis.

-reverse-analyze
(analyze option) Analyze machine requirements against jobs.

-verbose
(analyze option) When doing analysis, show progress and include the names of specific machines in the output.

15.33.7 General Remarks

The default output from *condor_q* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the HTCondor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_q* is strongly encouraged to use the **-format** option (described above, examples given below).

Although **-analyze** provides a very good first approximation, the analyzer cannot diagnose all possible situations, because the analysis is based on instantaneous and local information. Therefore, there are some situations such as when several submitters are contending for resources, or if the pool is rapidly changing state which cannot be accurately diagnosed.

It is possible to hold jobs that are in the X state. To avoid this it is best to construct a **-constraint** *expression* that option contains `JobStatus != 3` if the user wishes to avoid this condition.

15.33.8 Examples

The **-format** option provides a way to specify both the job attributes and formatting of those attributes. There must be only one conversion specification per **-format** option. As an example, to list only Jane Doe's jobs in the queue, choosing to print and format only the owner of the job, the command line arguments for the job, and the process ID of the job:

```
$ condor_q -submitter jdoe -format "%s" Owner -format " %s " Args -format␣
↪" ProcId = %d\n" ProcId
jdoe 16386 2800 ProcId = 0
jdoe 16386 3000 ProcId = 1
jdoe 16386 3200 ProcId = 2
jdoe 16386 3400 ProcId = 3
jdoe 16386 3600 ProcId = 4
jdoe 16386 4200 ProcId = 7
```

To display only the JobID's of Jane Doe's jobs you can use the following.

```
$ condor_q -submitter jdoe -format "%d." ClusterId -format "%d\n" ProcId
27.0
27.1
27.2
27.3
27.4
27.7
```

An example that shows the analysis in summary format:

```
$ condor_q -analyze:summary

-- Submitter: submit-1.chtc.wisc.edu : <192.168.100.43:9618?sock=11794_95bb_3> :
 submit-1.chtc.wisc.edu
Analyzing matches for 5979 slots
            Autocluster  Matches    Machine     Running  Serving
 JobId     Members/Idle  Reqmnts  Rejects Job  Users Job Other User Avail Owner
---------- ------------ -------- ------------ ---------- ---------- ----- -----
25764522.0  7/0             5910        820   7/10       5046        34   smith
25764682.0  9/0             2172        603   9/9        1531        29   smith
```

(continues on next page)

(continued from previous page)

```
25765082.0  18/0            2172        603   18/9       1531        29   smith
25765900.0  1/0             2172        603   1/9        1531        29   smith
```

An example that shows summary information by machine:

```
$ condor_q -ana:sum,rev

-- Submitter: s-1.chtc.wisc.edu : <192.168.100.43:9618?sock=11794_95bb_3> : s-1.chtc.
↪wisc.edu
Analyzing matches for 2885 jobs
                             Slot  Slot's Req    Job's Req     Both
Name                         Type  Matches Job  Matches Slot    Match %
-------------------------    ----  -----------  ------------ ----------
slot1@INFO.wisc.edu          Stat         2729  0                 0.00
slot2@INFO.wisc.edu          Stat         2729  0                 0.00
slot1@aci-001.chtc.wisc.edu  Part            0  2793              0.00
slot1_1@a-001.chtc.wisc.edu  Dyn          2644  2792             91.37
slot1_2@a-001.chtc.wisc.edu  Dyn          2623  2601             85.10
slot1_3@a-001.chtc.wisc.edu  Dyn          2644  2632             85.82
slot1_4@a-001.chtc.wisc.edu  Dyn          2644  2792             91.37
slot1@a-002.chtc.wisc.edu    Part            0  2633              0.00
slot1_10@a-002.chtc.wisc.edu Den          2623  2601             85.10
```

An example with two independent DAGs in the queue:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:35169?...
OWNER  BATCH_NAME    SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
wenger DAG: 3696    2/12 11:55      _     10       _     10 3698.0 ... 3707.0
wenger DAG: 3697    2/12 11:55      1      1       1     10 3709.0 ... 3710.0

14 jobs; 0 completed, 0 removed, 1 idle, 13 running, 0 held, 0 suspended
```

Note that the "13 running" in the last line is two more than the total of the RUN column, because the two *condor_dagman* jobs themselves are counted in the last line but not the RUN column.

Also note that the "completed" value in the last line does not correspond to the total of the DONE column, because the "completed" value in the last line only counts jobs that are completed but still in the queue, whereas the DONE column counts jobs that are no longer in the queue.

Here's an example with a held job, illustrating the addition of the HOLD column to the output:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
OWNER  BATCH_NAME        SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS
wenger CMD: /bin/slee   9/13 16:25      _      3       _      1      4 599.0 ...

4 jobs; 0 completed, 0 removed, 0 idle, 3 running, 1 held, 0 suspended
```

Here are some examples with a nested-DAG workflow in the queue, which is one of the most complicated cases. The workflow consists of a top-level DAG with nodes NodeA and NodeB, each with two two-proc clusters; and a sub-DAG SubZ with nodes NodeSA and NodeSB, each with two two-proc clusters.

First of all, non-batch mode with all of the node jobs in the queue:

```
$ condor_q -nobatch

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger          9/13 16:05   0+00:00:13 R  0    2.4 condor_dagman -p 0
 592.0   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 60
 592.1   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 300
 593.0   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 60
 593.1   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 300
 594.0   wenger          9/13 16:05   0+00:00:07 R  0    2.4 condor_dagman -p 0
 595.0   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 60
 595.1   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 300
 596.0   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 60
 596.1   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 300

10 jobs; 0 completed, 0 removed, 0 idle, 10 running, 0 held, 0 suspended
```

Now non-batch mode with the **-dag** option (unfortunately, *condor_q* doesn't do a good job of grouping procs in the same cluster together):

```
$ condor_q -nobatch -dag

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER/NODENAME      SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger             9/13 16:05   0+00:00:27 R  0    2.4 condor_dagman -
 592.0    |-NodeA           9/13 16:05   0+00:00:21 R  0    0.0 sleep 60
 593.0    |-NodeB           9/13 16:05   0+00:00:21 R  0    0.0 sleep 60
 594.0    |-SubZ            9/13 16:05   0+00:00:21 R  0    2.4 condor_dagman -
 595.0     |-NodeSA         9/13 16:05   0+00:00:15 R  0    0.0 sleep 60
 596.0     |-NodeSB         9/13 16:05   0+00:00:15 R  0    0.0 sleep 60
 592.1    |-NodeA           9/13 16:05   0+00:00:21 R  0    0.0 sleep 300
 593.1    |-NodeB           9/13 16:05   0+00:00:21 R  0    0.0 sleep 300
 595.1     |-NodeSA         9/13 16:05   0+00:00:15 R  0    0.0 sleep 300
 596.1     |-NodeSB         9/13 16:05   0+00:00:15 R  0    0.0 sleep 300

10 jobs; 0 completed, 0 removed, 0 idle, 10 running, 0 held, 0 suspended
```

Now, finally, the non-batch (default) mode:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
OWNER  BATCH_NAME     SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
wenger ex1.dag+591   9/13 16:05      _      8      _      5 592.0 ... 596.1

10 jobs; 0 completed, 0 removed, 0 idle, 10 running, 0 held, 0 suspended
```

There are several things about this output that may be slightly confusing:

- The TOTAL column is less than the RUN column. This is because, for DAG node jobs, their contribution to the TOTAL column is the number of clusters, not the number of procs (but their contribution to the RUN column is the number of procs). So the four DAG nodes (8 procs) contribute 4, and the sub-DAG contributes 1, to the TOTAL column. (But, somewhat confusingly, the sub-DAG job is not counted in the RUN column.)

- The sum of the RUN and IDLE columns (8) is less than the 10 jobs listed in the totals line at the bottom. This is because the top-level DAG and sub-DAG jobs are not counted in the RUN column, but they are counted in the totals line.

Now here is non-batch mode after proc 0 of each node job has finished:

```
$ condor_q -nobatch

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger          9/13 16:05   0+00:01:19 R  0    2.4 condor_dagman -p 0
 592.1   wenger          9/13 16:05   0+00:01:13 R  0    0.0 sleep 300
 593.1   wenger          9/13 16:05   0+00:01:13 R  0    0.0 sleep 300
 594.0   wenger          9/13 16:05   0+00:01:13 R  0    2.4 condor_dagman -p 0
 595.1   wenger          9/13 16:05   0+00:01:07 R  0    0.0 sleep 300
 596.1   wenger          9/13 16:05   0+00:01:07 R  0    0.0 sleep 300

6 jobs; 0 completed, 0 removed, 0 idle, 6 running, 0 held, 0 suspended
```

The same state also with the **-dag** option:

```
$ condor_q -nobatch -dag

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER/NODENAME      SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger             9/13 16:05   0+00:01:30 R  0    2.4 condor_dagman -
 592.1    |-NodeA           9/13 16:05   0+00:01:24 R  0    0.0 sleep 300
 593.1    |-NodeB           9/13 16:05   0+00:01:24 R  0    0.0 sleep 300
 594.0    |-SubZ            9/13 16:05   0+00:01:24 R  0    2.4 condor_dagman -
 595.1     |-NodeSA         9/13 16:05   0+00:01:18 R  0    0.0 sleep 300
 596.1     |-NodeSB         9/13 16:05   0+00:01:18 R  0    0.0 sleep 300

6 jobs; 0 completed, 0 removed, 0 idle, 6 running, 0 held, 0 suspended
```

And, finally, that state in batch (default) mode:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
OWNER  BATCH_NAME     SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
wenger ex1.dag+591   9/13 16:05      _      4       _      5 592.1 ... 596.1

6 jobs; 0 completed, 0 removed, 0 idle, 6 running, 0 held, 0 suspended
```

15.33.9 Exit Status

condor_q will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.34 *condor_qedit*

modify job attributes

15.34.1 Synopsis

condor_qedit [**-debug**] [**-n** *schedd-name*] [**-pool** *pool-name*] [**-forward**] *{cluster | cluster.proc | owner | -constraint constraint} edit-list*

15.34.2 Description

condor_qedit modifies job ClassAd attributes of queued HTCondor jobs. The jobs are specified either by cluster number, job ID, owner, or by a ClassAd constraint expression. The *edit-list* can take one of 3 forms

- ***attribute-name attribute-value ...***
  This is the older form, which behaves the same as the format below.
- ***attribute-name=attribute-value ...***
  The *attribute-value* may be any ClassAd expression. String expressions must be surrounded by double quotes. Multiple attribute value pairs may be listed on the same command line.
- **-edits[:auto|long|xml|json|new]** ***file-name***
  The file indicated by *file-name* is read as a classad of the given format. If no format is specified or `auto` is specified the format will be detected. if *file-name* is `-` standard input will be read.

To ensure security and correctness, *condor_qedit* will not allow modification of the following ClassAd attributes:

- `Owner`
- `ClusterId`
- `ProcId`
- `MyType`
- `TargetType`
- `JobStatus`

Since `JobStatus` may not be changed with *condor_qedit*, use *condor_hold* to place a job in the hold state, and use *condor_release* to release a held job, instead of attempting to modify `JobStatus` directly.

If a job is currently running, modified attributes for that job will not affect the job until it restarts. As an example, for `PeriodicRemove` to affect when a currently running job will be removed from the queue, that job must first be evicted from a machine and returned to the queue. The same is true for other periodic expressions, such as `PeriodicHold` and `PeriodicRelease`.

condor_qedit validates both attribute names and attribute values, checking for correct ClassAd syntax. An error message is printed, and no attribute is set or changed if any name or value is invalid.

15.34.3 Options

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-n *schedd-name*
Modify job attributes in the queue of the specified schedd

-pool *pool-name*
Modify job attributes in the queue of the schedd specified in the specified pool

-forward
Forward modifications to shadow/gridmanager

15.34.4 Examples

```
$ condor_qedit -name north.cs.wisc.edu -pool condor.cs.wisc.edu 249.0 answer 42
Set attribute "answer".
$ condor_qedit -name perdita 1849.0 In '"myinput"'
Set attribute "In".
% condor_qedit jbasney OnExitRemove=FALSE
Set attribute "OnExitRemove".
% condor_qedit -constraint 'JobUniverse == 1'␣
↪'Requirements=(Arch == "INTEL") && (OpSys == "SOLARIS26") && (Disk >= ExecutableSize) && (VirtualMemo
Set attribute "Requirements".
```

15.34.5 General Remarks

A job's ClassAd attributes may be viewed with

```
$ condor_q -long
```

15.34.6 Exit Status

condor_qedit will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.35 *condor_qusers*

add, enable and disable or show Users in the AP

15.35.1 Synopsis

condor_qusers [**-help**] [**-version**] [**-debug**] [**-name** *schedd-name*] [**-pool** *pool-name*] [**-long** | **--af** *{attrs}* | **-format** *fmt attr*] [**-add** | **-enable** | **--disable** [**-reason** *reason-string*]] *{users}*

15.35.2 Description

condor_qusers adds, enables or eisables or shows User records in the AP. Which user records are specified by name. The tool will do only one of these things at a time. It will print user records if no add, enable, or disable option is chosen.

15.35.3 Options

-help
Print useage then exit.

-version
Print the version and then exit.

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-name ***schedd-name***
Modify job attributes in the queue of the specified schedd

-pool ***pool-name***
Modify job attributes in the queue of the schedd specified in the specified pool

-long
Print User ClassAds in long form

-format ***fmt attr***
Print selected attribute of the User ClassAds using the given format.

-autoformat[:lhVr,tng] ***attr1 [attr2 . . .]*** **or -af[:lhVr,tng]** ***attr1 [attr2 . . .]***
(output option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings. This output option does not work in conjunction with any of the options **-add**, **-enable**, or **-disable**

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may not be used together. The **l** and **h** characters may not be used together.

-add ***user1 [user2 . . .]***
Add User ClassAds for the given users.

-enable ***user1 [user2 . . .]***
Enable the given users, adding them if necessary.

-disable ***user1 [user2 . . .]***
Disable the given users. Disabled users cannot submit jobs.

-reason ***reason-string***
Provide a reason for disabling when used with **-disable**. The disable reason will be included in the error message when submit fails because a user is disabled.

15.35.4 Examples

```
$ condor_qusers -name north.cs.wisc.edu -pool condor.cs.wisc.edu
Print users from north.cs.wisc.edu in the condor.cs.wisc.edu pool
$ condor_qusers -name perdita
Print users from perdition in the local pool
% condor_qusers -add bob
Add user bob to the local AP
% condor_qusers -disable -bob -reason "talk to admin"
Disable user bob with the reason "talk to admin"
```

15.35.5 General Remarks

An APs User ClassAds have attributes that count the number of jobs that user has in the queue, as well as enable/disable and the short and fully-qualified user name. The full set of attributes can can be viewed with

```
$ condor_qusers -long
```

15.35.6 Exit Status

condor_qusers will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.36 *condor_qsub*

Queue jobs that use PBS/SGE-style submission

15.36.1 Synopsis

condor_qsub [**–version**]

condor_qsub [**Specific options**] [**Directory options**] [**Environmental options**] [**File options**] [**Notification options**] [**Resource options**] [**Status options**] [**Submission options**] *commandfile*

15.36.2 Description

condor_qsub submits an HTCondor job. This job is specified in a PBS/Torque style or an SGE style. *condor_qsub* permits the submission of dependent jobs without the need to specify the full dependency graph at submission time. Doing things this way is neither as efficient as HTCondor's DAGMan, nor as functional as SGE's *qsub* or *qalter*. *condor_qsub* serves as a minimal translator to be able to use software originally written to interact with PBS, Torque, and SGE in an HTCondor pool.

condor_qsub attempts to behave like *qsub*. Less than half of the *qsub* functionality is implemented. Option descriptions describe the differences between the behavior of *qsub* and *condor_qsub*. *qsub* options not listed here are not supported. Some concepts present in PBS and SGE do not apply to HTCondor, and so these options are not implemented.

For a full listing of *qsub* options, please see

POSIX
: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/qsub.html

SGE
: http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html

PBS/Torque
: http://docs.adaptivecomputing.com/torque/4-1-3/Content/topics/commands/qsub.htm

condor_qsub accepts either command line options or the single file, *commandfile*, that contains all of the commands.

condor_qsub does the opposite of job submission within the **grid** universe **batch** grid type, which takes HTCondor jobs submitted with HTCondor syntax and submits them to PBS, SGE, or LSF.

15.36.3 Options

-a *date_time*
(Submission option) Specify a deferred execution date and time. The PBS/Torque syntax of *date_time* is a string in the form *[[[[CC]YY]MM]DD]hhmm[.SS]*. The portions of this string which are optional are *CC*, *YY*, *MM*, *DD*, and *SS*. For SGE, *MM* and *DD* are not optional. For PBS, *MM* and *DD* are optional. *condor_qsub* follows the PBS style.

-A *account_string*
(Status option) Uses group accounting where the string *account_string* is the accounting group associated with this job. Unlike SGE, there is no default group of `"sge"`.

-b *y|n*
(Submission option) Using the SGE definition of its *-b* option, a value of *y* causes *condor_qsub* to not parse the file for additional *condor_qsub* commands. The default value is *n*. If the command line argument **-f** *filename* is also specified, it negates a value of *y*.

-condor-keep-files
(Specific option) Directs HTCondor to not remove temporary files generated by *condor_qsub*, such as HTCondor submit files and sentinel jobs. These temporary files may be important for debugging.

-cwd
(Directory option) Specifies the initial directory in which the job will run to be the current directory from which the job was submitted. This sets **initialdir** for *condor_submit*.

-d *path* or -wd *path*
(Directory option) Specifies the initial directory in which the job will run to be *path*. This sets **initialdir** for *condor_submit*.

-e *filename*
(File option) Specifies the *condor_submit* command **error** , the file where `stderr` is written. If not specified, set to the default name of `` <commandfile>.e<ClusterId>``, where <`commandfile`> is the *condor_qsub* argument, and `` <ClusterId>`` is the job attribute `ClusterId` assigned for the job.

-f *qsub_file*
(Specific option) Parse *qsub_file* to search for and set additional *condor_submit* commands. Within the file, commands will appear as `#PBS` or `#SGE`. *condor_qsub* will parse the batch file listed as *qsub_file*.

-h
(Status option) Placed submitted job directly into the hold state.

-help
(Specific option) Print usage information and exit.

-hold_jid <*jid*>
(Status option) Submits a job in the hold state. This job is released only when a previously submitted job, identified by its cluster ID as <*jid*>, exits successfully. Successful completion is defined as not exiting with exit code 100. In implementation, there are three jobs that define this SGE feature. The first job is the previously submitted job. The second job is the newly submitted one that is waiting for the first to finish successfully. The third job is what SGE calls a sentinel job; this is an HTCondor local universe job that watches the history for the first job's exit code. This third job will exit once it has seen the exit code and, for a successful termination of the first job, run *condor_release* on the second job. If the first job is an array job, the second job will only be released after all individual jobs of the first job have completed.

-i *[hostname:]filename*
(File option) Specifies the *condor_submit* command **input** , the file from which `stdin` is read.

-j *characters*
(File option) Acceptable characters for this option are `e`, `o`, and `n`. The only sequence that is relevant is `eo`; it specifies that both standard output and standard error are to be sent to the same file. The file will be the one specified by the **-o** option, if both the **-o** and **-e** options exist. The file will be the one specified by the **-e** option, if only the **-e** option is provided. If neither the **-o** nor the **-e** options are provided, the file will be the default used for the **-o** option.

-l *resource_spec*
(Resource option) Specifies requirements for the job, such as the amount of RAM and the number of CPUs. Only PBS-style resource requests are supported. *resource_spec* is a comma separated list of key/value pairs. Each pair is of the form `resource_name=value`. `resource_name` and `value` may be +——————+——————+——————+ | `resource_name` | `value` | Description | +——————+——————+——————+ | arch | string | Sets `Arch` machine | | | attribute. Enclose in | | | double quotes. | +——————+——————+——————+ | file | size | Disk space requested. | +——————+——————+——————+ | host | string | Host machine on which | | | the job must run. |

+——————————+——————————+——————————+ | mem | size | Amount of memory | | | | requested. | +——————————+——————————+——————————+ | nodes | {<node_count> | <hostn | Number and/or properties | | | ame>} [:ppn=<ppn>] [:gpu | of nodes to be used. For | | | s=<gpu>] [: <property> [: | examples, please see | | | <property>] ...] [+ ...] | http://docs.adaptivecom | | | | puting.com/torque/4-1-3/ | | | | Content/topics/2-jobs/re | | | | questingRes.htm#qsub | +——————————+——————————+——————————+ | opsys | string | Sets OpSys machine | | | attribute. Enclose in | | | double quotes. | +——————————+——————————+——————————+ | procs | integer | Number of CPUs | | | requested. | +——————————+——————————+——————————+

A size value is an integer specified in bytes, following the PBS/Torque default. Append Kb, Mb, Gb, or Tb to specify the value in powers of two quantities greater than bytes.

-m ***a|e|n***
(Notification option) Identify when HTCondor sends notification e-mail. If *a*, send e-mail when the job terminates abnormally. If *e*, send e-mail when the job terminates. If *n*, never send e-mail.

-M ***e-mail_address***
(Notification option) Sets the destination address for HTCondor e-mail.

-o ***filename***
(File option) Specifies the *condor_submit* command **output** , the file where stdout is written. If not specified, set to the default name of `` <commandfile>.o<ClusterId>``, where <commandfile> is the *condor_qsub* argument, and `` <ClusterId>`` is the job attribute ClusterId assigned for the job.

-p ***integer***
(Status option) Sets the **priority** submit command for the job, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority.

-print
(Specific option) Send to stdout the contents of the HTCondor submit description file that *condor_qsub* generates.

-r ***y|n***
(Status option) The default value of *y* implements the default HTCondor policy of assuming that jobs that do not complete are placed back in the queue to be run again. When *n*, job submission is restricted to only running the job if the job ClassAd attribute NumJobStarts is currently 0. This identifies the job as not re-runnable, limiting it to start once.

-S ***shell***
(Submission option) Specifies the path and executable name of a shell. Alters the HTCondor submit description file produced, such that the executable becomes a wrapper script. Within the submit description file will be executable = <shell> and arguments = <commandfile>.

-t ***start [-stop:step]***
(Submission option) Queues a set of nearly identical jobs. The SGE-style syntax is supported. *start*, *stop*, and *step* are all integers. *start* is the starting index of the jobs, *stop* is the ending index (inclusive) of the jobs, and *step* is the step size through the indices. Note that using more than one processor or node in a job will not work with this option.

-test
(Specific option) With the intention of testing a potential job submission, parse files and commands to generate error output. Produces, but then removes the HTCondor submit description file. Never submits the job, even if no errors are encountered.

-v ***variable list***
(Environmental option) Used to set the submit command **environment** for the job. *variable list* is as

that defined for the submit command. Note that the syntax needed is specialized to deal with quote marks and white space characters.

-V
(Environmental option) Sets `getenv = True` in the submit description file.

-W *attr_name=attr_value[,attr_name=attr_value...]*
(File option) PBS/Torque supports a number of attributes. However, *condor_qsub* only supports the names *stagein* and *stageout* for *attr_name*. The format of *attr_value* for *stagein* and *stageout* is `local_file@hostname:remote_file[,...]` and we strip it to `remote_file[,...]`. HTCondor's file transfer mechanism is then used if needed.

-version
(Specific option) Print version information for the *condor_qsub* program and exit. Note that *condor_qsub* has its own version numbers which are separate from those of HTCondor.

15.36.4 Exit Status

condor_qsub will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure to submit a job.

15.37 *condor_reconfig*

Reconfigure HTCondor daemons

15.37.1 Synopsis

condor_reconfig [**-help | -version**]

condor_reconfig [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname*]

15.37.2 Description

condor_reconfig reconfigures all of the HTCondor daemons in accordance with the current status of the HTCondor configuration file(s). Once reconfiguration is complete, the daemons will behave according to the policies stated in the configuration file(s). The main exception is with the `DAEMON_LIST` variable, which will only be updated if the *condor_restart* command is used. Other configuration variables that can only be changed if the HTCondor daemons are restarted are listed in the HTCondor manual in the section on configuration. In general, *condor_reconfig* should be used when making changes to the configuration files, since it is faster and more efficient than restarting the daemons.

The command *condor_reconfig* with no arguments or with the **-daemon** *master* option will cause the reconfiguration of the *condor_master* daemon and all the child processes of the *condor_master*.

For security reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

15.37.3 Options

-help
Display usage information

-version
Display version information

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname***
Send the command to a machine identified by *hostname*

hostname
Send the command to a machine identified by *hostname*

-addr ***"<a.b.c.d:port>"***
Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>"
Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression***
Apply this command only to machines matching the given ClassAd *expression*

-all
Send the command to all machines in the pool

-daemon ***daemonname***
Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

15.37.4 Exit Status

condor_reconfig will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.37.5 Examples

To reconfigure the *condor_master* and all its children on the local host:

```
$ condor_reconfig
```

To reconfigure only the *condor_startd* on a named machine:

```
$ condor_reconfig -name bluejay -daemon startd
```

To reconfigure a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reconfigures the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
$ condor_reconfig -pool condor.cae.wisc.edu -name cae17
```

15.38 *condor_release*

release held jobs in the HTCondor queue

15.38.1 Synopsis

condor_release [**-help | -version**]

condor_release [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] *cluster...* | *cluster.process...* | *user...* | **-constraint** *expression ...*

condor_release [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] **-all**

15.38.2 Description

condor_release releases jobs from the HTCondor job queue that were previously placed in hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be released are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can release the job.

15.38.3 Options

-help
: Display usage information

-version
: Display version information

-pool ***centralmanagerhostname[:portnumber]***
: Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname***
: Send the command to a machine identified by *scheddname*

-addr ***"<a.b.c.d:port>"***
: Send the command to a machine located at *"<a.b.c.d:port>"*

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

cluster
: Release all jobs in the specified cluster

cluster.process
: Release the specific job in the cluster

user
: Release jobs belonging to specified user

-constraint *expression*
Release all jobs which match the job ClassAd expression constraint

-all
Release all the jobs in the queue

15.38.4 See Also

condor_hold

15.38.5 Examples

To release all of the jobs of a user named Mary:

```
$ condor_release Mary
```

15.38.6 Exit Status

condor_release will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.39 *condor_remote_cluster*

Manage and configure the clusters to be accessed.

15.39.1 Synopsis

condor_remote_cluster [**-h** || **–help**]

condor_remote_cluster [**-l** || **–list**] [**-a** || **–add <host> [schedd]**] [**-r** || **–remove <host>**] [**-s** || **–status <host>**] [**-t** || **–test <host>**]

15.39.2 Description

condor_remote_cluster is part of a feature for accessing high throughput computing resources from a local desktop using only an SSH connection.

condor_remote_cluster enables management and configuration of the access point of the remote computing resource. After initial setup, jobs can be submitted to the local job queue, which are then forwarded to the remote system.

A **<host>** is of the form `user@fqdn.example.com`.

15.39.3 Options

-help
Print usage information and exit.

-list
List all installed clusters.

-remove <*host*>
Remove an already installed cluster, where the cluster is identified by <*host*>.

-add <*host*> *[scheduler]*
Install and add a cluster defined by <*host*>. The optional *scheduler* specifies the scheduler on the cluster. Valid values are `pbs`, `lsf`, `condor`, `sge` or `slurm`. If not given, the default will be `pbs`.

-status <*host*>
Query and print the status of an already installed cluster, where the cluster is identified by <*host*>.

-test <*host*>
Attempt to submit a test job to an already installed cluster, where the cluster is identified by <*host*>.

15.40 *condor_reschedule*

Update scheduling information to the central manager

15.40.1 Synopsis

condor_reschedule [**-help | -version**]

condor_reschedule [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

15.40.2 Description

condor_reschedule updates the information about a set of machines' resources and jobs to the central manager. This command is used to force an update before viewing the current status of a machine. Viewing the status of a machine is done with the *condor_status* command. *condor_reschedule* also starts a new negotiation cycle between resource owners and resource providers on the central managers, so that jobs can be matched with machines right away. This can be useful in situations where the time between negotiation cycles is somewhat long, and an administrator wants to see if a job in the queue will get matched without waiting for the next negotiation cycle.

A new negotiation cycle cannot occur more frequently than every 20 seconds. Requests for new negotiation cycle within that 20 second window will be deferred until 20 seconds have passed since that last cycle.

15.40.3 Options

-help
Display usage information

-version
Display version information

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname***
Send the command to a machine identified by *hostname*

hostname
Send the command to a machine identified by *hostname*

-addr ***"<a.b.c.d:port>"***
Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>"
Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression***
Apply this command only to machines matching the given ClassAd *expression*

-all
Send the command to all machines in the pool

15.40.4 Exit Status

condor_reschedule will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.40.5 Examples

To update the information on three named machines:

```
$ condor_reschedule robin cardinal bluejay
```

To reschedule on a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reschedules the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
$ condor_reschedule -pool condor.cae.wisc.edu -name cae17
```

15.41 *condor_restart*

Restart a set of HTCondor daemons

15.41.1 Synopsis

condor_restart [**-help | -version**]

condor_restart [**-debug[:opts]**]] [**-graceful | -fast | -peaceful | -drain**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname* | **-master**] [**-exec** *name*] [**-reason** *"reason-string"*] [**-request-id** *id*] [**-check** *expr*] [**-start** *expr*]

15.41.2 Description

condor_restart restarts a set of HTCondor daemons on a set of machines. The daemons will be put into a consistent state, killed, and then invoked anew.

If, for example, the *condor_master* needs to be restarted again with a fresh state, this is the command that should be used to do so. If the `DAEMON_LIST` variable in the configuration file has been changed, this command is used to restart the *condor_master* in order to see this change. The *condor_reconfigure* command cannot be used in the case where the `DAEMON_LIST` expression changes.

The command *condor_restart* with no arguments or with the **-daemon** *master* option will safely shut down all running jobs and all submitted jobs from the machine(s) being restarted, then shut down all the child daemons of the *condor_master*, and then restart the *condor_master*. This, in turn, will allow the *condor_master* to start up other daemons as specified in the `DAEMON_LIST` configuration file entry.

When restarting down all daemons including the *condor_master*, the **-exec** argument can be used to tell the master to run a configured script before it restarts.

When the **-drain** option is chosen, draining options can be specified by using the optional **-reason**, **-request-id**, **-check**, and **-start** arguments.

For security reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

15.41.3 Options

-help
: Display usage information

-version
: Display version information

-debug[:opts]
: Causes debugging information to be sent to `stderr`. The debug level can be set by specifying an optional *opts* value. Otherwise, the configuration variable `TOOL_DEBUG` sets the debug level.

-graceful
: Gracefully shutdown daemons (the default) before restarting them

-fast
: Quickly shutdown daemons before restarting them

-peaceful
: Wait indefinitely for jobs to finish before shutting down daemons, prior to restarting them

-drain
Send a *condor_drain* command with the *-exit-on-completion* option to all *condor_startd* daemons that are managed by this master. Then wait for all *condor_startd* daemons to exit before before restarting.

-reason *"reason-string"*
Use with **-drain** to set a **-reason** *"reason-string"* value for the *condor_drain* command.

-request-id *id*
Use with **-drain** to set a **-request-id** *id* value for the *condor_drain* command.

-check *expr*
Use with **-drain** to set a **-check** *expr* value for the *condor_drain* command.

-start *expr*
Use with **-drain** to set a **-start** *expr* value for the *condor_drain* command.

-pool *centralmanagerhostname[:portnumber]*
Specify a pool by giving the central manager's host name and an optional port number

-name *hostname*
Send the command to a machine identified by *hostname*

hostname
Send the command to a machine identified by *hostname*

-addr *"<a.b.c.d:port>"*
Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>"
Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint *expression*
Apply this command only to machines matching the given ClassAd *expression*

-all
Send the command to all machines in the pool

-master
Restart the *condor_master* after shutting down all other daemons. This will have the effect of restarting all of the daemons.

-exec *name*
When used with **-master**, the *condor_master* will run the program configured as after shutting down all other daemons.

-daemon *daemonname*
Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

15.41.4 Exit Status

condor_restart will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.41.5 Examples

To restart the *condor_master* and all its children on the local host:

```
$ condor_restart
```

To restart only the *condor_startd* on a named machine:

```
$ condor_restart -name bluejay -daemon startd
```

To restart a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command restarts the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
$ condor_restart -pool condor.cae.wisc.edu -name cae17
```

15.42 *condor_rm*

remove jobs from the HTCondor queue

15.42.1 Synopsis

condor_rm [**-help | -version**]

condor_rm [**-debug**] [**-forcex**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] *cluster...* | *cluster.process...* | *user...* | **-constraint** *expression ...*

condor_rm [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] **-all**

15.42.2 Description

condor_rm removes one or more jobs from the HTCondor job queue. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be removed are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can remove the job.

When removing a grid job, the job may remain in the "X" state for a very long time. This is normal, as HTCondor is attempting to communicate with the remote scheduling system, ensuring that the job has been properly cleaned up. If it takes too long, or in rare circumstances is never removed, the job may be forced to leave the job queue by using the **-forcex** option. This forcibly removes jobs that are in the "X" state without attempting to finish any clean up at the remote scheduler.

15.42.3 Options

-help
Display usage information

-version
Display version information

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname***
Send the command to a machine identified by *scheddname*

-addr ***"<a.b.c.d:port>"***
Send the command to a machine located at *"<a.b.c.d:port>"*

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-forcex
Force the immediate local removal of jobs in the 'X' state (only affects jobs already being removed)

cluster
Remove all jobs in the specified cluster

cluster.process
Remove the specific job in the cluster

user
Remove jobs belonging to specified user

-constraint ***expression***
Remove all jobs which match the job ClassAd expression constraint

-all
Remove all the jobs in the queue

15.42.4 General Remarks

Use the *-forcex* argument with caution, as it will remove jobs from the local queue immediately, but can orphan parts of the job that are running remotely and have not yet been stopped or removed.

15.42.5 Examples

For a user to remove all their jobs that are not currently running:

```
$ condor_rm -constraint 'JobStatus =!= 2'
```

15.42.6 Exit Status

condor_rm will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.43 *condor_rmdir*

Windows-only no-fail deletion of directories

15.43.1 Synopsis

condor_rmdir [**/HELP | /?**]

condor_rmdir *@filename*

condor_rmdir [**/VERBOSE**] [**/DIAGNOSTIC**] [**/PATH:<path>**] [**/S**] [**/C**] [**/Q**] [**/NODEL**] *directory*

15.43.2 Description

condor_rmdir can delete a specified *directory*, and will not fail if the directory contains files that have ACLs that deny the SYSTEM process delete access, unlike the built-in Windows *rmdir* command.

The directory to be removed together with other command line arguments may be specified within a file named *filename*, prefixing this argument with an `@` character.

The *condor_rmdir.exe* executable is is intended to be used by HTCondor with the **/S /C** options, which cause it to recurse into subdirectories and continue on errors.

15.43.3 Options

/HELP
: Print usage information.

/?
: Print usage information.

/VERBOSE
: Print detailed output.

/DIAGNOSTIC
: Print out the internal flow of control information.

/PATH:<path>
: Remove the directory given by **<path>**.

/S
: Include subdirectories in those removed.

/C
: Continue even if access is denied.

/Q
: Print error output only.

/NODEL
: Do not remove directories. ACLs may still be changed.

15.43.4 Exit Status

condor_rmdir will exit with a status value of 0 (zero) upon success, and it will exit with the standard HRESULT error code upon failure.

15.44 *condor_router_history*

Display the history for routed jobs

15.44.1 Synopsis

condor_router_history [–**h**]

condor_router_history [–**show_records**] [–**show_iwd**] [–**age** *days*] [–**days** *days*] [–**start** *"YYYY-MM-DD HH:MM"*]

15.44.2 Description

condor_router_history summarizes statistics for routed jobs over the previous 24 hours. With no command line options, statistics for run time, number of jobs completed, and number of jobs aborted are listed per route (site).

15.44.3 Options

-h
: Display usage information and exit.

-show_records
: Displays individual records in addition to the summary.

-show_iwd
: Include working directory in displayed records.

-age ***days***
: Set the ending time of the summary to be *days* days ago.

-days ***days***
: Set the number of days to summarize.

-start ***"YYYY-MM-DD HH:MM"***
: Set the start time of the summary.

15.44.4 Exit Status

condor_router_history will exit with a status of 0 (zero) upon success, and non-zero otherwise.

15.45 *condor_router_q*

Display information about routed jobs in the queue

15.45.1 Synopsis

condor_router_q [**-S**] [**-R**] [**-I**] [**-H**] [**-route** *name*] [**-idle**] [**-held**] [**-constraint** *X*] [**condor_q options**]

15.45.2 Description

condor_router_q displays information about jobs managed by the *condor_job_router* that are in the HTCondor job queue. The functionality of this tool is that of *condor_q*, with additional options specialized for routed jobs. Therefore, any of the options for *condor_q* may also be used with *condor_router_q*.

15.45.3 Options

-S
: Summarize the state of the jobs on each route.

-R
: Summarize the running jobs on each route.

-I
: Summarize the idle jobs on each route.

-H
: Summarize the held jobs on each route.

-route *name*
: Display only the jobs on the route identified by *name*.

-idle
: Display only the idle jobs.

-held
: Display only the held jobs.

-constraint *X*
: Display only the jobs matching constraint *X*.

15.45.4 Exit Status

condor_router_q will exit with a status of 0 (zero) upon success, and non-zero otherwise.

15.46 *condor_router_rm*

Remove jobs being managed by the HTCondor Job Router

15.46.1 Synopsis

condor_router_rm [*router_rm options*] [*condor_rm options*]

15.46.2 Description

condor_router_rm is a script that provides additional features above those offered by *condor_rm*, for removing jobs being managed by the HTCondor Job Router.

The options that may be supplied to *condor_router_rm* belong to two groups:

- **router_rm options** provide the additional features
- **condor_rm options** are those options already offered by *condor_rm*. See the *condor_rm* manual page for specification of these options.

15.46.3 Options

-constraint *X*
(router_rm option) Remove jobs matching the constraint specified by *X*

-held
(router_rm option) Remove only jobs in the hold state

-idle
(router_rm option) Remove only idle jobs

-route *name*
(router_rm option) Remove only jobs on specified route

15.46.4 Exit Status

condor_router_rm will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.47 *condor_run*

Submit a shell command-line as an HTCondor job

15.47.1 Synopsis

condor_run [**-u** *universe*] [**-a** *submitcmd*] *"shell command"*

15.47.2 Description

condor_run bundles a shell command line into an HTCondor job and submits the job. The *condor_run* command waits for the HTCondor job to complete, writes the job's output to the terminal, and exits with the exit status of the HTCondor job. No output appears until the job completes.

Enclose the shell command line in double quote marks, so it may be passed to *condor_run* without modification. *condor_run* will not read input from the terminal while the job executes. If the shell command line requires input, redirect the input from a file, as illustrated by the example

```
$ condor_run "myprog < input.data"
```

condor_run jobs rely on a shared file system for access to any necessary input files. The current working directory of the job must be accessible to the machine within the HTCondor pool where the job runs.

Specialized environment variables may be used to specify requirements for the machine where the job may run.

CONDOR_ARCH
: Specifies the architecture of the required platform. Values will be the same as the `Arch` machine ClassAd attribute.

CONDOR_OPSYS
: Specifies the operating system of the required platform. Values will be the same as the `OpSys` machine ClassAd attribute.

CONDOR_REQUIREMENTS
: Specifies any additional requirements for the HTCondor job. It is recommended that the value defined for `CONDOR_REQUIREMENTS` be enclosed in parenthesis.

When one or more of these environment variables is specified, the job is submitted with:

```
Requirements = $CONDOR_REQUIREMENTS && Arch == $CONDOR_ARCH && OpSys == $CONDOR_OPSYS
```

Without these environment variables, the job receives the default requirements expression, which requests a machine of the same platform as the machine on which *condor_run* is executed.

All environment variables set when *condor_run* is executed will be included in the environment of the HTCondor job.

condor_run removes the HTCondor job from the queue and deletes its temporary files, if *condor_run* is killed before the HTCondor job completes.

15.47.3 Options

-u *universe*
: Submit the job under the specified universe. The default is vanilla. While any universe may be specified, only the vanilla, scheduler, and local universes result in a submit description file that may work properly.

-a *submitcmd*
: Add the specified submit command to the implied submit description file for the job. To include spaces within *submitcmd*, enclose the submit command in double quote marks. And, to include double quote marks within *submitcmd*, enclose the submit command in single quote marks.

15.47.4 Examples

condor_run may be used to compile an executable on a different platform. As an example, first set the environment variables for the required platform:

```
$ export CONDOR_ARCH="SUN4u"
$ export CONDOR_OPSYS="SOLARIS28"
```

Then, use *condor_run* to submit the compilation as in the following two examples.

```
$ condor_run "f77 -O -o myprog myprog.f"
```

or

```
$ condor_run "make"
```

15.47.5 Files

condor_run creates the following temporary files in the user's working directory. The placeholder <pid> is replaced by the process id of *condor_run*.

.condor_run.<pid>
A shell script containing the shell command line.

.condor_submit.<pid>
The submit description file for the job.

.condor_log.<pid>
The HTCondor job's log file; it is monitored by *condor_run*, to determine when the job exits.

.condor_out.<pid>
The output of the HTCondor job before it is output to the terminal.

.condor_error.<pid>
Any error messages for the HTCondor job before they are output to the terminal.

condor_run removes these files when the job completes. However, if *condor_run* fails, it is possible that these files will remain in the user's working directory, and the HTCondor job may remain in the queue.

15.47.6 General Remarks

condor_run is intended for submitting simple shell command lines to HTCondor. It does not provide the full functionality of *condor_submit*. Therefore, some *condor_submit* errors and system failures may not be handled correctly.

All processes specified within the single shell command line will be executed on the single machine matched with the job. HTCondor will not distribute multiple processes of a command line pipe across multiple machines.

condor_run will use the shell specified in the `SHELL` environment variable, if one exists. Otherwise, it will use */bin/sh* to execute the shell command-line.

By default, *condor_run* expects Perl to be installed in `/usr/bin/perl`. If Perl is installed in another path, ask the Condor administrator to edit the path in the *condor_run* script, or explicitly call Perl from the command line:

```
$ perl path-to-condor/bin/condor_run "shell-cmd"
```

15.47.7 Exit Status

condor_run exits with a status value of 0 (zero) upon complete success. The exit status of *condor_run* will be non-zero upon failure. The exit status in the case of a single error due to a system call will be the error number (`errno`) of the failed call.

15.48 *condor_set_shutdown*

Set a program to execute upon *condor_master* shut down

15.48.1 Synopsis

condor_set_shutdown [**-help | -version**]

condor_set_shutdown -exec *programname* [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

15.48.2 Description

condor_set_shutdown sets a program (typically a script) to execute when the *condor_master* daemon shuts down. The **-exec** *programname* argument is required, and specifies the program to run. The string *programname* must match the string that defines `Name` in the configuration variable `MASTER_SHUTDOWN_<Name>` in the *condor_master* daemon's configuration. If it does not match, the *condor_master* will log an error and ignore the request.

For security reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

15.48.3 Options

-help
: Display usage information

-version
: Display version information

-exec ***name***
: Select the program the master should exec the next time it shuts down. The master will run the program configured as `MASTER_SHUTDOWN_<name>` from the configuration of the *condor_master*.

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-pool ***centralmanagerhostname[:portnumber]***
: Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname***
: Send the command to a machine identified by *hostname*

hostname
: Send the command to a machine identified by *hostname*

-addr ***"<a.b.c.d:port>"***
: Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>"
Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint ***expression***
Apply this command only to machines matching the given ClassAd *expression*

-all
Send the command to all machines in the pool

15.48.4 Exit Status

condor_set_shutdown will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.48.5 Examples

To have all *condor_master* daemons run the program */bin/reboot* upon shut down, configure the *condor_master* to contain a definition similar to:

```
MASTER_SHUTDOWN_REBOOT = /sbin/reboot
```

where REBOOT is an invented name for this program that the *condor_master* will execute. On the command line, run

```
$ condor_set_shutdown -exec reboot -all
$ condor_off -graceful -all
```

where the string reboot matches the invented name.

15.49 *condor_sos*

Issue a command that will be serviced with a higher priority

15.49.1 Synopsis

condor_sos [**-help | -version**]

condor_sos [**-debug**] [**-timeoutmult** *value*] *condor_command*

15.49.2 Description

condor_sos sends the *condor_command* in such a way that the command is serviced ahead of other waiting commands. It appears to have a higher priority than other waiting commands.

condor_sos is intended to give administrators a way to query the *condor_schedd* and *condor_collector* daemons when they are under such a heavy load that they are not responsive.

There must be a special command port configured, in order for a command to be serviced with priority. The *condor_schedd* and *condor_collector* always have the special command port. Other daemons require configuration by setting configuration variable <SUBSYS>_SUPER_ADDRESS_FILE.

15.49.3 Options

-help
Display usage information

-version
Display version information

-debug
Print extra debugging information as the command executes.

-timeoutmult ***value***
Multiply any timeouts set for the command by the integer *value*.

15.49.4 Examples

The example command

```
$ condor_sos -timeoutmult 5 condor_hold -all
```

causes the `condor_hold -all` command to be handled by the *condor_schedd* with priority over any other commands that the *condor_schedd* has waiting to be serviced. It also extends any set timeouts by a factor of 5.

15.49.5 Exit Status

condor_sos will exit with the value 1 on error and with the exit value of the invoked command when the command is successfully invoked.

15.50 *condor_ssh_start*

15.50.1 Synopsis

condor_ssh_start

15.50.2 Description

condor_ssh_start is part of a system for accessing high throughput computing resources from a local desktop.

This command is not meant to be executed on the command line by users.

15.51 *condor_ssh_to_job*

create an ssh session to a running job

15.51.1 Synopsis

condor_ssh_to_job [**-help**]

condor_ssh_to_job [**-debug**] [**-name** *schedd-name*] [**-pool** *pool-name*] [**-ssh** *ssh-command*] [**-keygen-options** *ssh-keygen-options*] [**-shells** *shell1,shell2,...*] [**-auto-retry**] [**-remove-on-interrupt**] *cluster* | *cluster.process* | *cluster.process.node* [*remote-command*]

15.51.2 Description

condor_ssh_to_job creates an *ssh* session to a running job. The job is specified with the argument. If only the job *cluster* id is given, then the job *process* id defaults to the value 0.

condor_ssh_to_job is available in Unix HTCondor distributions, and works with two kinds of jobs: those in the vanilla, vm, java, local, or parallel universes, and those jobs in the grid universe which use EC2 resources. It will not work with other grid universe jobs.

For jobs in the vanilla, vm, java, local, or parallel universes, the user must be the owner of the job or must be a queue super user, and both the *condor_schedd* and *condor_starter* daemons must allow *condor_ssh_to_job* access. If no *remote-command* is specified, an interactive shell is created. An alternate *ssh* program such as *sftp* may be specified, using the **-ssh** option, for uploading and downloading files.

The remote command or shell runs with the same user id as the running job, and it is initialized with the same working directory. The environment is initialized to be the same as that of the job, plus any changes made by the shell setup scripts and any environment variables passed by the *ssh* client. In addition, the environment variable `_CONDOR_JOB_PIDS` is defined. It is a space-separated list of PIDs associated with the job. At a minimum, the list will contain the PID of the process started when the job was launched, and it will be the first item in the list. It may contain additional PIDs of other processes that the job has created.

The *ssh* session and all processes it creates are treated by HTCondor as though they are processes belonging to the job. If the slot is preempted or suspended, the *ssh* session is killed or suspended along with the job. If the job exits before the *ssh* session finishes, the slot remains in the Claimed Busy state and is treated as though not all job processes have exited until all *ssh* sessions are closed. Multiple *ssh* sessions may be created to the same job at the same time. Resource consumption of the *sshd* process and all processes spawned by it are monitored by the *condor_starter* as though these processes belong to the job, so any policies such as `PREEMPT` that enforce a limit on resource consumption also take into account resources consumed by the *ssh* session.

condor_ssh_to_job stores ssh keys in temporary files within a newly created and uniquely named directory. The newly created directory will be within the directory defined by the environment variable `TMPDIR`. When the ssh session is finished, this directory and the ssh keys contained within it are removed.

See the HTCondor administrator's manual section on configuration for details of the configuration variables related to *condor_ssh_to_job*.

An *ssh* session works by first authenticating and authorizing a secure connection between *condor_ssh_to_job* and the *condor_starter* daemon, using HTCondor protocols. The *condor_starter* generates an ssh key pair and sends it securely to *condor_ssh_to_job*. Then the *condor_starter* spawns *sshd* in inetd mode with its stdin and stdout attached to the TCP connection from *condor_ssh_to_job*. *condor_ssh_to_job* acts as a proxy for the *ssh* client to communicate with *sshd*, using the existing connection authorized by HTCondor. At no point is *sshd* listening on the network for connections or running with any privileges other than that of the user identity running the job. If CCB is being used to enable connectivity to the execute node from outside of a firewall or private network, *condor_ssh_to_job* is able to make use of CCB in order to form the *ssh* connection.

The login shell of the user id running the job is used to run the requested command, *sshd* subsystem, or interactive shell. This is hard-coded behavior in *OpenSSH* and cannot be overridden by configuration. This means that *condor_ssh_to_job* access is effectively disabled if the login shell disables access, as in the example programs */bin/true* and */sbin/nologin*.

condor_ssh_to_job is intended to work with *OpenSSH* as installed in typical environments. It does not work on Windows platforms. If the *ssh* programs are installed in non-standard locations, then the paths to these programs will need to be customized within the HTCondor configuration. Versions of *ssh* other than *OpenSSH* may work, but they will likely require additional configuration of command-line arguments, changes to the *sshd* configuration template file, and possibly modification of the $(LIBEXEC)/condor_ssh_to_job_sshd_setup script used by the *condor_starter* to set up *sshd*.

For jobs in the grid universe which use EC2 resources, a request that HTCondor have the EC2 service create a new key pair for the job by specifying **ec2_keypair_file** causes *condor_ssh_to_job* to attempt to connect to the corresponding instance via *ssh*. This attempts invokes *ssh* directly, bypassing the HTCondor networking layer. It supplies *ssh* with the public DNS name of the instance and the name of the file with the new key pair's private key. For the connection to succeed, the instance must have started an *ssh* server, and its security group(s) must allow connections on port 22. Conventionally, images will allow logins using the key pair on a single specific account. Because *ssh* defaults to logging in as the current user, the **-l <username>** option or its equivalent for other versions of *ssh* will be needed as part of the *remote-command* argument. Although the **-X** option does not apply to EC2 jobs, adding **-X** or **-Y** to the *remote-command* argument can duplicate the effect.

15.51.3 Options

-help
Display brief usage information and exit.

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-name ***schedd-name***
Specify an alternate *condor_schedd*, if the default (local) one is not desired.

-pool ***pool-name***
Specify an alternate HTCondor pool, if the default one is not desired. Does not apply to EC2 jobs.

-ssh ***ssh-command***
Specify an alternate *ssh* program to run in place of *ssh*, for example *sftp* or *scp*. Additional arguments are specified as *ssh-command*. Since the arguments are delimited by spaces, place double quote marks around the whole command, to prevent the shell from splitting it into multiple arguments to *condor_ssh_to_job*. If any arguments must contain spaces, enclose them within single quotes. Does not apply to EC2 jobs.

-keygen-options ***ssh-keygen-options***
Specify additional arguments to the *ssh_keygen* program, for creating the ssh key that is used for the duration of the session. For example, a different number of bits could be used, or a different key type than the default. Does not apply to EC2 jobs.

-shells ***shell1,shell2,...***
Specify a comma-separated list of shells to attempt to launch. If the first shell does not exist on the remote machine, then the following ones in the list will be tried. If none of the specified shells can be found, */bin/sh* is used by default. If this option is not specified, it defaults to the environment variable `SHELL` from within the *condor_ssh_to_job* environment. Does not apply to EC2 jobs.

-auto-retry
Specifies that if the job is not yet running, *condor_ssh_to_job* should keep trying periodically until

it succeeds or encounters some other error.

-remove-on-interrupt
If specified, attempt to remove the job from the queue if *condor_ssh_to_job* is interrupted via a CTRL-c or otherwise terminated abnormally.

-X
Enable X11 forwarding. Does not apply to EC2 jobs.

-x
Disable X11 forwarding.

15.51.4 Examples

```
$ condor_ssh_to_job 32.0
Welcome to slot2@tonic.cs.wisc.edu!
Your condor job is running with pid(s) 65881.
$ gdb -p 65881
(gdb) where
...
$ logout
Connection to condor-job.tonic.cs.wisc.edu closed.
```

To upload or download files interactively with *sftp*:

```
$ condor_ssh_to_job -ssh sftp 32.0
Connecting to condor-job.tonic.cs.wisc.edu...
sftp> ls
...
sftp> get outputfile.dat
```

This example shows downloading a file from the job with *scp*. The string "remote" is used in place of a host name in this example. It is not necessary to insert the correct remote host name, or even a valid one, because the connection to the job is created automatically. Therefore, the placeholder string "remote" is perfectly fine.

```
$ condor_ssh_to_job -ssh scp 32 remote:outputfile.dat .
```

This example uses *condor_ssh_to_job* to accomplish the task of running *rsync* to synchronize a local file with a remote file in the job's working directory. Job id 32.0 is used in place of a host name in this example. This causes *rsync* to insert the expected job id in the arguments to *condor_ssh_to_job*.

```
$ rsync -v -e "condor_ssh_to_job" 32.0:outputfile.dat .
```

Note that *condor_ssh_to_job* was added to HTCondor in version 7.3. If one uses *condor_ssh_to_job* to connect to a job on an execute machine running a version of HTCondor older than the 7.3 series, the command will fail with the error message

```
Failed to send CREATE_JOB_OWNER_SEC_SESSION to starter
```

15.51.5 Exit Status

condor_ssh_to_job will exit with a non-zero status value if it fails to set up an ssh session. If it succeeds, it will exit with the status value of the remote command or shell.

15.52 *condor_ssl_fingerprint*

list the fingerprint of X.509 certificates for use with SSL authentication

15.52.1 Synopsis

condor_ssl_fingerprint *[FILE]*

15.52.2 Description

condor_ssl_fingerprint parses provided file for X.509 certificcates and prints prints them to `stdout`. If no file is provided, then it defaults to printing out the user's `known_hosts` file (typically, in `~/.condor/known_hosts`).

If a single PEM-formatted X.509 certificate is found, then its fingerprint is printed.

The X.509 fingerprints can be used to verify the authenticity of an SSL authentication with a remote daemon.

15.52.3 Examples

To print the fingerprint of a host certificate

```
$ condor_token_list
Header: {"alg":"HS256","kid":"POOL"} Payload: {"exp":1565576872,"iat":1565543872,"iss":
↪"htcondor.cs.wisc.edu","scope":"condor:\/DAEMON","sub":"k8sworker@wisc.edu"} File: /
↪home/bucky/.condor/tokens.d/token1
Header: {"alg":"HS256","kid":"POOL"} Payload: {"iat":1572414350,"iss":"htcondor.cs.wisc.
↪edu","scope":"condor:\/WRITE","sub":"bucky@wisc.edu"} File: /home/bucky/.condor/tokens.
↪d/token2
```

15.52.4 Exit Status

condor_token_list will exit with a non-zero status value if it fails to read the token directory, tokens are improperly formatted, or if it experiences some other error. Otherwise, it will exit 0.

15.52.5 See also

condor_token_create(1), *condor_token_fetch(1)*, *condor_token_request(1)*

15.52.6 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.53 *condor_stats*

Display historical information about the HTCondor pool

15.53.1 Synopsis

condor_stats [**-f** *filename*] [**-orgformat**] [**-pool** *centralmanagerhostname[:portnumber]*] [**time-range**] *query-type*

15.53.2 Description

condor_stats displays historic information about an HTCondor pool. Based on the type of information requested, a query is sent to the *condor_collector* daemon, and the information received is displayed using the standard output. If the **-f** option is used, the information will be written to a file instead of to standard output. The **-pool** option can be used to get information from other pools, instead of from the local (default) pool. The *condor_stats* tool is used to query resource information (single or by platform), submitter and user information. If a time range is not specified, the default query provides information for the previous 24 hours. Otherwise, information can be retrieved for other time ranges such as the last specified number of hours, last week, last month, or a specified date range.

The information is displayed in columns separated by tabs. The first column always represents the time, as a percentage of the range of the query. Thus the first entry will have a value close to 0.0, while the last will be close to 100.0. If the **-orgformat** option is used, the time is displayed as number of seconds since the Unix epoch. The information in the remainder of the columns depends on the query type.

Note that logging of pool history must be enabled in the *condor_collector* daemon, otherwise no information will be available.

One query type is required. If multiple queries are specified, only the last one takes effect.

15.53.3 Time Range Options

-lastday
: Get information for the last day.

-lastweek
: Get information for the last week.

-lastmonth
: Get information for the last month.

-lasthours *n*
: Get information for the n last hours.

-from *m d y*
: Get information for the time since the beginning of the specified date. A start date prior to the Unix epoch causes *condor_stats* to print its usage information and quit.

-to *m d y*
: Get information for the time up to the beginning of the specified date, instead of up to now. A finish date in the future causes *condor_stats* to print its usage information and quit.

15.53.4 Query Type Arguments

The query types that do not list all of a category require further specification as given by an argument.

-resourcequery ***hostname***
A single resource query provides information about a single machine. The information also includes the keyboard idle time (in seconds), the load average, and the machine state.

-resourcelist
A query of a single list of resources to provide a list of all the machines for which the *condor_collector* daemon has historic information within the query's time range.

-resgroupquery ***arch/opsys | "Total"***
A query of a specified group to provide information about a group of machines based on their platform (operating system and architecture). The architecture is defined by the machine ClassAd `Arch`, and the operating system is defined by the machine ClassAd `OpSys`. The string "Total" ask for information about all platforms.

The columns displayed are the number of machines that are unclaimed, matched, claimed, preempting, owner, shutdown, delete, backfill, and drained state.

-resgrouplist
Queries for a list of all the group names for which the *condor_collector* has historic information within the query's time range.

-userquery ***email_address/submit_machine***
Query for a specific submitter on a specific machine. The information displayed includes the number of running jobs and the number of idle jobs. An example argument appears as

```
-userquery jondoe@sample.com/onemachine.sample.com
```

-userlist
Queries for the list of all submitters for which the *condor_collector* daemon has historic information within the query's time range.

-usergroupquery ***email_address | "Total"***
Query for all jobs submitted by the specific user, regardless of the machine they were submitted from, or all jobs. The information displayed includes the number of running jobs and the number of idle jobs.

-usergrouplist
Queries for the list of all users for which the *condor_collector* has historic information within the query's time range.

15.53.5 Options

-f ***filename***
Write the information to a file instead of the standard output.

-pool ***centralmanagerhostname[:portnumber]***
Contact the specified central manager instead of the local one.

-orgformat
Display the information in an alternate format for timing, which presents timestamps since the Unix epoch. This argument only affects the display of *resoursequery*, *resgroupquery*, *userquery*, and *usergroupquery*.

15.53.6 Exit Status

condor_stats will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.54 *condor_status*

Display status of the HTCondor pool

15.54.1 Synopsis

condor_status [**-debug**] [*help options*] [*query options*] [*display options*] [*custom options*] [*name ...*]

15.54.2 Description

condor_status is a versatile tool that may be used to monitor and query the HTCondor pool. The *condor_status* tool can be used to query resource information, submitter information, and daemon master information. The specific query sent and the resulting information display is controlled by the query options supplied. Queries and display formats can also be customized.

The options that may be supplied to *condor_status* belong to five groups:

- **Help options** provide information about the *condor_status* tool.
- **Query options** control the content and presentation of status information.
- **Display options** control the display of the queried information.
- **Custom options** allow the user to customize query and display information.
- **Host options** specify specific machines to be queried

At any time, only one *help option*, one *query option* and one *display option* may be specified. Any number of *custom options* and *host options* may be specified.

15.54.3 Options

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-help
: (Help option) Display usage information.

-diagnose
: (Help option) Print out ClassAd query without performing the query.

-absent
: (Query option) Query for and display only absent resources.

-ads ***filename***
: (Query option) Read the set of ClassAds in the file specified by *filename*, instead of querying the *condor_collector*.

-annex ***name***
: (Query option) Query for and display only resources in the named annex.

-any
(Query option) Query all ClassAds and display their type, target type, and name.

-avail
(Query option) Query *condor_startd* ClassAds and identify resources which are available.

-claimed
(Query option) Query *condor_startd* ClassAds and print information about claimed resources.

-cod
(Query option) Display only machine ClassAds that have COD claims. Information displayed includes the claim ID, the owner of the claim, and the state of the COD claim.

-collector
(Query option) Query *condor_collector* ClassAds and display attributes.

-defrag
(Query option) Query *condor_defrag* ClassAds.

-direct *hostname*
(Query option) Go directly to the given host name to get the ClassAds to display. By default, returns the *condor_startd* ClassAd. If **-schedd** is also given, return the *condor_schedd* ClassAd on that host.

-grid
(Query option) Query grid resource ClassAds.

-java
(Query option) Display only Java-capable resources.

-license
(Query option) Display license attributes.

-master
(Query option) Query *condor_master* ClassAds and display daemon master attributes.

-negotiator
(Query option) Query *condor_negotiator* ClassAds and display attributes.

-pool *centralmanagerhostname[:portnumber]*
(Query option) Query the specified central manager using an optional port number. *condor_status* queries the machine specified by the configuration variable `COLLECTOR_HOST` by default.

-run
(Query option) Display information about machines currently running jobs.

-schedd
(Query option) Query *condor_schedd* ClassAds and display attributes.

-server
(Query option) Query *condor_startd* ClassAds and display resource attributes.

-startd
(Query option) Query *condor_startd* ClassAds.

-state
(Query option) Query *condor_startd* ClassAds and display resource state information.

-statistics *WhichStatistics*
(Query option) Can only be used if the **-direct** option has been specified. Identifies which Statistics attributes to include in the ClassAd. *WhichStatistics* is specified using the same syntax as defined for `STATISTICS_TO_PUBLISH`. A definition is in the HTCondor Administrator's manual section on configuration (*HTCondor-wide Configuration File Entries*).

-storage
(Query option) Display attributes of machines with network storage resources.

-submitters
(Query option) Query ClassAds sent by submitters and display important submitter attributes.

-subsystem *type*
(Query option) If *type* is one of *collector*, *negotiator*, *master*, *schedd*, or *startd*, then behavior is the same as the query option without the **-subsystem** option. For example, **-subsystem** *collector* is the same as **-collector**. A value of *type* of *CkptServer*, *Machine*, *DaemonMaster*, or *Scheduler* targets that type of ClassAd.

-vm
(Query option) Query *condor_startd* ClassAds, and display only VM-enabled machines. Information displayed includes the machine name, the virtual machine software version, the state of machine, the virtual machine memory, and the type of networking.

-offline
(Query option) Query *condor_startd* ClassAds, and display, for each machine with at least one offline universe, which universes are offline for it.

-attributes *Attr1[,Attr2 ...]*
(Display option) Explicitly list the attributes in a comma separated list which should be displayed when using the **-xml**, **-json** or **-long** options. Limiting the number of attributes increases the efficiency of the query.

-expert
(Display option) Display shortened error messages.

-long
(Display option) Display entire ClassAds. Implies that totals will not be displayed.

-limit num
(Query option) At most *num* results should be displayed.

-sort *expr*
(Display option) Change the display order to be based on ascending values of an evaluated expression given by *expr*. Evaluated expressions of a string type are in a case insensitive alphabetical order. If multiple **-sort** arguments appear on the command line, the primary sort will be on the leftmost one within the command line, and it is numbered 0. A secondary sort will be based on the second expression, and it is numbered 1. For informational or debugging purposes, the ClassAd output to be displayed will appear as if the ClassAd had two additional attributes. `CondorStatusSortKeyExpr<N>` is the expression, where `<N>` is replaced by the number of the sort. `CondorStatusSortKey<N>` gives the result of evaluating the sort expression that is numbered `<N>`.

-total
(Display option) Display totals only.

-xml
(Display option) Display entire ClassAds, in XML format. The XML format is fully defined in the reference manual, obtained from the ClassAds web page, with a link at http://htcondor.org/classad/classad.html.

-json
(Display option) Display entire ClassAds in JSON format.

-constraint *const*
(Custom option) Add constraint expression.

-compact
(Custom option) Show compact form, with a single line per machine using information from the

partitionable slot. Some information will be incorrect if the machine has static slots.

-format *fmt attr*
(Custom option) Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)`-style conversion specifier. Attributes must be from the resource ClassAd. Expressions are ClassAd expressions and may refer to attributes in the resource ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. %r prints the unevaluated, or raw values. The conversion specifier must match the type of the attribute or expression. %s is suitable for strings such as `Name`, %d for integers such as `LastHeardFrom`, and %f for floating point numbers such as `LoadAvg`. %v identifies the type of the attribute, and then prints the value in an appropriate format. %V identifies the type of the attribute, and then prints the value in an appropriate format as it would appear in the **-long** format. As an example, strings used with %V will have quote marks. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)`-style formats, one may include other text that will be reproduced directly. A format without any conversion specifiers may be specified, but an attribute is still required. Include a backslash followed by an 'n' to specify a line break.

-autoformat[:lhVr,tng] *attr1 [attr2 ...]* **or -af[:lhVr,tng]** *attr1 [attr2 ...]*
(Output option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings. This output option does not work in conjunction with the **-run** option.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may not be used together. The **l** and **h** characters may not be used together.

-print-format *file*
Read output formatting information from the given custom print format file. see *Print Formats* for more information about custom print format files.

-target *filename*
(Custom option) Where evaluation requires a target ClassAd to evaluate against, file *filename* contains the target ClassAd.

-merge *filename*
(Custom option) Ads will be read from *filename*, which may be - to indicate standard in, and compared to the ads selected by the query specified by the remainder of the command line. Ads will be considered the same if their sort keys match; sort keys may be specified with [**-sort** <*key*>]. This option will cause up to three tables to print, in the following order, depending on where a given ad appeared: first, the ads which appeared in the query but not in *filename*; second, the ads which appeared in both the query and in *filename*; third, the ads which appeared in *filename* but not in the query.

By default, banners will label each table. If **-xml** is also given, the same banners will separate three valid XML documents, one for each table. If **-json** is also given, a single JSON object will be produced, with the usual JSON output for each table labeled as an element in the object.

The **-annex** option changes this default so that the banners are not printed and the tables are formatted differently. In this case, the ads in *filename* are expected to have different contents from the ads in the query, so many others will behave strangely.

15.54.4 General Remarks

- The default output from *condor_status* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the HTCondor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_status* is strongly encouraged to use the **-format** option (described above).
- The information obtained from *condor_startd* and *condor_schedd* daemons may sometimes appear to be inconsistent. This is normal since *condor_startd* and *condor_schedd* daemons update the HTCondor manager at different rates, and since there is a delay as information propagates through the network and the system.
- Note that the `ActivityTime` in the `Idle` state is not the amount of time that the machine has been idle. See the section on *condor_startd* states in the Administrator's Manual for more information (*Starting Up, Shutting Down and Reconfiguring the System*).
- When using *condor_status* on a pool with SMP machines, you can either provide the host name, in which case you will get back information about all slots that are represented on that host, or you can list specific slots by name. See the examples below for details.
- If you specify host names, without domains, HTCondor will automatically try to resolve those host names into fully qualified host names for you. This also works when specifying specific nodes of an SMP machine. In this case, everything after the "@" sign is treated as a host name and that is what is resolved.
- You can use the **-direct** option in conjunction with almost any other set of options. However, at this time, not all daemons will respond to direct queries for its ad(s). The *condor_startd* will respond to requests for Startd ads. The *condor_schedd* will respond to requests for Schedd and Submitter ads. So the only options currently not supported with **-direct** are **-master** and **-collector**. Most other options use startd ads for their information, so they work seamlessly with **-direct**. The only other restriction on **-direct** is that you may only use 1 **-direct** option at a time. If you want to query information directly from multiple hosts, you must run *condor_status* multiple times.
- Unless you use the local host name with **-direct**, *condor_status* will still have to contact a collector to find the address where the specified daemon is listening. So, using a **-pool** option in conjunction with **-direct** just tells *condor_status* which collector to query to find the address of the daemon you want. The information actually displayed will still be retrieved directly from the daemon you specified as the argument to **-direct**. Do not use **-direct** to query the Collector ad, just use **-pool** and **-collector**.

15.54.5 Examples

Example 1 To view information from all nodes of an SMP machine, use only the host name. For example, if you had a 4-CPU machine, named `vulture.cs.wisc.edu`, you might see

```
$ condor_status vulture

Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

slot1@vulture.cs.w LINUX      INTEL  Claimed   Busy     1.050   512  0+01:47:42
slot2@vulture.cs.w LINUX      INTEL  Claimed   Busy     1.000   512  0+01:48:19
slot3@vulture.cs.w LINUX      INTEL  Unclaimed Idle     0.070   512  1+11:05:32
slot4@vulture.cs.w LINUX      INTEL  Unclaimed Idle     0.000   512  1+11:05:34

                     Total Owner Claimed Unclaimed Matched Preempting Backfill

         INTEL/LINUX     4     0       2         2       0          0        0

               Total     4     0       2         2       0          0        0
```

Example 2 To view information from a specific nodes of an SMP machine, specify the node directly. You do this by providing the name of the slot. This has the form `slot#@hostname`. For example:

```
$ condor_status slot3@vulture

Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

slot3@vulture.cs.w LINUX      INTEL  Unclaimed Idle     0.070   512  1+11:10:32

                     Total Owner Claimed Unclaimed Matched Preempting Backfill

         INTEL/LINUX     1     0       0         1       0          0        0

               Total     1     0       0         1       0          0        0
```

Example 3 The **-compact** option gives a one line summary of each machine using information from the partitionable slot. If the normal output is this

```
$ condor_status vulture

Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

slot1@vulture.cs.w LINUX      X86_64 Unclaimed Idle      0.000  679  1+03:18:58
slot1_1@vulture.cs LINUX      X86_64 Claimed   Busy      1.160 1152  0+03:21:02
slot1_2@vulture.cs LINUX      X86_64 Claimed   Busy      1.150 2560  0+10:20:50
slot1_3@vulture.cs LINUX      X86_64 Claimed   Busy      1.160 2816  0+01:32:08
slot1_4@vulture.cs LINUX      X86_64 Claimed   Busy      0.000 5081  0+00:00:00

                     Machines Owner Claimed Unclaimed Matched Preempting  Drain

        X86_64/LINUX        5     0       4         1       0          0      0

               Total        5     0       4         1       0          0      0
```

For the same machine in the same state the **-compact** option will show this

```
$ condor_status -compact vulture

Machine            Platform    Slots Cpus Gpus  TotalGb FreCpu  FreeGb  CpuLoad ST Jobs/
↪Min MaxSlotGb

vulture.cs.wisc.ed x64/CentOS7     4    8    2       12      0     .66      .98 Cb      .
↪25      4.96

                     Machines Owner Claimed Unclaimed Matched Preempting  Drain

        X86_64/CentOS7      4     0       4         1       0          0      0

               Total        4     0       4         1       0          0      0
```

The `Slots` column shows that 4 slots have been carved out of the partitionable slot, leaving 0 cpus and .66 Gigabytes of memory free. Static slots will not be counted in the `Slots` column.

The `ST` column shows the consensus state of the dynamic slots using a two character code. The first character is the State, the second is the activity. If there is not a consensus for either the state or activity, then # will be shown. The example shows Cb for Claimed/Busy since all of the dynamic slots are in that state. If one of the dynamic slots were Idle, then C# would be shown.

The `Jobs/Min` shows the recent job start rate for the machine. A large number here is normal for a machine that just came online, but if this number stays above 1 for more than a minute, that can be an indication of a machine is acting as a black hole for jobs, starting them quickly and then failing them just as quickly.

The `MaxSlotGb` column shows the memory allocated to the largest slot in Gigabytes, If the memory allocated for the largest slot cannot be determined, * will be displayed. Static slots are not counted in the `MaxSlotGb` column.

Constraint option examples

The Unix command to use the constraint option to see all machines with the `OpSys` of `"LINUX"`:

```
$ condor_status -constraint OpSys==\"LINUX\"
```

Note that quotation marks must be escaped with the backslash characters for most shells.

The Windows command to do the same thing:

```
> condor_status -constraint " OpSys==""LINUX"" "
```

Note that quotation marks are used to delimit the single argument which is the expression, and the quotation marks that identify the string must be escaped by using a set of two double quote marks without any intervening spaces.

To see all machines that are currently in the Idle state, the Unix command is

```
$ condor_status -constraint State==\"Idle\"
```

To see all machines that are bench marked to have a MIPS rating of more than 750, the Unix command is

```
$ condor_status -constraint 'Mips>750'
```

-cod option example

The **-cod** option displays the status of COD claims within a given HTCondor pool.

```
Name        ID   ClaimState TimeInState RemoteUser JobId Keyword
astro.cs.wi COD1 Idle        0+00:00:04 wright
chopin.cs.w COD1 Running     0+00:02:05 wright     3.0   fractgen
chopin.cs.w COD2 Suspended   0+00:10:21 wright     4.0   fractgen

               Total  Idle  Running  Suspended  Vacating  Killing
 INTEL/LINUX       3     1        1          1         0        0
       Total       3     1        1          1         0        0
```

-format option example To display the name and memory attributes of each job ClassAd in a format that is easily parsable by other tools:

```
$ condor_status -format "%s " Name -format "%d\n" Memory
```

To do the same with the **autoformat** option, run

```
$ condor_status -autoformat Name Memory
```

15.54.6 Exit Status

condor_status will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.55 *condor_store_cred*

securely stash a credential

15.55.1 Synopsis

condor_store_cred -h

condor_store_cred action [*options*]

15.55.2 Description

condor_store_cred stores credentials in a secure manner. There are three actions, each of which can optionally be followed by a hyphen and one of three types.

The actions are:

add[-type]
: Add credential to secure storage

delete[-type]
: Remove credential from secure storage

query[-type]
: Check if a credential has been stored

The types are:

-pwd
: Credential is a password (default)

-krb
Credential is a Kerberos/AFS token

-oauth
Credential is Scitoken or Oauth2 token

Credentials are stashed in a persistent manner; they are maintained across system reboots. When adding a credential, if there is already a credential stashed, the old credential will be overwritten by the new one.

There are two separate uses of the password actions of *condor_store_cred*:

1. A shared pool password is needed in order to implement the `PASSWORD` authentication method. *condor_store_cred* using the **-c** option deals with the password for the implied condor_pool@$(UID_DOMAIN) user name.

   On a Unix machine, *condor_store_cred add[-pwd]* with the **-f** option is used to set the pool password, as needed when used with the `PASSWORD` authentication method. The pool password is placed in a file specified by the `SEC_PASSWORD_FILE` configuration variable.

2. In order to submit a job from a Windows platform machine, or to execute a job on a Windows platform machine utilizing the **run_as_owner** functionality, *condor_store_cred add[-pwd]* stores the password of a user/domain pair securely in the Windows registry. Using this stored password, HTCondor may act on behalf of the submitting user to access files, such as writing output or log files. HTCondor is able to run jobs with the user ID of the submitting user. The password is stored in the same manner as the system does when setting or changing account passwords.

Unless the *-p* argument is used with the *add* or *add-pwd* action, the user is prompted to enter the password twice for confirmation, and characters are not echoed.

The *add-krb* and *add-oauth* actions must be used with the *-i* argument to specify a filename to read from.

The *-oauth* actions require a *-s* service name argument. The *-S* and *-A* options may be used with *add-oauth* to add scopes and/or audience to the credentials or with *query-oauth* to make sure that the scopes or audience match the previously stored credentials. If either *-S* or *-A* are used then the credentials must be in JSON format.

15.55.3 Options

-h
Displays a brief summary of command options.

-c
[-pwd] actions refer to the pool password, as used in the `PASSWORD` authentication method.

-f *filename*
For Unix machines only, generates a pool password file named *filename* that may be used with the `PASSWORD` authentication method.

-i *filename*
Read credential from *filename*. If *filename* is -, read from stdin. Required for *add-krb* and *add-oauth*.

-s *service*
The Oauth2 service. Required for all *-oauth* actions.

-H *handle*
Specify a handle for the given OAuth2 service.

-S *scopes*
Optional comma-separated list of scopes to request for *add-oauth* action. If used with the *query-oauth* action, makes sure that the same scopes were requested in the original credential. Requires credentials to be in JSON format.

-A ***audience***
Optional audience to request for *add-oauth* action. If used with the *query-oauth* action, makes sure that the same audience was requested in the original credential. Requires credentials to be in JSON format.

-n ***machinename***
Apply the command on the given machine.

-p ***password***
Stores *password*, rather than prompting the user to enter a password.

-u ***username***
Specify the user name.

15.55.4 Exit Status

condor_store_cred will exit with a status value of 0 (zero) upon success. If the *query-oauth* action finds a credential but the scopes or audience don't match, *condor_store_cred* will exit with a status value 2 (two). Otherwise, it will exit with the value 1 (one) upon failure.

15.56 *condor_submit*

Queue jobs for execution under HTCondor

15.56.1 Synopsis

condor_submit [**-terse**] [**-verbose**] [**-unused**] [**-file** *submit_file*] [**-name** *schedd_name*] [**-remote** *schedd_name*] [**-addr** <*ip:port*>] [**-pool** *pool_name*] [**-disable**] [**-password** *passphrase*] [**-debug**] [**-append** *command* **...**][**-batch-name** *batch_name*] [**-spool**] [**-dump** *filename*] [**-interactive**] [**-factory**] [**-allow-crlf-script**] [**-dry-run**] [**-maxjobs** *number-of-jobs*] [**-single-cluster**] [**<submit-variable>=<value>**] [*submit description file*] [**-queue** *queue_arguments*]

15.56.2 Description

condor_submit is the program for submitting jobs for execution under HTCondor. *condor_submit* requires one or more submit description commands to direct the queuing of jobs. These commands may come from a file, standard input, the command line, or from some combination of these. One submit description may contain specifications for the queuing of many HTCondor jobs at once. A single invocation of *condor_submit* may cause one or more clusters. A cluster is a set of jobs specified in the submit description between **queue** commands for which the executable is not changed. It is advantageous to submit multiple jobs as a single cluster because the schedd uses much less memory to hold the jobs.

Multiple clusters may be specified within a single submit description. Each cluster must specify a single executable.

The job ClassAd attribute `ClusterId` identifies a cluster.

The *submit description file* argument is the path and file name of the submit description file. If this optional argument is the dash character (-), then the commands are taken from standard input. If - is specified for the *submit description file*, **-verbose** is implied; this can be overridden by specifying **-terse**.

If no *submit description file* argument is given, and no *-queue* argument is given, commands are taken automatically from standard input.

Note that submission of jobs from a Windows machine requires a stashed password to allow HTCondor to impersonate the user submitting the job. To stash a password, use the *condor_store_cred* command. See the manual page for details.

For lengthy lines within the submit description file, the backslash (\) is a line continuation character. Placing the backslash at the end of a line causes the current line's command to be continued with the next line of the file. Submit description files may contain comments. A comment is any line beginning with a pound character (#).

15.56.3 Options

-terse
Terse output - display JobId ranges only.

-verbose
Verbose output - display the created job ClassAd

-unused
As a default, causes no warnings to be issued about user-defined macros not being used within the submit description file. The meaning reverses (toggles) when the configuration variable `WARN_ON_UNUSED_SUBMIT_FILE_MACROS` is set to the non default value of `False`. Printing the warnings can help identify spelling errors of submit description file commands. The warnings are sent to stderr.

-file *submit_file*
Use *submit_file* as the submit description file. This is equivalent to providing *submit_file* as an argument without the preceding *-file*.

-name *schedd_name*
Submit to the specified *condor_schedd*. Use this option to submit to a *condor_schedd* other than the default local one. *schedd_name* is the value of the `Name` ClassAd attribute on the machine where the *condor_schedd* daemon runs.

-remote *schedd_name*
Submit to the specified *condor_schedd*, spooling all required input files over the network connection. *schedd_name* is the value of the `Name` ClassAd attribute on the machine where the *condor_schedd* daemon runs. This option is equivalent to using both **-name** and **-spool**.

-addr <*ip:port*>
Submit to the *condor_schedd* at the IP address and port given by the sinful string argument <*ip:port*>.

-pool *pool_name*
Look in the specified pool for the *condor_schedd* to submit to. This option is used with **-name** or **-remote**.

-disable
Disable file permission checks when submitting a job for read permissions on all input files, such as those defined by commands **input** and **transfer_input_files** , as well as write permission to output files, such as a log file defined by **log** and output files defined with **output** or **transfer_output_files** .

-debug
Cause debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-append *command*
Augment the commands in the submit description file with the given *command*. This command will be considered to immediately precede the **queue** command within the submit description file, and come after all other previous commands. If the *command* specifies a **queue** command, as in the example

```
condor_submit mysubmitfile -append "queue input in A, B, C"
```

then the entire **-append** command line option and its arguments are converted to

```
condor_submit mysubmitfile -queue input in A, B, C
```

The submit description file is not modified. Multiple commands are specified by using the **-append** option multiple times. Each new command is given in a separate **-append** option. Commands with spaces in them will need to be enclosed in double quote marks.

-batch-name *batch_name*
Set the batch name for this submit. The batch name is displayed by *condor_q* **-batch**. It is intended for use by users to give meaningful names to their jobs and to influence how *condor_q* groups jobs for display. Use of this argument takes precedence over a batch name specified in the submit description file itself.

-spool
Spool all required input files, job event log, and proxy over the connection to the *condor_schedd*. After submission, modify local copies of the files without affecting your jobs. Any output files for completed jobs need to be retrieved with *condor_transfer_data*.

-dump *filename*
Sends all ClassAds to the specified file, instead of to the *condor_schedd*.

-interactive
Indicates that the user wants to run an interactive shell on an execute machine in the pool. This is equivalent to creating a submit description file of a vanilla universe sleep job, and then running *condor_ssh_to_job* by hand. Without any additional arguments, *condor_submit* with the -interactive flag creates a dummy vanilla universe job that sleeps, submits it to the local scheduler, waits for the job to run, and then launches *condor_ssh_to_job* to run a shell. If the user would like to run the shell on a machine that matches a particular **requirements** expression, the submit description file is specified, and it will contain the expression. Note that all policy expressions specified in the submit description file are honored, but any **executable** or **universe** commands are overwritten to be sleep and vanilla. The job ClassAd attribute `InteractiveJob` is set to `True` to identify interactive jobs for *condor_startd* policy usage.

-factory
Sends all of the jobs as a late materialization job factory. A job factory consists of a single cluster classad and a digest containing the submit commands necessary to describe the differences between jobs. If the `Queue` statement has itemdata, then the itemdata will be sent. Using this option is equivalent to using the **max_materialize** submit command.

-allow-crlf-script
Changes the check for an invalid line ending on the executable script's `#!` line from an ERROR to a WARNING. The `#!` line will be ignored by Windows, so it won't matter if it is invalid; but Unix and Linux will not run a script that has a Windows/DOS line ending on the first line of the script. So *condor_submit* will not allow such a script to be submitted as the job's executable unless this option is supplied.

-dry-run *file*
Parse the submit description file, sending the resulting job ClassAd to the file given by *file*, but do not submit the job(s). This permits observation of the job specification, and it facilitates debugging the submit description file contents. If *file* is **-**, the output is written to `stdout`.

-maxjobs *number-of-jobs*
If the total number of jobs specified by the submit description file is more than the integer value given by *number-of-jobs*, then no jobs are submitted for execution and an error message is generated. A 0 or negative value for the *number-of-jobs* causes no limit to be imposed.

-single-cluster
If the jobs specified by the submit description file causes more than a single cluster value to be assigned, then no jobs are submitted for execution and an error message is generated.

<submit-variable>=<value>
Defines a submit command or submit variable with a value, and parses it as if it was placed at the beginning of the submit description file. The submit description file is not changed. To correctly parse the *condor_submit* command line, this option must be specified without white space characters before and after the equals sign (=), or the entire option must be surrounded by double quote marks.

-queue ***queue_arguments***
A command line specification of how many jobs to queue, which is only permitted if the submit description file does not have a **queue** command. The *queue_arguments* are the same as may be within a submit description file. The parsing of the *queue_arguments* finishes at the end of the line or when a dash character (-) is encountered. Therefore, its best placement within the command line will be at the end of the command line.

On a Unix command line, the shell expands file globs before parsing occurs.

15.56.4 Submit Description File Commands

Note: more information on submitting HTCondor jobs can be found here: *Submitting a Job*.

As of version 8.5.6, the *condor_submit* language supports multi-line values in commands. The syntax is the same as the configuration language (see more details here: *Multi-Line Values*).

Each submit description file describes one or more clusters of jobs to be placed in the HTCondor execution pool. All jobs in a cluster must share the same executable, but they may have different input and output files, and different program arguments. The submit description file is generally the last command-line argument to *condor_submit*. If the submit description file argument is omitted, *condor_submit* will read the submit description from standard input.

The submit description file must contain at least one *executable* command and at least one *queue* command. All of the other commands have default actions.

Note that a submit file that contains more than one executable command will produce multiple clusters when submitted. This is not generally recommended, and is not allowed for submit files that are run as DAG node jobs by condor_dagman.

The commands which can appear in the submit description file are numerous. They are listed here in alphabetical order by category.

BASIC COMMANDS

arguments = <argument_list>
List of arguments to be supplied to the executable as part of the command line.

In the **java** universe, the first argument must be the name of the class containing `main`.

There are two permissible formats for specifying arguments, identified as the old syntax and the new syntax. The old syntax supports white space characters within arguments only in special circumstances; when used, the command line arguments are represented in the job ClassAd attribute `Args`. The new syntax supports uniform quoting of white space characters within arguments; when used, the command line arguments are represented in the job ClassAd attribute `Arguments`.

Old Syntax

In the old syntax, individual command line arguments are delimited (separated) by space characters. To allow a double quote mark in an argument, it is escaped with a backslash; that is, the two character sequence \" becomes a single double quote mark within an argument.

Further interpretation of the argument string differs depending on the operating system. On Windows, the entire argument string is passed verbatim (other than the backslash in front of double quote marks) to the Windows application. Most Windows applications will allow spaces within an argument value by surrounding the argument with double quotes marks. In all other cases, there is no further interpretation of the arguments.

Example:

```
arguments = one \"two\" 'three'
```

Produces in Unix vanilla universe:

```
argument 1: one
argument 2: "two"
argument 3: 'three'
```

New Syntax

Here are the rules for using the new syntax:

1. The entire string representing the command line arguments is surrounded by double quote marks. This permits the white space characters of spaces and tabs to potentially be embedded within a single argument. Putting the double quote mark within the arguments is accomplished by escaping it with another double quote mark.
2. The white space characters of spaces or tabs delimit arguments.
3. To embed white space characters of spaces or tabs within a single argument, surround the entire argument with single quote marks.
4. To insert a literal single quote mark, escape it within an argument already delimited by single quote marks by adding another single quote mark.

Example:

```
arguments = "3 simple arguments"
```

Produces:

```
argument 1: 3
argument 2: simple
argument 3: arguments
```

Another example:

```
arguments = "one 'two with spaces' 3"
```

Produces:

```
argument 1: one
argument 2: two with spaces
argument 3: 3
```

And yet another example:

```
arguments = "one ""two"" 'spacey ''quoted'' argument'"
```

Produces:

```
argument 1: one
argument 2: "two"
argument 3: spacey 'quoted' argument
```

Notice that in the new syntax, the backslash has no special meaning. This is for the convenience of Windows users.

environment = <parameter_list>
List of environment variables.

There are two different formats for specifying the environment variables: the old format and the new format. The old format is retained for backward-compatibility. It suffers from a platform-dependent syntax and the inability to insert some special characters into the environment.

The new syntax for specifying environment values:

1. Put double quote marks around the entire argument string. This distinguishes the new syntax from the old. The old syntax does not have double quote marks around it. Any literal double quote marks within the string must be escaped by repeating the double quote mark.
2. Each environment entry has the form

   ```
   <name>=<value>
   ```

3. Use white space (space or tab characters) to separate environment entries.
4. To put any white space in an environment entry, surround the space and as much of the surrounding entry as desired with single quote marks.
5. To insert a literal single quote mark, repeat the single quote mark anywhere inside of a section surrounded by single quote marks.

Example:

```
environment = "one=1 two=""2"" three='spacey ''quoted'' value'"
```

Produces the following environment entries:

```
one=1
two="2"
three=spacey 'quoted' value
```

Under the old syntax, there are no double quote marks surrounding the environment specification. Each environment entry remains of the form

```
<name>=<value>
```

Under Unix, list multiple environment entries by separating them with a semicolon (;). Under Windows, separate multiple entries with a vertical bar (|). There is no way to insert a literal semicolon under Unix or a literal vertical bar under Windows. Note that spaces are accepted, but rarely desired, characters within parameter names and values, because they are treated as literal characters, not separators or ignored white space. Place spaces within the parameter list only if required.

A Unix example:

```
environment = one=1;two=2;three="quotes have no 'special' meaning"
```

This produces the following:

```
one=1
two=2
three="quotes have no 'special' meaning"
```

If the environment is set with the **environment** command and **getenv** is also set, values specified with **environment** override values in the submitter's environment (regardless of the order of the **environment** and **getenv** commands).

error = <pathname>
A path and file name used by HTCondor to capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). A path is given with respect to the file system of the machine on which the job is submitted. The file is written (by the job) in the remote scratch directory of the machine where the job is executed. When the job exits, the resulting file is transferred back to the machine where the job was submitted, and the path is utilized for file placement. If you specify a relative path, the final path will be relative to the job's initial working directory, and HTCondor will create directories as necessary to transfer the file. If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, error messages are ignored for submission to a Windows machine. More than one job should not use the same error file, since this will cause one job to overwrite the errors of another. If HTCondor detects that the error and output files for a job are the same, it will run the job such that the output and error data is merged.

executable = <pathname>
An optional path and a required file name of the executable file for this job cluster. Only one **executable** command within a submit description file is guaranteed to work properly. More than one often works.

If no path or a relative path is used, then the executable file is presumed to be relative to the current working directory of the user as the *condor_submit* command is issued.

batch_name = <batch_name>
Set the batch name for this submit. The batch name is displayed by *condor_q* **-batch**. It is intended for use by users to give meaningful names to their jobs and to influence how *condor_q* groups jobs for display. This value in a submit file can be overridden by specifying the **-batch-name** argument on the *condor_submit* command line.

getenv = <<matchlist> | True | False>
If **getenv** is set to `True`, then *condor_submit* will copy all of the user's current shell environment variables at the time of job submission into the job ClassAd. The job will therefore execute with the same set of environment variables that the user had at submit time. Defaults to `False`. A wholesale import of the user's environment is very likely to lead to problems executing the job on a remote machine unless there is a shared file system for users' home directories between the access point and execute machine. So rather than setting getenv to `True`, it is much better to set it to a list of environment variables to import.

Matchlist is a comma, semicolon or space separated list of environment variable names and name patterns that match or reject names. Matchlist members are matched case-insensitively to each name in the environment and those that match are imported. Matchlist members can contain * as wildcard character which matches anything at that position. Members can have two * characters if one of them is at the end. Members can be prefixed with ! to force a matching environment variable to not be

imported. The order of members in the Matchlist has no effect on the result. `getenv = true` is equivalent to `getenv = *`

Prior to HTCondor 8.9.7 `getenv` allows only `True` or `False` as values.

Examples:

```
# import everything except PATH and INCLUDE (also path, include and other␣
↪case-variants)
getenv = !PATH, !INCLUDE

# import everything with CUDA in the name
getenv = *cuda*

# Import every environment variable that starts with P or Q, except PATH
getenv = !path, P*, Q*
```

If the environment is set with the **environment** command and **getenv** is also set, values specified with **environment** override values in the submitter's environment (regardless of the order of the **environment** and **getenv** commands).

input = <pathname>

HTCondor assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (`stdin`, `stdout`, and `stderr`), must refer to files. Thus, the file name specified with **input** should contain any keyboard input the program requires (that is, this file becomes `stdin`). A path is given with respect to the file system of the machine on which the job is submitted. The file is transferred before execution to the remote scratch directory of the machine where the job is executed. If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, input is ignored for submission to a Windows machine.

Note that this command does not refer to the command-line arguments of the program. The command-line arguments are specified by the **arguments** command.

log = <pathname>

Use **log** to specify a file name where HTCondor will write a log file of what is happening with this job cluster, called a job event log. For example, HTCondor will place a log entry into this file when and where the job begins running, when it transfers files, if the job is evicted, and when the job completes. Most users find specifying a **log** file to be handy; its use is recommended. If no **log** entry is specified, HTCondor does not create a log for this cluster. If a relative path is specified, it is relative to the current working directory as the job is submitted or the directory specified by submit command **initialdir** on the access point.

notification = <Always | Complete | Error | Never>

Owners of HTCondor jobs are notified by e-mail when certain events occur. If defined by *Always* or *Complete*, the owner will be notified when the job terminates. If defined by *Error*, the owner will only be notified if the job terminates abnormally, (as defined by `JobSuccessExitCode`, if defined) or if the job is placed on hold because of a failure, and not by user request. If defined by *Never* (the default), the owner will not receive e-mail, regardless to what happens to the job. The HTCondor User's manual documents statistics included in the e-mail.

notify_user = <email-address>
Used to specify the e-mail address to use when HTCondor sends e-mail about a job. If not specified, HTCondor defaults to using the e-mail address defined by

```
job-owner@UID_DOMAIN
```

where the configuration variable `UID_DOMAIN` is specified by the HTCondor site administrator. If `UID_DOMAIN` has not been specified, HTCondor sends the e-mail to:

```
job-owner@submit-machine-name
```

output = <pathname>
The **output** file captures any information the program would ordinarily write to the screen (that is, this file becomes `stdout`). A path is given with respect to the file system of the machine on which the job is submitted. The file is written (by the job) in the remote scratch directory of the machine where the job is executed. When the job exits, the resulting file is transferred back to the machine where the job was submitted, and the path is utilized for file placement. If you specify a relative path, the final path will be relative to the job's initial working directory, and HTCondor will create directories as necessary to transfer the file. If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, output is ignored for submission to a Windows machine. Multiple jobs should not use the same output file, since this will cause one job to overwrite the output of another. If HTCondor detects that the error and output files for a job are the same, it will run the job such that the output and error data is merged.

Note that if a program explicitly opens and writes to a file, that file should not be specified as the **output** file.

priority = <integer>
An HTCondor job priority can be any integer, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority. Note that this priority is on a per user basis. One user with many jobs may use this command to order his/her own jobs, and this will have no effect on whether or not these jobs will run ahead of another user's jobs.

Note that the priority setting in an HTCondor submit file will be overridden by *condor_dagman* if the submit file is used for a node in a DAG, and the priority of the node within the DAG is non-zero (see *Setting Priorities for Nodes* for more details).

queue [<int expr>]
Places zero or more copies of the job into the HTCondor queue.

queue
[**<int expr>**] [**<varname>**] **in** [**slice**] **<list of items>** Places zero or more copies of the job in the queue based on items in a **<list of items>**

queue
[**<int expr>**] [**<varname>**] **matching** [**files | dirs**] [**slice**] **<list of items with file globbing>**] Places zero or more copies of the job in the queue based on files that match a **<list of items with file globbing>**

queue
[**<int expr>**] [**<list of varnames>**] **from** [**slice**] **<file name> | <list of items>**] Places zero or more copies of the job in the queue based on lines from the submit file or from **<file name>**

The optional argument <*int expr*> specifies how many times to repeat the job submission for a given set of arguments. It may be an integer or an expression that evaluates to an integer, and it defaults to 1. All but the first form of this command are various ways of specifying a list of items. When these forms are used <*int expr*> jobs will be queued for each item in the list. The *in*, *matching* and *from* keyword indicates how the list will be specified.

- *in* The list of items is an explicit comma and/or space separated **<list of items>**. If the **<list of items>** begins with an open paren, and the close paren is not on the same line as the open, then the list continues until a line that begins with a close paren is read from the submit file.
- *matching* Each item in the **<list of items with file globbing>** will be matched against the names of files and directories relative to the current directory, the set of matching names is the resulting list of items.
  - *files* Only filenames will matched.
  - *dirs* Only directory names will be matched.
- *from* **<file name> | <list of items>** Each line from **<file name>** or **<list of items>** is a single item, this allows for multiple variables to be set for each item. Lines from **<file name>** or **<list of items>** will be split on comma and/or space until there are values for each of the variables specified in **<list of varnames>**. The last variable will contain the remainder of the line. When the **<list of items>** form is used, the list continues until the first line that begins with a close paren, and lines beginning with pound sign ('#') will be skipped. When using the **<file name>** form, if the **<file name>** ends with |, then it will be executed as a script whatever the script writes to `stdout` will be the list of items.

The optional argument <*varname*> or <*list of varnames*> is the name or names of of variables that will be set to the value of the current item when queuing the job. If no <*varname*> is specified the variable ITEM will be used. Leading and trailing whitespace be trimmed. The optional argument <*slice*> is a python style slice selecting only some of the items in the list of items. Negative step values are not supported.

A submit file may contain more than one **queue** statement, and if desired, any commands may be placed between subsequent **queue** commands, such as new **input** , **output** , **error** , **initialdir** , or **arguments** commands. This is handy when submitting multiple runs into one cluster with one submit description file.

universe = <vanilla | scheduler | local | grid | java | vm | parallel | docker | container>
Specifies which HTCondor universe to use when running this job. The HTCondor universe specifies an HTCondor execution environment.

The **vanilla** universe is the default (except where the configuration variable `DEFAULT_UNIVERSE` defines it otherwise).

The **scheduler** universe is for a job that is to run on the machine where the job is submitted. This universe is intended for a job that acts as a metascheduler and will not be preempted.

The **local** universe is for a job that is to run on the machine where the job is submitted. This universe runs the job immediately and will not preempt the job.

The **grid** universe forwards the job to an external job management system. Further specification of the **grid** universe is done with the **grid_resource** command.

The **java** universe is for programs written to the Java Virtual Machine.

The **vm** universe facilitates the execution of a virtual machine.

The **parallel** universe is for parallel jobs (e.g. MPI) that require multiple machines in order to run.

The **docker** universe runs a docker container as an HTCondor job.

The **container** universe runs a container as an HTCondor job using a supported container runtime system on the Execution Point.

max_materialize = <limit>
Submit jobs as a late materialization factory and instruct the *condor_schedd* to keep the given number of jobs materialized. Use this option to reduce the load on the *condor_schedd* when submitting a large number of jobs. The limit can be an expression but it must evaluate to a constant at submit time. A limit less than 1 will be treated as unlimited. The *condor_schedd* can be configured to have a materialization limit as well, the lower of the two limits will be used. (see *Submitting Lots of Jobs* for more details).

max_idle = <limit>
Submit jobs as a late materialization factory and instruct the *condor_schedd* to keep the given number of non-running jobs materialized. Use this option to reduce the load on the *condor_schedd* when submitting a large number of jobs. The limit may be an expression but it must evaluate to a constant at submit time. Jobs in the Held state are considered to be Idle for this limit. A limit of less than 1 will prevent jobs from being materialized although the factory will still be submitted to the *condor_schedd*. (see *Submitting Lots of Jobs* for more details).

COMMANDS FOR MATCHMAKING

rank = <ClassAd Float Expression>
A ClassAd Floating-Point expression that states how to rank machines which have already met the requirements expression. Essentially, rank expresses preference. A higher numeric value equals better rank. HTCondor will give the job the machine with the highest rank. For example,

```
request_memory = max({60, Target.TotalSlotMemory})
rank = Memory
```

asks HTCondor to find all available machines with more than 60 megabytes of memory and give to the job the machine with the most amount of memory. The HTCondor User's Manual contains complete information on the syntax and available attributes that can be used in the ClassAd expression.

request_cpus = <num-cpus>
A requested number of CPUs (cores). If not specified, the number requested will be 1. If specified, the expression

```
&& (RequestCpus <= Target.Cpus)
```

is appended to the **requirements** expression for the job.

For pools that enable dynamic *condor_startd* provisioning, specifies the minimum number of CPUs requested for this job, resulting in a dynamic slot being created with this many cores.

request_disk = <quantity>
The requested amount of disk space in KiB requested for this job. If not specified, it will be set to the job ClassAd attribute `DiskUsage`. The expression

```
&& (RequestDisk <= Target.Disk)
```

is appended to the **requirements** expression for the job.

For pools that enable dynamic *condor_startd* provisioning, a dynamic slot will be created with at least this much disk space.

Characters may be appended to a numerical value to indicate units. `K` or `KB` indicates KiB, 2^{10} numbers of bytes. `M` or `MB` indicates MiB, 2^{20} numbers of bytes. `G` or `GB` indicates GiB, 2^{30} numbers of bytes. `T` or `TB` indicates TiB, 2^{40} numbers of bytes.

request_gpus = <num-gpus>
A requested number of GPUs. If not specified, no GPUs will be requested. If specified and `require_gpus` is not also specified, the expression

```
&& (Target.GPUs >= RequestGPUs)
```

is appended to the **requirements** expression for the job.

For pools that enable dynamic *condor_startd* provisioning, specifies the minimum number of GPUs requested for this job, resulting in a dynamic slot being created with this many GPUs.

require_gpus = <constraint-expression>
A constraint on the properties of GPUs when used with a non-zero `request_gpus` value. If not specified, no constraint on GPUs will be added to the job. If specified and `request_gpus` is non-zero, the expression

```
&& (countMatches(MY.RequireGPUs, TARGET.AvailableGPUs) >= RequestGPUs)
```

is appended to the **requirements** expression for the job. This expression cannot be evaluated by HTCondor prior to version 9.8.0. A warning to this will effect will be printed when *condor_submit* detects this condition.

For pools that enable dynamic *condor_startd* provisioning and are at least version 9.8.0, the constraint will be tested against the properties of AvailableGPUs and only those that match will be assigned to the dynamic slot.

request_memory = <quantity>
The required amount of memory in MiB that this job needs to avoid excessive swapping. If not specified and the submit command **vm_memory** is specified, then the value specified for **vm_memory** defines **request_memory** . If neither **request_memory** nor **vm_memory** is specified, the value is set by the configuration variable `JOB_DEFAULT_REQUESTMEMORY` . The actual amount of memory used by a job is represented by the job ClassAd attribute `MemoryUsage`.

For pools that enable dynamic *condor_startd* provisioning, a dynamic slot will be created with at least this much RAM.

The expression

```
&& (RequestMemory <= Target.Memory)
```

is appended to the **requirements** expression for the job.

Characters may be appended to a numerical value to indicate units. K or KB indicates KiB, 2^{10} numbers of bytes. M or MB indicates MiB, 2^{20} numbers of bytes. G or GB indicates GiB, 2^{30} numbers of bytes. T or TB indicates TiB, 2^{40} numbers of bytes.

request_<name> = <quantity>
The required amount of the custom machine resource identified by <name> that this job needs. The custom machine resource is defined in the machine's configuration. Machines that have available GPUs will define <name> to be GPUs. <name> must be at least two characters, and must not begin with _. If <name> is either Cpu or Gpu a warning will be printed since these are common typos.

cuda_version = <version>
The version of the CUDA runtime, if any, used or required by this job, specified as <major>.<minor> (for example, 9.1). If the minor version number is zero, you may specify only the major version number. A single version number of 1000 or higher is assumed to be the integer-coded version number (major * 1000 + (minor % 100)).

This does *not* arrange for the CUDA runtime to be present, only for the job to run on a machine whose driver supports the specified version.

requirements = <ClassAd Boolean Expression>
The requirements command is a boolean ClassAd expression which uses C-like operators. In order for any job in this cluster to run on a given machine, this requirements expression must evaluate to true on the given machine.

For scheduler and local universe jobs, the requirements expression is evaluated against the Scheduler ClassAd which represents the the *condor_schedd* daemon running on the access point, rather than a remote machine. Like all commands in the submit description file, if multiple requirements commands are present, all but the last one are ignored. By default, *condor_submit* appends the following clauses to the requirements expression:

1. Arch and OpSys are set equal to the Arch and OpSys of the submit machine. In other words: unless you request otherwise, HTCondor will give your job machines with the same architecture and operating system version as the machine running *condor_submit*.
2. Cpus >= RequestCpus, if the job ClassAd attribute RequestCpus is defined.
3. Disk >= RequestDisk, if the job ClassAd attribute RequestDisk is defined. Otherwise, Disk >= DiskUsage is appended to the requirements. The DiskUsage attribute is initialized to the size of the executable plus the size of any files specified in a **transfer_input_files** command. It exists to ensure there is enough disk space on the target machine for HTCondor to copy over both the executable and needed input files. The DiskUsage attribute represents the maximum amount of total disk space required by the job in kilobytes. HTCondor automatically updates the DiskUsage attribute approximately every 20 minutes while the job runs with the amount of space being used by the job on the execute machine.
4. Memory >= RequestMemory, if the job ClassAd attribute RequestMemory is defined.
5. If Universe is set to Vanilla, FileSystemDomain is set equal to the access point's FileSystemDomain.

View the requirements of a job which has already been submitted (along with everything else about the job ClassAd) with the command *condor_q -l*; see the command reference for *condor_q*. Also, see the HTCondor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

FILE TRANSFER COMMANDS

dont_encrypt_input_files = < file1,file2,file... >
A comma and/or space separated list of input files that are not to be network encrypted when transferred with the file transfer mechanism. Specification of files in this manner overrides configuration that would use encryption. Each input file must also be in the list given by **transfer_input_files** . When a path to an input file or directory is specified, this specifies the path to the file on the submit side. A single wild card character (*) may be used in each file name.

dont_encrypt_output_files = < file1,file2,file... >
A comma and/or space separated list of output files that are not to be network encrypted when transferred back with the file transfer mechanism. Specification of files in this manner overrides configuration that would use encryption. The output file(s) must also either be in the list given by **transfer_output_files** or be discovered and to be transferred back with the file transfer mechanism. When a path to an output file or directory is specified, this specifies the path to the file on the execute side. A single wild card character (*) may be used in each file name.

encrypt_execute_directory = <True | False>
Defaults to `False`. If set to `True`, HTCondor will encrypt the contents of the remote scratch directory of the machine where the job is executed. This encryption is transparent to the job itself, but ensures that files left behind on the local disk of the execute machine, perhaps due to a system crash, will remain private. In addition, *condor_submit* will append to the job's **requirements** expression

```
&& (TARGET.HasEncryptExecuteDirectory)
```

to ensure the job is matched to a machine that is capable of encrypting the contents of the execute directory. This support is limited to Windows platforms that use the NTFS file system and Linux platforms with the *ecryptfs-utils* package installed.

encrypt_input_files = < file1,file2,file... >
A comma and/or space separated list of input files that are to be network encrypted when transferred with the file transfer mechanism. Specification of files in this manner overrides configuration that would not use encryption. Each input file must also be in the list given by **transfer_input_files** . When a path to an input file or directory is specified, this specifies the path to the file on the submit side. A single wild card character (*) may be used in each file name. The method of encryption utilized will be as agreed upon in security negotiation; if that negotiation failed, then the file transfer mechanism must also fail for files to be network encrypted.

encrypt_output_files = < file1,file2,file... >
A comma and/or space separated list of output files that are to be network encrypted when transferred back with the file transfer mechanism. Specification of files in this manner overrides configuration that would not use encryption. The output file(s) must also either be in the list given by **transfer_output_files** or be discovered and to be transferred back with the file transfer mechanism. When a path to an output file or directory is specified, this specifies the path to the file on the execute side. A single wild card character (*) may be used in each file name. The method of encryption utilized will be as agreed upon in security negotiation; if that negotiation failed, then the file transfer mechanism must also fail for files to be network encrypted.

erase_output_and_error_on_restart
If false, and `when_to_transfer_output` is `ON_EXIT_OR_EVICT`, HTCondor will append to the output and error logs rather than erase (truncate) them when the job restarts.

max_transfer_input_mb = <ClassAd Integer Expression>
This integer expression specifies the maximum allowed total size in MiB of the input files that are transferred for a job. This expression does not apply to grid universe or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not defined, the value set by configuration variable `MAX_TRANSFER_INPUT_MB` is used. If the observed size of all input files at submit time is larger than the limit, the job will be immediately placed on hold with a `HoldReasonCode` value of 32. If the job passes this initial test, but the size of the input files increases or the limit decreases so that the limit is violated, the job will be placed on hold at the time when the file transfer is attempted.

max_transfer_output_mb = <ClassAd Integer Expression>
This integer expression specifies the maximum allowed total size in MiB of the output files that are transferred for a job. This expression does not apply to grid universe or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not set, the value set by configuration variable `MAX_TRANSFER_OUTPUT_MB` is used. If the total size of the job's output files to be transferred is larger than the limit, the job will be placed on hold with a `HoldReasonCode` value of 33. The output will be transferred up to the point when the limit is hit, so some files may be fully transferred, some partially, and some not at all.

output_destination = <destination-URL>
When present, defines a URL that specifies both a plug-in and a destination for the transfer of the entire output sandbox or a subset of output files as specified by the submit command **transfer_output_files** . The plug-in does the transfer of files, and no files are sent back to the access point. The HTCondor Administrator's manual has full details.

should_transfer_files = <YES | NO | IF_NEEDED >
The **should_transfer_files** setting is used to define if HTCondor should transfer files to and from the remote machine where the job runs. The file transfer mechanism is used to run jobs on machines which do not have a shared file system with the submit machine. **should_transfer_files** equal to *YES* will cause HTCondor to always transfer files for the job. *NO* disables HTCondor's file transfer mechanism. *IF_NEEDED* will not transfer files for the job if it is matched with a resource in the same `FileSystemDomain` as the access point (and therefore, on a machine with the same shared file system). If the job is matched with a remote resource in a different `FileSystemDomain`, HTCondor will transfer the necessary files.

For more information about this and other settings related to transferring files, see the HTCondor User's manual section on the file transfer mechanism.

Note that **should_transfer_files** is not supported for jobs submitted to the grid universe.

skip_filechecks = <True | False>
When `True`, file permission checks for the submitted job are disabled. When `False`, file permissions are checked; this is the behavior when this command is not present in the submit description file. File permissions are checked for read permissions on all input files, such as those defined by commands **input** and **transfer_input_files** , and for write permission to output files, such as a log file defined by **log** and output files defined with **output** or **transfer_output_files** .

stream_error = <True | False>
If `True`, then `stderr` is streamed back to the machine from which the job was submitted. If `False`, `stderr` is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferErr` is `False`. The default value is `False`. This command must be used in conjunction with **error** , otherwise `stderr` will sent to `/dev/null` on Unix machines and ignored on Windows machines.

stream_input = <True | False>
If `True`, then `stdin` is streamed from the machine on which the job was submitted. The default value is `False`. The command is only relevant for jobs submitted to the vanilla or java universes, and it is ignored by the grid universe. This command must be used in conjunction with **input** , otherwise `stdin` will be `/dev/null` on Unix machines and ignored on Windows machines.

stream_output = <True | False>
If `True`, then `stdout` is streamed back to the machine from which the job was submitted. If `False`, `stdout` is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferOut` is `False`. The default value is `False`. This command must be used in conjunction with **output** , otherwise `stdout` will sent to `/dev/null` on Unix machines and ignored on Windows machines.

transfer_executable = <True | False>
This command is applicable to jobs submitted to the grid and vanilla universes. If **transfer_executable** is set to `False`, then HTCondor looks for the executable on the remote machine, and does not transfer the executable over. This is useful for an already pre-staged executable; HTCondor behaves more like rsh. The default value is `True`.

transfer_input_files = < file1,file2,file... >
A comma-delimited list of all the files and directories to be transferred into the working directory for the job, before the job is started. By default, the file specified in the **executable** command and any file specified in the **input** command (for example, `stdin`) are transferred.

When a path to an input file or directory is specified, this specifies the path to the file on the submit side. The file is placed in the job's temporary scratch directory on the execute side, and it is named using the base name of the original path. For example, `/path/to/input_file` becomes `input_file` in the job's scratch directory.

When a directory is specified, the behavior depends on whether there is a trailing path separator character. When a directory is specified with a trailing path separator, it is as if each of the items within the directory were listed in the transfer list. Therefore, the contents are transferred, but the directory itself is not. When there is no trailing path separator, the directory itself is transferred with all of its contents inside it. On platforms such as Windows where the path separator is not a forward slash (/), a trailing forward slash is treated as equivalent to a trailing path separator. An example of an input directory specified with a trailing forward slash is `input_data/`.

For grid universe jobs other than HTCondor-C, the transfer of directories is not currently supported.

Symbolic links to files are transferred as the files they point to. Transfer of symbolic links to directories is not currently supported.

For vanilla and vm universe jobs only, a file may be specified by giving a URL, instead of a file name. The implementation for URL transfers requires both configuration and available plug-in.

If you have a plugin which handles `https://` URLs (and HTCondor ships with one enabled), HTCondor supports pre-signing S3 URLs. This allows you to specify S3 URLs for this command, for `transfer_output_remaps`, and for `output_destination`. By pre-signing the URLs on the submit node, HTCondor avoids transferring your S3 credentials to the execute node. You must specify `aws_access_key_id_file` and `aws_secret_access_key_file`; you may specify `aws_region`, if necessary; see below. To use the S3 service provided by AWS, use S3 URLs of the following forms:

```
# For older buckets that aren't region-specific.
s3://<bucket>/<key>

# For newer, region-specific buckets.
s3://<bucket>.s3.<region>.amazonaws.com/<key>
```

To use other S3 services, where `<host>` must contain a `.`:

```
s3://<host>/<key>

# If necessary
aws_region = <region>
```

You may specify the corresponding access key ID and secret access key with `s3_access_key_id_file` and `s3_secret_access_key_file` if you prefer (which may reduce confusion, if you're not using AWS).

If you must access S3 using temporary credentials, you may specify the temporary credentials using `aws_access_key_id_file` and `aws_secret_access_key_file` for the files containing the corresponding temporary token, and `+EC2SessionToken` for the file containing the session token.

Temporary credentials have a limited lifetime. If you are using S3 only to download input files, the job must start before the credentials expire. If you are using S3 to upload output files, the job must finish before the credentials expire. HTCondor does not know when the credentials will expire; if they do so before they are needed, file transfer will fail.

HTCondor does not presently support transferring entire buckets or directories from S3.

HTCondor supports Google Cloud Storage URLs – `gs://` – via Google's "interoperability" API. You may specify `gs://` URLs as if they were `s3://` URLs, and they work the same way. You may specify the corresponding access key ID and secret access key with `gs_access_key_id_file` and `gs_secret_access_key_file` if you prefer (which may reduce confusion).

Note that (at present), you may not provide more than one set of credentials for `s3://` or `gs://` file transfer; this implies that all such URLs download from or upload to the same service.

transfer_output_files = < file1,file2,file... >
This command forms an explicit list of output files and directories to be transferred back from the temporary working directory on the execute machine to the access point. If there are multiple files, they must be delimited with commas. Setting **transfer_output_files** to the empty string ("") means that no files are to be transferred.

For HTCondor-C jobs and all other non-grid universe jobs, if **transfer_output_files** is not specified, HTCondor will automatically transfer back all files in the job's temporary working directory which have been modified or created by the job. Subdirectories are not scanned for output, so if output from subdirectories is desired, the output list must be explicitly specified. For grid universe jobs other than HTCondor-C, desired output files must also be explicitly listed. Another reason to explicitly list output files is for a job that creates many files, and the user wants only a subset transferred back.

For grid universe jobs other than with grid type **condor**, to have files other than standard output and standard error transferred from the execute machine back to the access point, do use **transfer_output_files**, listing all files to be transferred. These files are found on the execute machine in the working directory of the job.

When a path to an output file or directory is specified, it specifies the path to the file on the execute side. As a destination on the submit side, the file is placed in the job's initial working directory, and it is named using the base name of the original path. For example, `path/to/output_file` becomes `output_file` in the job's initial working directory. The name and path of the file that is written on the submit side may be modified by using **transfer_output_remaps** . Note that this remap function only works with files but not with directories.

When a directory is specified, the behavior depends on whether there is a trailing path separator character. When a directory is specified with a trailing path separator, it is as if each of the items within the directory were listed in the transfer list. Therefore, the contents are transferred, but the directory itself is not. When there is no trailing path separator, the directory itself is transferred with all of its contents inside it. On platforms such as Windows where the path separator is not a forward slash (/), a trailing forward slash is treated as equivalent to a trailing path separator. An example of an input directory specified with a trailing forward slash is `input_data/`.

For grid universe jobs other than HTCondor-C, the transfer of directories is not currently supported.

Symbolic links to files are transferred as the files they point to. Transfer of symbolic links to directories is not currently supported.

transfer_checkpoint_files = < file1,file2,file3... >
If present, this command defines the list of files and/or directories which constitute the job's checkpoint. When the job successfully checkpoints – see `checkpoint_exit_code` – these files will be transferred to the submit node's spool.

If this command is absent, the output is transferred instead.

If no files or directories are specified, nothing will be transferred. This is generally not useful.

The list is interpreted like `transfer_output_files`, but there is no corresponding `remaps` command.

preserve_relative_paths = < True | False >
For vanilla and Docker -universe jobs (and others that use the shadow), this command modifies the behavior of the file transfer commands. When set to true, the destination for an entry that is a relative path in a file transfer list becomes its relative path, not its basename. For example, `input_data/b` (and its contents, if it is a directory) will be transferred to `input_data/b`, not `b`. This applies to the input, output, and checkpoint lists.

Trailing slashes are ignored when `preserve_relative_paths` is set.

transfer_output_remaps = < " name = newname ; name2 = newname2 ... ">
This specifies the name (and optionally path) to use when downloading output files from the completed job. Normally, output files are transferred back to the initial working directory with the same name they had in the execution directory. This gives you the option to save them with a different path or name. If you specify a relative path, the final path will be relative to the job's initial working directory, and HTCondor will create directories as necessary to transfer the file.

name describes an output file name produced by your job, and *newname* describes the file name it should be downloaded to. Multiple remaps can be specified by separating each with a semicolon. If

you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash. You cannot specify directories to be remapped.

Note that whether an output file is transferred is controlled by **transfer_output_files**. Listing a file in **transfer_output_remaps** is not sufficient to cause it to be transferred.

transfer_plugins = < tag=plugin ; tag2,tag3=plugin2 ... >
Specifies the file transfer plugins (see *Third Party/Delegated file and credential transfer*) that should be transferred along with the input files prior to invoking file transfer plugins for files specified in *transfer_input_files*. *tag* should be a URL prefix that is used in *transfer_input_files*, and *plugin* is the path to a file transfer plugin that will handle that type of URL transfer.

when_to_transfer_output = < ON_EXIT | ON_EXIT_OR_EVICT | ON_SUCCESS >
Setting `when_to_transfer_output` to `ON_EXIT` will cause HTCondor to transfer the job's output files back to the submitting machine when the job completes (exits on its own). If a job is evicted and started again, the subsequent execution will start with only the executable and input files in the scratch directory sandbox. If `transfer_output_files` is not set, HTCondor considers all new files in the sandbox's top-level directory to be the output; subdirectories and their contents will not be transferred.

Setting `when_to_transfer_output` to `ON_EXIT_OR_EVICT` will cause HTCondor to transfer the job's output files when the job completes (exits on its own) and when the job is evicted. When the job is evicted, HTCondor will transfer the output files to a temporary directory on the submit node (determined by the `SPOOL` configuration variable). When the job restarts, these files will be transferred instead of the input files. If `transfer_output_files` is not set, HTCondor considers all files in the sandbox's top-level directory to be the output; subdirectories and their contents will not be transferred.

Setting `when_to_transfer_output` to `ON_SUCCESS` will cause HTCondor to transfer the job's output files when the job completes successfully. Success is defined by the `success_exit_code` command, which must be set, even if the successful value is the default `0`. If `transfer_output_files` is not set, HTCondor considers all new files in the sandbox's top-level directory to be the output; subdirectories and their contents will not be transferred.

In all three cases, the job will go on hold if `transfer_output_files` specifies a file which does not exist at transfer time.

aws_access_key_id_file, s3_access_key_id_file
One of these commands is required if you specify an `s3://` URL; they specify the file containing the access key ID (and only the access key ID) used to pre-sign the URLs. Use only one.

aws_secret_access_key_file, s3_secret_access_key_file
One of these commands is required if you specify an `s3://` URL; they specify the file containing the secret access key (and only the secret access key) used to pre-sign the URLs. Use only one.

aws_region
Optional if you specify an S3 URL (and ignored otherwise), this command specifies the region to use if one is not specified in the URL.

gs_access_key_id_file
Required if you specify a `gs://` URLs, this command specifies the file containing the access key ID (and only the access key ID) used to pre-sign the URLs.

gs_secret_access_key_file
Required if you specify a `gs://` URLs, this command specifies the file containing the secret access key (and only the secret access key) used to pre-sign the URLs.

POLICY COMMANDS

allowed_execute_duration = <integer>
The longest time for which a job may be executing. Jobs which exceed this duration will go on hold. This time does not include file-transfer time. Jobs which self-checkpoint have this long to write out each checkpoint.

This attribute is intended to help minimize the time wasted by jobs which may erroneously run forever.

allowed_job_duration = <integer>
The longest time for which a job may continuously be in the running state. Jobs which exceed this duration will go on hold. Exiting the running state resets the job duration used by this command.

This command is intended to help minimize the time wasted by jobs which may erroneously run forever.

max_retries = <integer>
The maximum number of retries allowed for this job (must be non-negative). If the job fails (does not exit with the **success_exit_code** exit code) it will be retried up to **max_retries** times (unless retries are ceased because of the **retry_until** command). If **max_retries** is not defined, and either **retry_until** or **success_exit_code** is, the value of `DEFAULT_JOB_MAX_RETRIES` will be used for the maximum number of retries.

The combination of the **max_retries**, **retry_until**, and **success_exit_code** commands causes an appropriate `OnExitRemove` expression to be automatically generated. If retry command(s) and **on_exit_remove** are both defined, the `OnExitRemove` expression will be generated by OR'ing the expression specified in `OnExitRemove` and the expression generated by the retry commands.

retry_until <Integer | ClassAd Boolean Expression>
An integer value or boolean expression that prevents further retries from taking place, even if **max_retries** have not been exhausted. If **retry_until** is an integer, the job exiting with that exit code will cause retries to cease. If **retry_until** is a ClassAd expression, the expression evaluating to `True` will cause retries to cease. For example, if you only want to retry exit codes 17, 34, and 81:

```
max_retries = 5
retry_until = !member( ExitCode, {17, 34, 81} )
```

success_exit_code = <integer>
The exit code that is considered successful for this job. Defaults to 0 if not defined.

Note: non-zero values of success_exit_code should generally not be used for DAG node jobs, unless `when_to_transfer_output` is set to `ON_SUCCESS` in order to avoid failed jobs going on hold.

At the present time, *condor_dagman* does not take into account the value of **success_exit_code**. This means that, if **success_exit_code** is set to a non-zero value, *condor_dagman* will consider the job failed when it actually succeeds. For single-proc DAG node jobs, this can be overcome by using a POST script that takes into account the value of **success_exit_code** (although this is not recommended). For multi-proc DAG node jobs, there is currently no way to overcome this limitation.

checkpoint_exit_code = <integer>
The exit code which indicates that the executable has exited after successfully taking a checkpoint. The checkpoint will transferred and the executable restarted. See *Self-Checkpointing Applications* for details.

hold = <True | False>
If **hold** is set to `True`, then the submitted job will be placed into the Hold state. Jobs in the Hold state will not run until released by *condor_release*. Defaults to `False`.

keep_claim_idle = <integer>
An integer number of seconds that a job requests the *condor_schedd* to wait before releasing its claim after the job exits or after the job is removed.

The process by which the *condor_schedd* claims a *condor_startd* is somewhat time-consuming. To amortize this cost, the *condor_schedd* tries to reuse claims to run subsequent jobs, after a job using a claim is done. However, it can only do this if there is an idle job in the queue at the moment the previous job completes. Sometimes, and especially for the node jobs when using DAGMan, there is a subsequent job about to be submitted, but it has not yet arrived in the queue when the previous job completes. As a result, the *condor_schedd* releases the claim, and the next job must wait an entire negotiation cycle to start. When this submit command is defined with a non-negative integer, when the job exits, the *condor_schedd* tries as usual to reuse the claim. If it cannot, instead of releasing the claim, the *condor_schedd* keeps the claim until either the number of seconds given as a parameter, or a new job which matches that claim arrives, whichever comes first. The *condor_startd* in question will remain in the Claimed/Idle state, and the original job will be "charged" (in terms of priority) for the time in this state.

leave_in_queue = <ClassAd Boolean Expression>
When the ClassAd Expression evaluates to `True`, the job is not removed from the queue upon completion. This allows the user of a remotely spooled job to retrieve output files in cases where HTCondor would have removed them as part of the cleanup associated with completion. The job will only exit the queue once it has been marked for removal (via *condor_rm*, for example) and the **leave_in_queue** expression has become `False`. **leave_in_queue** defaults to `False`.

As an example, if the job is to be removed once the output is retrieved with *condor_transfer_data*, then use

```
leave_in_queue = (JobStatus == 4) && ((StageOutFinish =?= UNDEFINED) ||\
                 (StageOutFinish == 0))
```

next_job_start_delay = <ClassAd Boolean Expression>
This expression specifies the number of seconds to delay after starting up this job before the next job is started. The maximum allowed delay is specified by the HTCondor configuration variable `MAX_NEXT_JOB_START_DELAY` , which defaults to 10 minutes. This command does not apply to **scheduler** or **local** universe jobs.

This command has been historically used to implement a form of job start throttling from the job submitter's perspective. It was effective for the case of multiple job submission where the transfer of extremely large input data sets to the execute machine caused machine performance to suffer. This command is no longer useful, as throttling should be accomplished through configuration of the *condor_schedd* daemon.

on_exit_hold = <ClassAd Boolean Expression>
The ClassAd expression is checked when the job exits, and if `True`, places the job into the Hold state. If `False` (the default value when not defined), then nothing happens and the `on_exit_remove` expression is checked to determine if that needs to be applied.

For example: Suppose a job is known to run for a minimum of an hour. If the job exits after less than an hour, the job should be placed on hold and an e-mail notification sent, instead of being allowed to leave the queue.

```
on_exit_hold = (time() - JobStartDate) < (60 * $(MINUTE))
```

This expression places the job on hold if it exits for any reason before running for an hour. An e-mail will be sent to the user explaining that the job was placed on hold because this expression became `True`.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. This expression is available for the vanilla, java, parallel, grid, local and scheduler universes.

on_exit_hold_reason = <ClassAd String Expression>
When the job is placed on hold due to the **on_exit_hold** expression becoming `True`, this expression is evaluated to set the value of `HoldReason` in the job ClassAd. If this expression is `UNDEFINED` or produces an empty or invalid string, a default description is used.

on_exit_hold_subcode = <ClassAd Integer Expression>
When the job is placed on hold due to the **on_exit_hold** expression becoming `True`, this expression is evaluated to set the value of `HoldReasonSubCode` in the job ClassAd. The default subcode is 0. The `HoldReasonCode` will be set to 3, which indicates that the job went on hold due to a job policy expression.

on_exit_remove = <ClassAd Boolean Expression>
The ClassAd expression is checked when the job exits, and if `True` (the default value when undefined), then it allows the job to leave the queue normally. If `False`, then the job is placed back into the Idle state. If the user job runs under the vanilla universe, then the job restarts from the beginning.

For example, suppose a job occasionally segfaults, but chances are that the job will finish successfully if the job is run again with the same data. The **on_exit_remove** expression can cause the job to run again with the following command. Assume that the signal identifier for the segmentation fault is 11 on the platform where the job will be running.

```
on_exit_remove = (ExitBySignal == False) || (ExitSignal != 11)
```

This expression lets the job leave the queue if the job was not killed by a signal or if it was killed by a signal other than 11, representing segmentation fault in this example. So, if the exited due to signal

11, it will stay in the job queue. In any other case of the job exiting, the job will leave the queue as it normally would have done.

As another example, if the job should only leave the queue if it exited on its own with status 0, this **on_exit_remove** expression works well:

```
on_exit_remove = (ExitBySignal == False) && (ExitCode == 0)
```

If the job was killed by a signal or exited with a non-zero exit status, HTCondor would leave the job in the queue to run again.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression.

periodic_hold = <ClassAd Boolean Expression>
This expression is checked periodically when the job is not in the Held state. If it becomes `True`, the job will be placed on hold. If unspecified, the default value is `False`.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. Note that, by default, this expression is only checked once every 60 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL`, `MAX_PERIODIC_EXPR_INTERVAL`, and `PERIODIC_EXPR_TIMESLICE` configuration macros.

periodic_hold_reason = <ClassAd String Expression>
When the job is placed on hold due to the **periodic_hold** expression becoming `True`, this expression is evaluated to set the value of `HoldReason` in the job ClassAd. If this expression is `UNDEFINED` or produces an empty or invalid string, a default description is used.

periodic_hold_subcode = <ClassAd Integer Expression>
When the job is placed on hold due to the **periodic_hold** expression becoming true, this expression is evaluated to set the value of `HoldReasonSubCode` in the job ClassAd. The default subcode is 0. The `HoldReasonCode` will be set to 3, which indicates that the job went on hold due to a job policy expression.

periodic_release = <ClassAd Boolean Expression>
This expression is checked periodically when the job is in the Held state. If the expression becomes `True`, the job will be released. If the job was held via *condor_hold* (i.e. `HoldReasonCode` is 1), then this expression is ignored.

Only job ClassAd attributes will be defined for use by this ClassAd expression. Note that, by default, this expression is only checked once every 60 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL`, `MAX_PERIODIC_EXPR_INTERVAL`, and `PERIODIC_EXPR_TIMESLICE` configuration macros.

periodic_remove = <ClassAd Boolean Expression>
This expression is checked periodically. If it becomes `True`, the job is removed from the queue. If unspecified, the default value is `False`.

See the Examples section of this manual page for an example of a **periodic_remove** expression.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions. So, the `periodic_remove` expression takes precedent over the `on_exit_remove` expression, if the two describe conflicting actions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. Note that, by default, this expression is only checked once every 60 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL`, `MAX_PERIODIC_EXPR_INTERVAL`, and `PERIODIC_EXPR_TIMESLICE` configuration macros.

COMMANDS FOR THE GRID

arc_application = <XML-string>
For grid universe jobs of type **arc**, provides additional XML attributes under the <`Application`> section of the ARC ADL job description which are not covered by regular submit description file parameters.

arc_resources = <XML-string>
For grid universe jobs of type **arc**, provides additional XML attributes under the <`Resources`> section of the ARC ADL job description which are not covered by regular submit description file parameters.

arc_rte = < rte1 option,rte2 >
For grid universe jobs of type **arc**, provides a list of Runtime Environment names that the job requires on the ARC system. The list is comma-delimited. If a Runtime Environment name supports options, those can be provided after the name, separated by spaces. Runtime Environment names are defined by the ARC server.

azure_admin_key = <pathname>
For grid type **azure** jobs, specifies the path and file name of a file that contains an SSH public key. This key can be used to log into the administrator account of the instance via SSH.

azure_admin_username = <account name>
For grid type **azure** jobs, specifies the name of an administrator account to be created in the instance. This account can be logged into via SSH.

azure_auth_file = <pathname>
For grid type **azure** jobs, specifies a path and file name of the authorization file that grants permission for HTCondor to use the Azure account. If it's not defined, then HTCondor will attempt to use the default credentials of the Azure CLI tools.

azure_image = <image id>
For grid type **azure** jobs, identifies the disk image to be used for the boot disk of the instance. This image must already be registered within Azure.

azure_location = <image id>
For grid type **azure** jobs, identifies the location within Azure where the instance should be run. As an example, one current location is `centralus`.

azure_size = <machine type>
For grid type **azure** jobs, the hardware configuration that the virtual machine instance is to run on.

batch_extra_submit_args = <command-line arguments>
Used for **batch** grid universe jobs. Specifies additional command-line arguments to be given to the target batch system's job submission command.

batch_project = <projectname>
Used for **batch** grid universe jobs. Specifies the name of the PBS/LSF/SGE/SLURM project, account, or allocation that should be charged for the resources used by the job.

batch_queue = <queuename>
Used for **batch** grid universe jobs. Specifies the name of the PBS/LSF/SGE/SLURM job queue into which the job should be submitted. If not specified, the default queue is used. For a multi-cluster SLURM configuration, which cluster to use can be specified by supplying the name after an `@` symbol. For example, to submit a job to the `debug` queue on cluster `foo`, you would use the value `debug@foo`.

batch_runtime = <seconds>
Used for **batch** grid universe jobs. Specifies a limit in seconds on the execution time of the job. This limit is enforced by the PBS/LSF/SGE/SLURM scheduler.

cloud_label_names = <name0,name1,name...>
For grid type **gce** jobs, specifies the case of tag names that will be associated with the running instance. This is only necessary if a tag name case matters. By default the list will be automatically generated.

cloud_label_<name> = <value>
For grid type **gce** jobs, specifies a label and value to be associated with the running instance. The label name will be lower-cased; use **cloud_label_names** to change the case.

delegate_job_GSI_credentials_lifetime = <seconds>
Specifies the maximum number of seconds for which delegated proxies should be valid. The default behavior when this command is not specified is determined by the configuration variable `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME` , which defaults to one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. This setting currently only applies to proxies delegated for non-grid jobs and for HTCondor-C jobs. This variable has no effect if the configuration variable `DELEGATE_JOB_GSI_CREDENTIALS` is `False`, because in that case the job proxy is copied rather than delegated.

ec2_access_key_id = <pathname>
For grid type **ec2** jobs, identifies the file containing the access key.

ec2_ami_id = <EC2 xMI ID>
For grid type **ec2** jobs, identifies the machine image. Services compatible with the EC2 Query API may refer to these with abbreviations other than `AMI`, for example `EMI` is valid for Eucalyptus.

ec2_availability_zone = <zone name>
For grid type **ec2** jobs, specifies the Availability Zone that the instance should be run in. This command is optional, unless **ec2_ebs_volumes** is set. As an example, one current zone is `us-east-1b`.

ec2_block_device_mapping = <block-device>:<kernel-device>,<block-device>:<kernel-device>, ...
For grid type **ec2** jobs, specifies the block device to kernel device mapping. This command is optional.

ec2_ebs_volumes = <ebs name>:<device name>,<ebs name>:<device name>,...
For grid type **ec2** jobs, optionally specifies a list of Elastic Block Store (EBS) volumes to be made available to the instance and the device names they should have in the instance.

ec2_elastic_ip = <elastic IP address>
For grid type **ec2** jobs, and optional specification of an Elastic IP address that should be assigned to this instance.

ec2_iam_profile_arn = <IAM profile ARN>
For grid type **ec2** jobs, an Amazon Resource Name (ARN) identifying which Identity and Access Management (IAM) (instance) profile to associate with the instance.

ec2_iam_profile_name= <IAM profile name>
For grid type **ec2** jobs, a name identifying which Identity and Access Management (IAM) (instance) profile to associate with the instance.

ec2_instance_type = <instance type>
For grid type **ec2** jobs, identifies the instance type. Different services may offer different instance types, so no default value is set.

ec2_keypair = <ssh key-pair name>
For grid type **ec2** jobs, specifies the name of an SSH key-pair that is already registered with the EC2 service. The associated private key can be used to *ssh* into the virtual machine once it is running.

ec2_keypair_file = <pathname>
For grid type **ec2** jobs, specifies the complete path and file name of a file into which HTCondor will write an SSH key for use with ec2 jobs. The key can be used to *ssh* into the virtual machine once it is running. If **ec2_keypair** is specified for a job, **ec2_keypair_file** is ignored.

ec2_parameter_names = ParameterName1, ParameterName2, ...
For grid type **ec2** jobs, a space or comma separated list of the names of additional parameters to pass when instantiating an instance.

ec2_parameter_<name> = <value>
For grid type **ec2** jobs, specifies the value for the correspondingly named (instance instantiation) parameter. **<name>** is the parameter name specified in the submit command **ec2_parameter_names** , but with any periods replaced by underscores.

ec2_secret_access_key = <pathname>
For grid type **ec2** jobs, specifies the path and file name containing the secret access key.

ec2_security_groups = group1, group2, ...
For grid type **ec2** jobs, defines the list of EC2 security groups which should be associated with the job.

ec2_security_ids = id1, id2, ...
For grid type **ec2** jobs, defines the list of EC2 security group IDs which should be associated with the job.

ec2_spot_price = <bid>
For grid type **ec2** jobs, specifies the spot instance bid, which is the most that the job submitter is willing to pay per hour to run this job.

ec2_tag_names = <name0,name1,name...>
For grid type **ec2** jobs, specifies the case of tag names that will be associated with the running instance. This is only necessary if a tag name case matters. By default the list will be automatically generated.

ec2_tag_<name> = <value>
For grid type **ec2** jobs, specifies a tag to be associated with the running instance. The tag name will be lower-cased; use **ec2_tag_names** to change the case.

WantNameTag = <True | False>
For grid type **ec2** jobs, a job may request that its 'name' tag be (not) set by HTCondor. If the job does not otherwise specify any tags, not setting its name tag will eliminate a call by the EC2 GAHP, improving performance.

ec2_user_data = <data>
For grid type **ec2** jobs, provides a block of data that can be accessed by the virtual machine. If both **ec2_user_data** and **ec2_user_data_file** are specified for a job, the two blocks of data are concatenated, with the data from this **ec2_user_data** submit command occurring first.

ec2_user_data_file = <pathname>
For grid type **ec2** jobs, specifies a path and file name whose contents can be accessed by the virtual machine. If both **ec2_user_data** and **ec2_user_data_file** are specified for a job, the two blocks of data are concatenated, with the data from that **ec2_user_data** submit command occurring first.

ec2_vpc_ip = <a.b.c.d>
For grid type **ec2** jobs, that are part of a Virtual Private Cloud (VPC), an optional specification of the IP address that this instance should have within the VPC.

ec2_vpc_subnet = <subnet specification string>
For grid type **ec2** jobs, an optional specification of the Virtual Private Cloud (VPC) that this instance should be a part of.

gce_account = <account name>
For grid type **gce** jobs, specifies the Google cloud services account to use. If this submit command isn't specified, then a random account from the authorization file given by **gce_auth_file** will be used.

gce_auth_file = <pathname>
For grid type **gce** jobs, specifies a path and file name of the authorization file that grants permission for HTCondor to use the Google account. If this command is not specified, then the default file of the Google command-line tools will be used.

gce_image = <image id>
For grid type **gce** jobs, the identifier of the virtual machine image representing the HTCondor job to be run. This virtual machine image must already be register with GCE and reside in Google's Cloud Storage service.

gce_json_file = <pathname>
For grid type **gce** jobs, specifies the path and file name of a file that contains JSON elements that should be added to the instance description submitted to the GCE service.

gce_machine_type = <machine type>
For grid type **gce** jobs, the long form of the URL that describes the machine configuration that the virtual machine instance is to run on.

gce_metadata = <name=value,...,name=value>
For grid type **gce** jobs, a comma separated list of name and value pairs that define metadata for a virtual machine instance that is an HTCondor job.

gce_metadata_file = <pathname>
For grid type **gce** jobs, specifies a path and file name of the file that contains metadata for a virtual machine instance that is an HTCondor job. Within the file, each name and value pair is on its own line; so, the pairs are separated by the newline character.

gce_preemptible = <True | False>
For grid type **gce** jobs, specifies whether the virtual machine instance should be preemptible. The default is for the instance to not be preemptible.

grid_resource = <grid-type-string> <grid-specific-parameter-list>
For each **grid-type-string** value, there are further type-specific values that must specified. This submit description file command allows each to be given in a space-separated list. Allowable **grid-type-string** values are **arc**, **azure**, **batch**, **condor**, **ec2**, and **gce**. The HTCondor manual chapter on Grid Computing details the variety of grid types.

For a **grid-type-string** of **batch**, the single parameter is the name of the local batch system, and will be one of `pbs`, `lsf`, `slurm`, or `sge`.

For a **grid-type-string** of **condor**, the first parameter is the name of the remote *condor_schedd* daemon. The second parameter is the name of the pool to which the remote *condor_schedd* daemon belongs.

For a **grid-type-string** of **ec2**, one additional parameter specifies the EC2 URL.

For a **grid-type-string** of **arc**, the single parameter is the name of the ARC resource to be used.

transfer_error = <True | False>
For jobs submitted to the grid universe only. If `True`, then the error output (from `stderr`) from the job is transferred from the remote machine back to the access point. The name of the file after transfer is given by the **error** command. If `False`, no transfer takes place (from the remote machine to access point), and the name of the file is given by the **error** command. The default value is `True`.

transfer_input = <True | False>
For jobs submitted to the grid universe only. If `True`, then the job input (`stdin`) is transferred from the machine where the job was submitted to the remote machine. The name of the file that is transferred is given by the **input** command. If `False`, then the job's input is taken from a pre-staged file on the remote machine, and the name of the file is given by the **input** command. The default value is `True`.

For transferring files other than `stdin`, see **transfer_input_files** .

transfer_output = <True | False>
For jobs submitted to the grid universe only. If `True`, then the output (from `stdout`) from the job is transferred from the remote machine back to the access point. The name of the file after transfer is given by the **output** command. If `False`, no transfer takes place (from the remote machine to access point), and the name of the file is given by the **output** command. The default value is `True`.

For transferring files other than `stdout`, see **transfer_output_files** .

use_x509userproxy = <True | False>
Set this command to `True` to indicate that the job requires an X.509 user proxy. If **x509userproxy** is set, then that file is used for the proxy. Otherwise, the proxy is looked for in the standard locations. If **x509userproxy** is set or if the job is a grid universe job of grid type **arc**, then the value of **use_x509userproxy** is forced to `True`. Defaults to `False`.

x509userproxy = <full-pathname>
Used to override the default path name for X.509 user certificates. The default location for X.509 proxies is the `/tmp` directory, which is generally a local file system. Setting this value would allow HTCondor to access the proxy in a shared file system (for example, AFS). HTCondor will use the proxy specified in the submit description file first. If nothing is specified in the submit description file, it will use the environment variable X509_USER_PROXY. If that variable is not present, it will search in the default location. Note that proxies are only valid for a limited time. Condor_submit will not submit a job with an expired proxy, it will return an error. Also, if the configuration parameter CRED_MIN_TIME_LEFT is set to some number of seconds, and if the proxy will expire before that many seconds, condor_submit will also refuse to submit the job. That is, if CRED_MIN_TIME_LEFT is set to 60, condor_submit will refuse to submit a job whose proxy will expire 60 seconds from the time of submission.

x509userproxy is relevant when the **universe** is **vanilla**, or when the **universe** is **grid** and the type of grid system is one of **condor**, or **arc**. Defining a value causes the proxy to be delegated to the execute machine. Further, VOMS attributes defined in the proxy will appear in the job ClassAd.

use_scitokens = <True | False | Auto>
Set this command to `True` to indicate that the job requires a scitoken. If **scitokens_file** is set, then that file is used for the scitoken filename. Otherwise, the the scitoken filename is looked for in the `BEARER_TOKEN_FILE` environment variable. If **scitokens_file** is set then the value of **use_scitokens** defaults to `True`. If the filename is not defined in on one of these two places, then *condor_submit* will fail with an error message. Set this command to `Auto` to indicate that the job will use a scitoken if **scitokens_file** or the `BEARER_TOKEN_FILE` environment variable is set, but it will not be an error if no file is specified.

scitokens_file = <full-pathname>
Used to set the path to the file containing the scitoken that the job needs, or to override the path to the scitoken contained in the `BEARER_TOKEN_FILE` environment variable.

COMMANDS FOR PARALLEL, JAVA, and SCHEDULER UNIVERSES

hold_kill_sig = <signal-number>
For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is put on hold with *condor_hold*. **signal-number** may be either the platform-specific name or value of the signal. If this command is not present, the value of **kill_sig** is used.

jar_files = <file_list>
Specifies a list of additional JAR files to include when using the Java universe. JAR files will be transferred along with the executable and automatically added to the classpath.

java_vm_args = <argument_list>
Specifies a list of additional arguments to the Java VM itself, When HTCondor runs the Java program, these are the arguments that go before the class name. This can be used to set VM-specific arguments like stack size, garbage-collector arguments and initial property values.

machine_count = <max>
For the parallel universe, a single value (*max*) is required. It is neither a maximum or minimum, but the number of machines to be dedicated toward running the job.

remove_kill_sig = <signal-number>
For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is removed with *condor_rm*. **signal-number** may be either the platform-specific name or value of the signal. This example shows it both ways for a Linux signal:

```
remove_kill_sig = SIGUSR1
remove_kill_sig = 10
```

If this command is not present, the value of **kill_sig** is used.

COMMANDS FOR THE VM UNIVERSE

vm_disk = file1:device1:permission1, file2:device2:permission2:format2, ...
A list of comma separated disk files. Each disk file is specified by 4 colon separated fields. The first field is the path and file name of the disk file. The second field specifies the device. The third field specifies permissions, and the optional fourth field specifies the image format. If a disk file will be transferred by HTCondor, then the first field should just be the simple file name (no path information).

An example that specifies two disk files:

```
vm_disk = /myxen/diskfile.img:sda1:w,/myxen/swap.img:sda2:w
```

vm_checkpoint = <True | False>
A boolean value specifying whether or not to take checkpoints. If not specified, the default value is `False`. In the current implementation, setting both **vm_checkpoint** and **vm_networking** to `True` does not yet work in all cases. Networking cannot be used if a vm universe job uses a checkpoint in order to continue execution after migration to another machine.

vm_macaddr = <MACAddr>
Defines that MAC address that the virtual machine's network interface should have, in the standard format of six groups of two hexadecimal digits separated by colons.

vm_memory = <MBytes-of-memory>
The amount of memory in MBytes that a vm universe job requires.

vm_networking = <True | False>
Specifies whether to use networking or not. In the current implementation, setting both **vm_checkpoint** and **vm_networking** to `True` does not yet work in all cases. Networking cannot be used if a vm universe job uses a checkpoint in order to continue execution after migration to another machine.

vm_networking_type = <nat | bridge >
When **vm_networking** is `True`, this definition augments the job's requirements to match only machines with the specified networking. If not specified, then either networking type matches.

vm_no_output_vm = <True | False>
When `True`, prevents HTCondor from transferring output files back to the machine from which the vm universe job was submitted. If not specified, the default value is `False`.

vm_type = <xen | kvm>
Specifies the underlying virtual machine software that this job expects.

xen_initrd = <image-file>
When **xen_kernel** gives a file name for the kernel image to use, this optional command may specify a path to a ramdisk (`initrd`) image file. If the image file will be transferred by HTCondor, then the value should just be the simple file name (no path information).

xen_kernel = <included | path-to-kernel>
A value of **included** specifies that the kernel is included in the disk file. If not one of these values, then the value is a path and file name of the kernel to be used. If a kernel file will be transferred by HTCondor, then the value should just be the simple file name (no path information).

xen_kernel_params = <string>
A string that is appended to the Xen kernel command line.

xen_root = <string>
A string that is appended to the Xen kernel command line to specify the root device. This string is required when **xen_kernel** gives a path to a kernel. Omission for this required case results in an error message during submission.

COMMANDS FOR THE DOCKER UNIVERSE

docker_image = < image-name >
Defines the name of the Docker image that is the basis for the docker container.

docker_network_type = < host | none | custom_admin_defined_value>
If docker_network_type is set to the string host, then the job is run using the host's network. If docker_network_type is set to the string none, then the job is run with no network. If this is not set, each job gets a private network interface. Some administrators may define site specific docker networks on a given worker node. When this is the case, additional values may be valid here.

docker_pull_policy = < always >
if docker_pull_policy is set to *always*, when a docker universe job starts on a worker node, the option "–pull always" will be passed to the docker run command. This only impacts worker nodes which already have a locally cached version of the image. With this option, docker will always check with the repo to see if the cached version is out of date. This requires more network connectivity, and may cause docker hub to throttle future pull requests. It is generally recommened to never mutate docker image tag name, and avoid needing this option.

container_service_names = <service-name>[, <service-name>]*
A string- or comma- separated list of *service name*s. Each *service-name* must have a corresponding `<service-name>_container_port` command specifying a port number (an integer from 0 to 65535). HTCondor will ask Docker to forward from a host port to the specified port inside the container. When Docker has done so, HTCondor will add an attribute to the job ad for each service,

`<service-name>HostPort`, which contains the port number on the host forwarding to the corresponding service.

COMMANDS FOR THE CONTAINER UNIVERSE

container_image = < image-name >
Defines the name of the container image. Can be a singularity .sif file, a singularity exploded directory, or a path to an image in a docker style repository

container_target_dir = < path-to-directory-inside-container >
Defines the working directory of the job inside the container. Will be mapped to the scratch directory on the worker node.

ADVANCED COMMANDS

accounting_group = <accounting-group-name>
Causes jobs to negotiate under the given accounting group. This value is advertised in the job ClassAd as `AcctGroup`. The HTCondor Administrator's manual contains more information about accounting groups.

accounting_group_user = <accounting-group-user-name>
Sets the name associated with this job to be used for resource usage accounting purposes, such as computation of fair-share priority and reporting via `condor_userprio`. If not set, defaults to the value of the job ClassAd attribute `User`. This value is advertised in the job ClassAd as `AcctGroupUser`.

concurrency_limits = <string-list>
A list of resources that this job needs. The resources are presumed to have concurrency limits placed upon them, thereby limiting the number of concurrent jobs in execution which need the named resource. Commas and space characters delimit the items in the list. Each item in the list is a string that identifies the limit, or it is a ClassAd expression that evaluates to a string, and it is evaluated in the context of machine ClassAd being considered as a match. Each item in the list also may specify a numerical value identifying the integer number of resources required for the job. The syntax follows the resource name by a colon character (:) and the numerical value. Details on concurrency limits are in the HTCondor Administrator's manual.

concurrency_limits_expr = <ClassAd String Expression>
A ClassAd expression that represents the list of resources that this job needs after evaluation. The ClassAd expression may specify machine ClassAd attributes that are evaluated against a matched machine. After evaluation, the list sets **concurrency_limits**.

copy_to_spool = <True | False>
If **copy_to_spool** is `True`, then *condor_submit* copies the executable to the local spool directory before running it on a remote host. As copying can be quite time consuming and unnecessary, the default value is `False` for all job universes. When `False`, *condor_submit* does not copy the executable to a local spool directory.

coresize = <size>
Should the user's program abort and produce a core file, **coresize** specifies the maximum size in bytes of the core file which the user wishes to keep. If **coresize** is not specified in the command file, this is set to 0 (meaning no core will be generated).

cron_day_of_month = <Cron-evaluated Day>
The set of days of the month for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_day_of_week = <Cron-evaluated Day>
The set of days of the week for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_hour = <Cron-evaluated Hour>
The set of hours of the day for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_minute = <Cron-evaluated Minute>
The set of minutes within an hour for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_month = <Cron-evaluated Month>
The set of months within a year for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_prep_time = <ClassAd Integer Expression>
Analogous to **deferral_prep_time** . The number of seconds prior to a job's deferral time that the job may be matched and sent to an execution machine.

cron_window = <ClassAd Integer Expression>
Analogous to the submit command **deferral_window** . It allows cron jobs that miss their deferral time to begin execution.

The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

dagman_log = <pathname>
DAGMan inserts this command to specify an event log that it watches to maintain the state of the DAG. If the **log** command is not specified in the submit file, DAGMan uses the **log** command to specify the event log.

deferral_prep_time = <ClassAd Integer Expression>
The number of seconds prior to a job's deferral time that the job may be matched and sent to an execution machine.

The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

deferral_time = <ClassAd Integer Expression>
Allows a job to specify the time at which its execution is to begin, instead of beginning execution as soon as it arrives at the execution machine. The deferral time is an expression that evaluates to a Unix Epoch timestamp (the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time). Deferral time is evaluated with respect to the execution machine. This option delays the start of execution, but not the matching and claiming of a machine for the job. If the job is not available and ready to begin execution at the deferral time, it has missed its deferral time. A job that misses its deferral time will be put on hold in the queue.

The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

Due to implementation details, a deferral time may not be used for scheduler universe jobs.

deferral_window = <ClassAd Integer Expression>
The deferral window is used in conjunction with the **deferral_time** command to allow jobs that miss their deferral time to begin execution.

The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

description = <string>
A string that sets the value of the job ClassAd attribute `JobDescription`. When set, tools which display the executable such as *condor_q* will instead use this string.

email_attributes = <list-of-job-ad-attributes>
A comma-separated list of attributes from the job ClassAd. These attributes and their values will be included in the e-mail notification of job completion.

image_size = <size>
Advice to HTCondor specifying the maximum virtual image size to which the job will grow during its execution. HTCondor will then execute the job only on machines which have enough resources, (such as virtual memory), to support executing the job. If not specified, HTCondor will automatically make a (reasonably accurate) estimate about the job's size and adjust this estimate as the program runs. If specified and underestimated, the job may crash due to the inability to acquire more address space; for example, if malloc() fails. If the image size is overestimated, HTCondor may have difficulty finding machines which have the required resources. *size* is specified in KiB. For example, for an image size of 8 MiB, *size* should be 8000.

initialdir = <directory-path>
Used to give jobs a directory with respect to file input and output. Also provides a directory (on the machine from which the job is submitted) for the job event log, when a full path is not specified.

For vanilla universe jobs where there is a shared file system, it is the current working directory on the machine where the job is executed.

For vanilla or grid universe jobs where file transfer mechanisms are utilized (there is not a shared file system), it is the directory on the machine from which the job is submitted where the input files come from, and where the job's output files go to.

For scheduler universe jobs, it is the directory on the machine from which the job is submitted where the job runs; the current working directory for file input and output with respect to relative path

names.

Note that the path to the executable is not relative to **initialdir** ; if it is a relative path, it is relative to the directory in which the *condor_submit* command is run.

job_ad_information_attrs = <attribute-list>
A comma-separated list of job ClassAd attribute names. The named attributes and their values are written to the job event log whenever any event is being written to the log. This implements the same thing as the configuration variable `EVENT_LOG_INFORMATION_ATTRS` (see the *Daemon Logging Configuration File Entries* page), but it applies to the job event log, instead of the system event log.

job_lease_duration = <number-of-seconds>
For vanilla, parallel, VM, and java universe jobs only, the duration in seconds of a job lease. The default value is 2,400, or forty minutes. If a job lease is not desired, the value can be explicitly set to 0 to disable the job lease semantics. The value can also be a ClassAd expression that evaluates to an integer. The HTCondor User's manual section on Special Environment Considerations has further details.

job_machine_attrs = <attr1, attr2, ...>
A comma and/or space separated list of machine attribute names that should be recorded in the job ClassAd in addition to the ones specified by the *condor_schedd* daemon's system configuration variable `SYSTEM_JOB_MACHINE_ATTRS` . When there are multiple run attempts, history of machine attributes from previous run attempts may be kept. The number of run attempts to store may be extended beyond the system-specified history length by using the submit file command **job_machine_attrs_history_length** . A machine attribute named `X` will be inserted into the job ClassAd as an attribute named `MachineAttrX0`. The previous value of this attribute will be named `MachineAttrX1`, the previous to that will be named `MachineAttrX2`, and so on, up to the specified history length. A history of length 1 means that only `MachineAttrX0` will be recorded. The value recorded in the job ClassAd is the evaluation of the machine attribute in the context of the job ClassAd when the *condor_schedd* daemon initiates the start up of the job. If the evaluation results in an `Undefined` or `Error` result, the value recorded in the job ad will be `Undefined` or `Error`, respectively.

want_graceful_removal = <boolean expression>
If `true`, this job will be given a chance to shut down cleanly when removed. The job will be given as much time as the administrator of the execute resource allows, which may be none. The default is `false`. For details, see the configuration setting *GRACEFULLY_REMOVE_JOBS*.

kill_sig = <signal-number>
When HTCondor needs to kick a job off of a machine, it will send the job the signal specified by **signal-number** . **signal-number** needs to be an integer which represents a valid signal on the execution machine. The default value is SIGTERM, which is the standard way to terminate a program in Unix.

kill_sig_timeout = <seconds>
This submit command should no longer be used as of HTCondor version 7.7.3; use **job_max_vacate_time** instead. If **job_max_vacate_time** is not defined, this defines the number

of seconds that HTCondor should wait following the sending of the kill signal defined by **kill_sig** and forcibly killing the job. The actual amount of time between sending the signal and forcibly killing the job is the smallest of this value and the configuration variable `KILLING_TIMEOUT` , as defined on the execute machine.

load_profile = <True | False>
When `True`, loads the account profile of the dedicated run account for Windows jobs. May not be used with **run_as_owner** .

log_xml = <True | False>
If **log_xml** is `True`, then the job event log file will be written in ClassAd XML. If not specified, XML is not used. Note that the file is an XML fragment; it is missing the file header and footer. Do not mix XML and non-XML within a single file. If multiple jobs write to a single job event log file, ensure that all of the jobs specify this option in the same way.

match_list_length = <integer value>
Defaults to the value zero (0). When **match_list_length** is defined with an integer value greater than zero (0), attributes are inserted into the job ClassAd. The maximum number of attributes defined is given by the integer value. The job ClassAds introduced are given as

```
LastMatchName0 = "most-recent-Name"
LastMatchName1 = "next-most-recent-Name"
```

The value for each introduced ClassAd is given by the value of the `Name` attribute from the machine ClassAd of a previous execution (match). As a job is matched, the definitions for these attributes will roll, with LastMatchName1 becoming LastMatchName2, LastMatchName0 becoming LastMatchName1, and LastMatchName0 being set by the most recent value of the `Name` attribute.

An intended use of these job attributes is in the requirements expression. The requirements can allow a job to prefer a match with either the same or a different resource than a previous match.

job_max_vacate_time = <integer expression>
An integer-valued expression (in seconds) that may be used to adjust the time given to an evicted job for gracefully shutting down. If the job's setting is less than the machine's, the job's is used. If the job's setting is larger than the machine's, the result depends on whether the job has any excess retirement time. If the job has more retirement time left than the machine's max vacate time setting, then retirement time will be converted into vacating time, up to the amount requested by the job.

Setting this expression does not affect the job's resource requirements or preferences. For a job to only run on a machine with a minimum `MachineMaxVacateTime`, or to preferentially run on such machines, explicitly specify this in the requirements and/or rank expressions.

manifest = <True | False>
For vanilla and Docker -universe jobs (and others that use the shadow), specifies if HTCondor (the starter) should produce a "manifest", which is directory containing three files: the list of files and directories at the top level of the sandbox when file transfer in completes (`in`), the same when file transfer out begins (`out`), and a dump of the environment set for the job (`env`).

This feature is not presently available for Windows.

manifest_dir = <directory name>
For vanilla and Docker -universe jobs (and others that use the shadow), specifies the directory in which to record the manifest. Specifying this enables the creation of a manifest. By default, the manifest directory is named <cluster>_<proc>_manifest, to avoid conflicts.

This feature is not presently available for Windows.

max_job_retirement_time = <integer expression>
An integer-valued expression (in seconds) that does nothing unless the machine that runs the job has been configured to provide retirement time. Retirement time is a grace period given to a job to finish when a resource claim is about to be preempted. The default behavior in many cases is to take as much retirement time as the machine offers, so this command will rarely appear in a submit description file.

When a resource claim is to be preempted, this expression in the submit file specifies the maximum run time of the job (in seconds, since the job started). This expression has no effect, if it is greater than the maximum retirement time provided by the machine policy. If the resource claim is not preempted, this expression and the machine retirement policy are irrelevant. If the resource claim is preempted the job will be allowed to run until the retirement time expires, at which point it is hard-killed. The job will be soft-killed when it is getting close to the end of retirement in order to give it time to gracefully shut down. The amount of lead-time for soft-killing is determined by the maximum vacating time granted to the job.

Any jobs running with **nice_user** priority have a default **max_job_retirement_time** of 0, so no retirement time is utilized by default. In all other cases, no default value is provided, so the maximum amount of retirement time is utilized by default.

Setting this expression does not affect the job's resource requirements or preferences. For a job to only run on a machine with a minimum `MaxJobRetirementTime`, or to preferentially run on such machines, explicitly specify this in the requirements and/or rank expressions.

nice_user = <True | False>
Normally, when a machine becomes available to HTCondor, HTCondor decides which job to run based upon user and job priorities. Setting **nice_user** equal to `True` tells HTCondor not to use your regular user priority, but that this job should have last priority among all users and all jobs. So jobs submitted in this fashion run only on machines which no other non-nice_user job wants - a true bottom-feeder job! This is very handy if a user has some jobs they wish to run, but do not wish to use resources that could instead be used to run other people's HTCondor jobs. Jobs submitted in this fashion have an accounting group. The accounting group is configurable by setting `NICE_USER_ACCOUNTING_GROUP_NAME` which defaults to `nice-user` The default value is `False`.

noop_job = <ClassAd Boolean Expression>
When this boolean expression is `True`, the job is immediately removed from the queue, and HTCondor makes no attempt at running the job. The log file for the job will show a job submitted event and a job terminated event, along with an exit code of 0, unless the user specifies a different signal or exit code.

noop_job_exit_code = <return value>
When **noop_job** is in the submit description file and evaluates to `True`, this command allows the job to specify the return value as shown in the job's log file job terminated event. If not specified, the job will show as having terminated with status 0. This overrides any value specified with **noop_job_exit_signal** .

noop_job_exit_signal = <signal number>
When **noop_job** is in the submit description file and evaluates to `True`, this command allows the job to specify the signal number that the job's log event will show the job having terminated with.

remote_initialdir = <directory-path>
The path specifies the directory in which the job is to be executed on the remote machine.

rendezvousdir = <directory-path>
Used to specify the shared file system directory to be used for file system authentication when submitting to a remote scheduler. Should be a path to a preexisting directory.

run_as_owner = <True | False>
A boolean value that causes the job to be run under the login of the submitter, if supported by the joint configuration of the submit and execute machines. On Unix platforms, this defaults to `True`, and on Windows platforms, it defaults to `False`. May not be used with **load_profile** . See the HTCondor manual Platform-Specific Information chapter for administrative details on configuring Windows to support this option.

stack_size = <size in bytes>
This command applies only to Linux platforms. An integer number of bytes, representing the amount of stack space to be allocated for the job. This value replaces the default allocation of stack space, which is unlimited in size.

submit_event_notes = <note>
A string that is appended to the submit event in the job's log file. For DAGMan jobs, the string `DAG Node:` and the node's name is automatically defined for **submit_event_notes**, causing the logged submit event to identify the DAG node job submitted.

ulog_execute_attrs = <attribute-list>
A comma-separated list of machine ClassAd attribute names. The named attributes and their values are written as part of the execution event in the job event log.

use_oauth_services = <list of credential service names>
A comma-separated list of credential-providing service names for which the job should be provided credentials for the job execution environment. The credential service providers must be configured by the pool admin.

<credential_service_name>_oauth_permissions[_<handle>] = <scope>
A string containing the scope(s) that should be requested for the credential named <credential_service_name>[_<handle>], where <handle> is optionally provided to differentiate between multiple credentials from the same credential service provider.

<credential_service_name>_oauth_resource[_<handle>] = <resource>
A string containing the resource (or "audience") that should be requested for the credential named <credential_service_name>[_<handle>], where <handle> is optionally provided to differentiate between multiple credentials from the same credential service provider.

MY.<attribute> = <value> or +<attribute> = <value>
A macro that begins with MY. or a line that begins with a '+' (plus) character instructs *condor_submit* to insert the given *attribute* (without + or MY.) into the job ClassAd with the given *value*. The macro can be referenced in other submit statements by using `$(MY.<attribute>)`. A +<attribute> is converted to MY.<attribute> when the file is read.

Note that setting an job attribute in this way should not be used in place of one of the specific commands listed above. Often, the command name does not directly correspond to an attribute name; furthermore, many submit commands result in actions more complex than simply setting an attribute or attributes. See *Job ClassAd Attributes* for a list of HTCondor job attributes.

MACROS AND COMMENTS

In addition to commands, the submit description file can contain macros and comments.

Macros
Parameterless macros in the form of `$(macro_name:default initial value)` may be used anywhere in HTCondor submit description files to provide textual substitution at submit time. Macros can be defined by lines in the form of

```
<macro_name> = <string>
```

Several pre-defined macros are supplied by the submit description file parser. The `$(Cluster)` or `$(ClusterId)` macro supplies the value of the `ClusterId` job ClassAd attribute, and the `$(Process)` or `$(ProcId)` macro supplies the value of the `ProcId` job ClassAd attribute. The `$(JobId)` macro supplies the full job id. It is equivalent to `$(ClusterId).$(ProcId)`. These macros are intended to aid in the specification of input/output files, arguments, etc., for clusters with lots of jobs, and/or could be used to supply an HTCondor process with its own cluster and process numbers on the command line.

The `$(Node)` macro is defined for parallel universe jobs, and is especially relevant for MPI applications. It is a unique value assigned for the duration of the job that essentially identifies the machine (slot) on which a program is executing. Values assigned start at 0 and increase monotonically. The values are assigned as the parallel job is about to start.

Recursive definition of macros is permitted. An example of a construction that works is the following:

```
foo = bar
foo =  snap $(foo)
```

As a result, `foo = snap bar`.

Note that both left- and right- recursion works, so

```
foo = bar
foo =  $(foo) snap
```

has as its result `foo = bar snap`.

The construction

```
foo = $(foo) bar
```

by itself will not work, as it does not have an initial base case. Mutually recursive constructions such as:

```
B = bar
C = $(B)
B = $(C) boo
```

will not work, and will fill memory with expansions.

A default value may be specified, for use if the macro has no definition. Consider the example

```
D = $(E:24)
```

Where E is not defined within the submit description file, the default value 24 is used, resulting in

```
D = 24
```

This is useful for creating submit templates where values can be passed on the *condor_submit* command line, but that have a default value as well. In the above example, if you give a value for E on the command line like this

```
condor_submit E=99 <submit-file>
```

The value of 99 is used for E, resulting in

```
D = 99
```

To use the dollar sign character ($) as a literal, without macro expansion, use

```
$(DOLLAR)
```

In addition to the normal macro, there is also a special kind of macro called a substitution macro that allows the substitution of a machine ClassAd attribute value defined on the resource machine itself (gotten after a match to the machine has been made) into specific commands within the submit description file. The substitution macro is of the form:

```
$$(attribute)
```

As this form of the substitution macro is only evaluated within the context of the machine ClassAd, use of a scope resolution prefix `TARGET.` or `MY.` is not allowed.

A common use of this form of the substitution macro is for the heterogeneous submission of an executable:

```
executable = povray.$$(OpSys).$$(Arch)
```

Values for the `OpSys` and `Arch` attributes are substituted at match time for any given resource. This example allows HTCondor to automatically choose the correct executable for the matched machine.

An extension to the syntax of the substitution macro provides an alternative string to use if the machine attribute within the substitution macro is undefined. The syntax appears as:

```
$$(attribute:string_if_attribute_undefined)
```

An example using this extended syntax provides a path name to a required input file. Since the file can be placed in different locations on different machines, the file's path name is given as an argument to the program.

```
arguments = $$(input_file_path:/usr/foo)
```

On the machine, if the attribute `input_file_path` is not defined, then the path `/usr/foo` is used instead.

As a special case that only works within the submit file *environment* command, the string $$(CondorScratchDir) is expanded to the value of the job's scratch directory. This does not work for scheduler universe or grid universe jobs.

For example, to set PYTHONPATH to a subdirectory of the job scratch dir, one could set

```
environment = PYTHONPATH=$$(CondorScratchDir)/some/directory
```

A further extension to the syntax of the substitution macro allows the evaluation of a ClassAd expression to define the value. In this form, the expression may refer to machine attributes by prefacing them with the `TARGET.` scope resolution prefix. To place a ClassAd expression into the substitution macro, square brackets are added to delimit the expression. The syntax appears as:

```
$$([ClassAd expression])
```

An example of a job that uses this syntax may be one that wants to know how much memory it can use. The application cannot detect this itself, as it would potentially use all of the memory on a multi-slot machine. So the job determines the memory per slot, reducing it by 10% to account for miscellaneous overhead, and passes this as a command line argument to the application. In the submit description file will be

```
arguments = --memory $$([TARGET.Memory * 0.9])
```

To insert two dollar sign characters ($$) as literals into a ClassAd string, use

```
$$(DOLLARDOLLAR)
```

The environment macro, $ENV, allows the evaluation of an environment variable to be used in setting a submit description file command. The syntax used is

```
$ENV(variable)
```

An example submit description file command that uses this functionality evaluates the submitter's home directory in order to set the path and file name of a log file:

```
log = $ENV(HOME)/jobs/logfile
```

The environment variable is evaluated when the submit description file is processed.

The $RANDOM_CHOICE macro allows a random choice to be made from a given list of parameters at submission time. For an expression, if some randomness needs to be generated, the macro may appear as

```
$RANDOM_CHOICE(0,1,2,3,4,5,6)
```

When evaluated, one of the parameters values will be chosen.

Comments

Blank lines and lines beginning with a pound sign ('#') character are ignored by the submit description file parser.

15.56.5 Submit Variables

While processing the **queue** command in a submit file or from the command line, *condor_submit* will set the values of several automatic submit variables so that they can be referred to by statements in the submit file. With the exception of Cluster and Process, if these variables are set by the submit file, they will not be modified during **queue** processing.

ClusterId
Set to the integer value that the `ClusterId` attribute that the job ClassAd will have when the job is submitted. All jobs in a single submit will normally have the same value for the `ClusterId`. If the **-dry-run** argument is specified, The value will be 1.

Cluster
Alternate name for the ClusterId submit variable. Before HTCondor version 8.4 this was the only name.

ProcId
Set to the integer value that the `ProcId` attribute of the job ClassAd will have when the job is submitted. The value will start at 0 and increment by 1 for each job submitted.

Process
Alternate name for the ProcId submit variable. Before HTCondor version 8.4 this was the only name.

JobId
Set to `$(ClusterId).$(ProcId)` so that it will expand to the full id of the job.

Node
For parallel universes, set to the value #pArAlLeLnOdE# or #MpInOdE# depending on the parallel universe type For other universes it is set to nothing.

Step
Set to the step value as it varies from 0 to N-1 where N is the number provided on the **queue** argument. This variable changes at the same rate as ProcId when it changes at all. For submit files that don't make use of the queue number option, Step will always be 0. For submit files that don't make use of any of the foreach options, Step and ProcId will always be the same.

ItemIndex
Set to the index within the item list being processed by the various queue foreach options. For submit files that don't make use of any queue foreach list, ItemIndex will always be 0 For submit files that make use of a slice to select only some items in a foreach list, ItemIndex will only be set to selected values.

Row
Alternate name for ItemIndex.

Item
when a queue foreach option is used and no variable list is supplied, this variable will be set to the value of the current item.

The automatic variables below are set before parsing the submit file, and will not vary during processing unless the submit file itself sets them.

ARCH
Set to the CPU architecture of the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYS
Set to the name of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYSANDVER
Set to the name and major version of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYSMAJORVER
Set to the major version of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYSVER
Set to the version of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

SPOOL
Set to the full path of the HTCondor spool directory. The value will be the same as the automatic configuration variable of the same name.

IsLinux
Set to true if the operating system of the machine running *condor_submit* is a Linux variant. Set to false otherwise.

IsWindows
Set to true if the operating system of the machine running *condor_submit* is a Microsoft Windows variant. Set to false otherwise.

SUBMIT_FILE
Set to the full pathname of the submit file being processed by *condor_submit*. If submit statements are read from standard input, it is set to nothing.

SUBMIT_TIME
Set to the unix timestamp of the current time when the job is submitted.

YEAR
Set to the 4 digit year when the job is submitted.

MONTH
Set to the 2 digit month when the job is submitted.

DAY
Set to the 2 digit day when the job is submitted.

15.56.6 Exit Status

condor_submit will exit with a status value of 0 (zero) upon success, and a non-zero value upon failure.

15.56.7 Examples

- Submit Description File Example 1: This example queues three jobs for execution by HTCondor. The first will be given command line arguments of *15* and *2000*, and it will write its standard output to `foo.out1`. The second will be given command line arguments of *30* and *2000*, and it will write its standard output to `foo.out2`. Similarly the third will have arguments of *45* and *6000*, and it will use `foo.out3` for its standard output. Standard error output (if any) from all three programs will appear in `foo.error`.

```
####################
#
# submit description file
# Example 1: queuing multiple jobs with differing
# command line arguments and output files.
```

(continues on next page)

(continued from previous page)

```
#
####################

Executable     = foo
Universe       = vanilla

Arguments      = 15 2000
Output  = foo.out0
Error   = foo.err0
Queue

Arguments      = 30 2000
Output  = foo.out1
Error   = foo.err1
Queue

Arguments      = 45 6000
Output  = foo.out2
Error   = foo.err2
Queue
```

Or you can get the same results as the above submit file by using a list of arguments with the Queue statement

```
####################
#
# submit description file
# Example 1b: queuing multiple jobs with differing
# command line arguments and output files, alternate syntax
#
####################

Executable     = foo
Universe       = vanilla

# generate different output and error filenames for each process
Output  = foo.out$(Process)
Error   = foo.err$(Process)

Queue Arguments From (
  15 2000
  30 2000
  45 6000
)
```

- Submit Description File Example 2: This submit description file example queues 150 runs of program *foo* which must have been compiled and linked for an Intel x86 processor running RHEL 3. HTCondor will not attempt to run the processes on machines which have less than 32 Megabytes of physical memory, and it will run them on machines which have at least 64 Megabytes, if such machines are available. Stdin, stdout, and stderr will refer to `in.0`, `out.0`, and `err.0` for the first run of this program (process 0). Stdin, stdout, and stderr will refer to `in.1`, `out.1`, and `err.1` for process 1, and so forth. A log file containing entries about where and when HTCondor runs, transfers file, if it's evicted, and when it terminates, among other things, the various processes in this cluster will be written into file `foo.log`.

```
####################
#
# Example 2: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
Universe       = vanilla
Requirements   = OpSys == "LINUX" && Arch =="INTEL"
Rank           = Memory >= 64
Request_Memory = 32 Mb
Image_Size     = 28 Mb

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log
Queue 150
```

- Submit Description File Example 3: This example targets the */bin/sleep* program to run only on a platform running a RHEL 6 operating system. The example presumes that the pool contains machines running more than one version of Linux, and this job needs the particular operating system to run correctly.

```
####################
#
# Example 3: Run on a RedHat 6 machine
#
####################
Universe     = vanilla
Executable   = /bin/sleep
Arguments    = 30
Requirements = (OpSysAndVer == "RedHat6")

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log     = sleep.log
Queue
```

- Command Line example: The following command uses the **-append** option to add two commands before the job(s) is queued. A log file and an error log file are specified. The submit description file is unchanged.

```
$ condor_submit -a "log = out.log" -a "error = error.log" mysubmitfile
```

  Note that each of the added commands is contained within quote marks because there are space characters within the command.

- `periodic_remove` example: A job should be removed from the queue, if the total suspension time of the job is more than half of the run time of the job.

  Including the command

```
periodic_remove = CumulativeSuspensionTime >
                  ((RemoteWallClockTime - CumulativeSuspensionTime) / 2.0)
```

in the submit description file causes this to happen.

15.56.8 General Remarks

- For security reasons, HTCondor will refuse to run any jobs submitted by user root (UID = 0) or by a user whose default group is group wheel (GID = 0). Jobs submitted by user root or a user with a default group of wheel will appear to sit forever in the queue in an idle state.
- All path names specified in the submit description file must be less than 256 characters in length, and command line arguments must be less than 4096 characters in length; otherwise, *condor_submit* gives a warning message but the jobs will not execute properly.
- Somewhat understandably, behavior gets bizarre if the user makes the mistake of requesting multiple HTCondor jobs to write to the same file, and/or if the user alters any files that need to be accessed by an HTCondor job which is still in the queue. For example, the compressing of data or output files before an HTCondor job has completed is a common mistake.

15.56.9 See Also

HTCondor User Manual

15.57 *condor_submit_dag*

Manage and queue jobs within a specified DAG for execution on remote machines

15.57.1 Synopsis

condor_submit_dag [**-help | -version**]

condor_submit_dag [**-no_submit**] [**-verbose**] [**-force**] [**-dagman** *DagmanExecutable*] [**-maxidle** *NumberOfProcs*] [**-maxjobs** *NumberOfClusters*] [**-maxpre** *NumberOfPreScripts*] [**-maxpost** *NumberOfPostScripts*] [**-notification** *value*] [**-r** *schedd_name*] [**-debug** *level*] [**-usedagdir**] [**-outfile_dir** *directory*] [**-config** *ConfigFileName*] [**-insert_sub_file** *FileName*] [**-append** *Command*] [**-batch-name** *batch_name*] [**-autorescue** *0|1*] [**-dorescuefrom** *number*] [**-load_save** *filename*] [**-allowversionmismatch**] [**-no_recurse**] [**-do_recurse**] [**-update_submit**] [**-import_env**] [**-include_env** *Variables*] [**-insert_env** *Key=Value*] [**-DumpRescue**] [**-valgrind**] [**-DontAlwaysRunPost**] [**-AlwaysRunPost**] [**-priority** *number*] [**-schedd-daemon-ad-file** *FileName*] [**-schedd-address-file** *FileName*] [**-suppress_notification**] [**-dont_suppress_notification**] [**-DoRecovery**] *DAGInputFile1* [*DAGInputFile2 . . .* *DAGInputFileN*]

15.57.2 Description

condor_submit_dag is the program for submitting a DAG (directed acyclic graph) of jobs for execution under HTCondor. The program enforces the job dependencies defined in one or more *DAGInputFile*s. Each *DAGInputFile* contains commands to direct the submission of jobs implied by the nodes of a DAG to HTCondor. Extensive documentation is in the HTCondor User Manual section on DAGMan.

Some options may be specified on the command line or in the configuration or in a node job's submit description file. Precedence is given to command line options or configuration over settings from a submit description file. An example is e-mail notifications. When configuration variable is its default value of `True`, and a node job's submit description file contains

```
notification = Complete
```

e-mail will not be sent upon completion, as the value of `DAGMAN_SUPPRESS_NOTIFICATION` is enforced.

15.57.3 Options

-help
Display usage information and exit.

-version
Display version information and exit.

-no_submit
Produce the HTCondor submit description file for DAGMan, but do not submit DAGMan as an HTCondor job.

-verbose
Cause *condor_submit_dag* to give verbose error messages.

-force
Require *condor_submit_dag* to overwrite the files that it produces, if the files already exist. Note that `dagman.out` will be appended to, not overwritten. If rescue files exist then DAGMan will run the original DAG and rename the rescue files. Any old-style rescue files will be deleted.

-dagman ***DagmanExecutable***
Allows the specification of an alternate *condor_dagman* executable to be used instead of the one found in the user's path. This must be a fully qualified path.

-maxidle ***NumberOfProcs***
Sets the maximum number of idle procs allowed before *condor_dagman* stops submitting more node jobs. If this option is omitted then the number of idle procs is limited by the configuration variable which defaults to 1000. To disable this limit, set *NumberOfProcs* to 0. The *NumberOfProcs* can be exceeded if a nodes job has a queue command with more than one proc to queue. i.e. `queue 500` will submit all procs even if *NumberOfProcs* is `250`. In this case DAGMan will wait for for the number of idle procs to fall below 250 before submitting more jobs to the **condor_schedd**.

-maxjobs ***NumberOfClusters***
Sets the maximum number of clusters within the DAG that will be submitted to HTCondor at one time. Each cluster is associated with one node job no matter how many individual procs are in the cluster. *NumberOfClusters* is a non-negative integer. If this option is omitted then the number of clusters is limited by the configuration variable which defaults to 0 (unlimited).

-maxpre ***NumberOfPreScripts***
Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPreScripts* is a non-negative integer. If this option is omitted, the number of PRE scripts is limited by the configuration variable which defaults to 20.

-maxpost ***NumberOfPostScripts***
Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPostScripts* is a non-negative integer. If this option is omitted, the number of POST scripts is limited by the configuration variable which defaults to 20.

-notification ***value***
Sets the e-mail notification for DAGMan itself. This information will be used within the HTCondor submit description file for DAGMan. This file is produced by *condor_submit_dag*. See the description of **notification** within *condor_submit* manual page for a specification of *value*.

-r ***schedd_name***
Submit *condor_dagman* to a *condor_schedd* on a remote machine. It is assumed that any necessary files will be present on the remote machine via some method like a shared filesystem between the local and remote machines. The user also requires the correct permissions to submit remotely similarly to *condor_submit*'s **-remote** option. If other options are desired, including transfer of other input files, consider using the **-no_submit** option and modifying the resulting submit file for specific needs before using *condor_submit* on the prouduced DAGMan job submit file.

-debug ***level***
Passes the the *level* of debugging output desired to *condor_dagman*. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. See the *condor_dagman* manual page for detailed descriptions of these values. If not specified, no **-debug** *V*alue is passed to *condor_dagman*.

-usedagdir
This optional argument causes *condor_dagman* to run each specified DAG as if *condor_submit_dag* had been run in the directory containing that DAG file. This option is most useful when running multiple DAGs in a single *condor_dagman*. Note that the **-usedagdir** flag must not be used when running an old-style Rescue DAG.

-outfile_dir ***directory***
Specifies the directory in which the `.dagman.out` file will be written. The *directory* may be specified relative to the current working directory as *condor_submit_dag* is executed, or specified with an absolute path. Without this option, the `.dagman.out` file is placed in the same directory as the first DAG input file listed on the command line.

-config ***ConfigFileName***
Specifies a configuration file to be used for this DAGMan run. This configuration will apply to all DAGs submitted in via DAGMan. Note that only one custom configuration file can be specified for a DAGMan workflow which will cause a failure if used in conjuntion with a DAG using the `CONFIG` command.

-insert_sub_file ***FileName***
Specifies a file to insert into the `.condor.sub` file created by *condor_submit_dag*. The specified file must contain only legal submit file commands. Only one file can be inserted. The specified file will override the file set by the configuration variable . The specified file is inserted into the `.condor.sub` file before the queue command and any commands specified with the **-append** option.

-append ***Command***
Specifies a command to append to the `.condor.sub` file created by *condor_submit_dag*. The specified command is appended to the `.condor.sub` file immediately before the queue command and after any commands added via **-insert_sub_file** or . Multiple commands are specified by using the **-append** option multiple times. Commands with spaces in them must be enclosed in double quotes.

-batch-name ***batch_name***
Set the batch name for this DAG/workflow. The batch name is displayed by *condor_q*. If omitted DAGMan will set the batch name to `DagFile+ClusterId` where *DagFile* is the name of the primary DAG submitted DAGMan and *ClusterId* is the DAGMan proper jobs `ClusterId`. The batch name is set in all jobs submitted by DAGMan and propagated down into sub-DAGs. Note: set the batch name to ' ' (space) to avoid overriding batch names specified in node job submit files.

-autorescue ***0|1***
Whether to automatically run the newest rescue DAG for the given DAG file, if one exists (0 = `false`, 1 = `true`).

-dorescuefrom ***number***
Forces *condor_dagman* to run the specified rescue DAG number for the given DAG. A value of 0 is the same as not specifying this option. Specifying a non-existent rescue DAG is a fatal error.

-load_save *filename*
: Specify a file with saved DAG progress to re-run the DAG from. If given a path DAGMan will attempt to read that file following that path. Otherwise, DAGMan will check for the file in the DAG's `save_files` sub-directory.

-allowversionmismatch
: This optional argument causes *condor_dagman* to allow a version mismatch between *condor_dagman* itself and the `.condor.sub` file produced by *condor_submit_dag* (or, in other words, between *condor_submit_dag* and *condor_dagman*). WARNING! This option should be used only if absolutely necessary. Allowing version mismatches can cause subtle problems when running DAGs.

-no_recurse
: This optional argument causes *condor_submit_dag* to not run itself recursively on nested DAGs (this is now the default; this flag has been kept mainly for backwards compatibility).

-do_recurse
: This optional argument causes *condor_submit_dag* to run itself recursively on nested DAGs to pre-produce their `.condor.sub` files. DAG nodes specified with the **SUBDAG EXTERNAL** keyword or with submit file names ending in `.condor.sub` are considered nested DAGs. This flag is useful when the configuration variable is `False` (Not default).

-update_submit
: This optional argument causes an existing `.condor.sub` file to not be treated as an error; rather, the `.condor.sub` file will be overwritten, but the existing values of **-maxjobs**, **-maxidle**, **-maxpre**, and **-maxpost** will be preserved.

-import_env
: This optional argument causes *condor_submit_dag* to import the current environment into the **environment** command of the `.condor.sub` file it generates.

-include_env ***Variables***
: This optional argument takes a comma separated list of enviroment variables to add to `.condor.sub` `getenv` environment filter which causes found matching environment variables to be added to the DAGMan manager jobs **environment**.

-insert_env ***Key=Value***
: This optional argument takes a delimited string of *Key=Value* pairs to explicitly set into the `.condor.sub` files `environment` macro. The base delimiter is a semicolon that can be overriden by setting the first character in the string to a valid delimiting character. If multiple **-insert_env** flags contain the same *Key* then the last occurances *Value* will be set in the DAGMan jobs **environment**.

-DumpRescue
: This optional argument tells *condor_dagman* to immediately dump a rescue DAG and then exit, as opposed to actually running the DAG. This feature is mainly intended for testing. The Rescue DAG file is produced whether or not there are parse errors reading the original DAG input file. The name of the file differs if there was a parse error.

-valgrind
: This optional argument causes the submit description file generated for the submission of *condor_dagman* to be modified. The executable becomes *valgrind* run on *condor_dagman*, with a specific set of arguments intended for testing *condor_dagman*. Note that this argument is intended for testing purposes only. Using the **-valgrind** option without the necessary *valgrind* software installed will cause the DAG to fail. If the DAG does run, it will run much more slowly than usual.

-DontAlwaysRunPost
: This option causes the submit description file generated for the submission of *condor_dagman* to be modified. It causes *condor_dagman* to not run the POST script of a node if the PRE script fails.

-AlwaysRunPost

This option causes the submit description file generated for the submission of *condor_dagman* to be modified. It causes *condor_dagman* to always run the POST script of a node, even if the PRE script fails.

-priority *number*
Sets the minimum job priority of node jobs submitted and running under the *condor_dagman* job submitted by this *condor_submit_dag* command.

-schedd-daemon-ad-file *FileName*
Specifies a full path to a daemon ad file dropped by a *condor_schedd*. Therefore this allows submission to a specific scheduler if several are available without repeatedly querying the *condor_collector*. The value for this argument defaults to the configuration attribute .

-schedd-address-file *FileName*
Specifies a full path to an address file dropped by a *condor_schedd*. Therefore this allows submission to a specific scheduler if several are available without repeatedly querying the *condor_collector*. The value for this argument defaults to the configuration attribute .

-suppress_notification
Causes jobs submitted by *condor_dagman* to not send email notification for events. The same effect can be achieved by setting configuration variable to `True`. This command line option is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself.

-dont_suppress_notification
Causes jobs submitted by *condor_dagman* to defer to content within the submit description file when deciding to send email notification for events. The same effect can be achieved by setting configuration variable to `False`. This command line flag is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself. If both **-dont_suppress_notification** and **-suppress_notification** are specified with the same command line, the last argument is used.

-DoRecovery
Causes *condor_dagman* to start in recovery mode. This means that DAGMan reads the relevant `.nodes.log` file to restore its previous state of node completions and failures to continue running.

15.57.4 Exit Status

condor_submit_dag will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.57.5 Examples

To run a single DAG:

```
$ condor_submit_dag diamond.dag
```

To run a DAG when it has already been run and the output files exist:

```
$ condor_submit_dag -force diamond.dag
```

To run a DAG, limiting the number of idle node jobs in the DAG to a maximum of five:

```
$ condor_submit_dag -maxidle 5 diamond.dag
```

To run a DAG, limiting the number of concurrent PRE scripts to 10 and the number of concurrent POST scripts to five:

```
$ condor_submit_dag -maxpre 10 -maxpost 5 diamond.dag
```

To run two DAGs, each of which is set up to run in its own directory:

```
$ condor_submit_dag -usedagdir dag1/diamond1.dag dag2/diamond2.dag
```

15.58 *condor_suspend*

suspend jobs from the HTCondor queue

15.58.1 Synopsis

condor_suspend [**-help | -version**]

condor_suspend [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] **

15.58.2 Description

condor_suspend suspends one or more jobs from the HTCondor job queue. When a job is suspended, the match between the *condor_schedd* and machine is not been broken, such that the claim is still valid. But, the job is not making any progress and HTCondor is no longer generating a load on the machine. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The job(s) to be suspended are identified by one of the job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can suspend the job.

15.58.3 Options

-help
: Display usage information

-version
: Display version information

-pool ***centralmanagerhostname[:portnumber]***
: Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname***
: Send the command to a machine identified by *scheddname*

-addr ***"<a.b.c.d:port>"***
: Send the command to a machine located at *"<a.b.c.d:port>"*

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

cluster
: Suspend all jobs in the specified cluster

cluster.process
: Suspend the specific job in the cluster

user
Suspend jobs belonging to specified user

-constraint ***expression***
Suspend all jobs which match the job ClassAd expression constraint

-all
Suspend all the jobs in the queue

15.58.4 Exit Status

condor_suspend will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.58.5 Examples

To suspend all jobs except for a specific user:

```
$ condor_suspend -constraint 'Owner =!= "foo"'
```

Run *condor_continue* to continue execution.

15.59 *condor_tail*

Display the last contents of a running job's standard output or file

15.59.1 Synopsis

condor_tail [**-help**] | [**-version**]

condor_tail [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *name*] [**-debug**] [**-maxbytes** *numbytes*] [**-auto-retry**] [**-follow**] [**-no-stdout**] [**-stderr**] *job-ID* [*filename1*] [*filename2 . . .*]

15.59.2 Description

condor_tail displays the last bytes of a file in the sandbox of a running job identified by the command line argument *job-ID*. `stdout` is tailed by default. The number of bytes displayed is limited to 1024, unless changed by specifying the **-maxbytes** option. This limit is applied for each individual tail of a file; for example, when following a file, the limit is applied each subsequent time output is obtained.

If you specify *filename*, that name must be specifically listed in the job's `transfer_output_files`.

15.59.3 Options

-help
Display usage information and exit.

-version
Display version information and exit.

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number.

-name ***name***
Query the *condor_schedd* daemon identified with *name*.

-debug
Display extra debugging information.

-maxbytes ***numbytes***
Limits the maximum number of bytes transferred per tail access. If not specified, the maximum number of bytes is 1024.

-auto-retry
Retry the tail of the file(s) every 2 seconds, if the job is not yet running.

-follow
Repetitively tail the file(s), until interrupted.

-no-stdout
Do not tail `stdout`.

-stderr
Tail `stderr` instead of `stdout`.

15.59.4 Exit Status

The exit status of *condor_tail* is zero on success.

15.60 *condor_test_token*

Create a short-lived SciToken to authenticate with local HTCondor daemons

15.60.1 Synopsis

condor_test_token [**–help**]

condor_test_token **–issuer** *issuer-url* **–scope** *scopes* [**–subject** *subject*] [**–lifetime** *lifetime*] [**–audience** *audience*] [**–cache** *cache-location*]

15.60.2 Description

condor_test_token generates a temporary signing key, adds it to the local SciTokens cache for the given issuer, creates a short-lived token signed by the key, and prints the token to stdout. Local HTCondor daemons will treat this token like any reguluar token generated by the given issuer for a short period of time (one hour).

If the HTCondor daemons were started as root, then the tool must be run as the condor user.

15.60.3 Options

–help
Display usage information

–issuer ***issuer-url***
Specify the issuer to impersonate

–scope ***scopes***
Specify the scope claim for the token

–subject ***subject***
specify the sub claim for the token (default is no sub claim)

–lifetime ***lifetime***
Specify the lifetime of the token in seconds (default 1 hour)

–audience ***audience***
Specify the aud claim for the token (default is no aud claim)

–cache ***cache-location***
Specify the SciTokens cache location (default is to find cache via HTCondor configuration files)

15.60.4 Examples

To create a SciToken with WRITE-level access for user Alice that appears to be issued by the SciTokens demo issuer:

```
$ condor_test_token --issuer https://demo.scitokens.org \
    --scope condor:/WRITE --sub alice@foo.org --aud ANY
```

15.60.5 Exit Status

condor_test_token will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.61 *condor_token_create*

given a password file, create an authentication token for the IDTOKENS authentication method

15.61.1 Synopsis

condor_token_create -identity *user@domain* [**-key** *keyid*] [**-authz** *authz* ...] [**-lifetime** *value*] [**-token** *filename*] [**-debug**]

condor_token_create [**-help**]

15.61.2 Description

condor_token_create will read an HTCondor password file inside the *SEC_PASSWORD_DIRECTORY* (by default, this is the pool password) and use it to create an authentication token. The authentication token may be subsequently used by clients to authenticate against a remote HTCondor server. Tokens allow fine-grained authentication as individual HTCondor users as opposed to pool password, where anything in possession of the pool password will authenticate as the same user.

An identity must be specified for the token; this will be the client's resulting identity at the remote HTCondor server. If the **-lifetime** or (one or more) **-authz** options are specified, the token will contain additional restrictions that limit what the client will be authorized to do. If an attacker is able to access the token, they will be able to authenticate with the identity listed in the token (subject to the restrictions above).

If successful, the resulting token will be sent to `stdout`; by specifying the **-token** option, it will instead be written to the user's token directory. If written to *SEC_TOKEN_SYSTEM_DIRECTORY* (default `/etc/condor/tokens.d`), then the token can be used for daemon-to-daemon authentication.

condor_token_create is only currently supported on Unix platforms.

15.61.3 Options

-authz ***authz***
: Adds a restriction to the token so it is only valid to be used for a given authorization level (such as `READ`, `WRITE`, `DAEMON`, `ADVERTISE_STARTD`). If multiple authorizations are needed, then **-authz** must be specified multiple times. If **-authz** is not specified, no authorization restrictions are added and authorization will be solely based on the token's identity. **NOTE** that **-authz** cannot be used to give an identity additional permissions at the remote host. If the server's admin only permits the user `READ` authorization, then specifying `-authz WRITE` in a token will not allow the user to perform writes.

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-help
: Display brief usage information and exit.

-identity ***user@domain***
: Set a specific client identity to be written into the token; a client will authenticate as this identity with a remote server.

-key ***keyid***
: Specify a key file to use under the directory specified by the *SEC_PASSWORD_DIRECTORY* configuration variable. The key name must match a file in the password directory; the file's contents must be created with *condor_store_cred* and will be used to sign the resulting token. If **-key** is not set, then the default pool password will be used.

-lifetime ***value***
: Specify the lifetime, in seconds, for the token to be valid (the token validity will start when the token

is signed). After the lifetime expires, the token cannot be used for authentication. If not specified, the token will contain no lifetime restrictions.

-token *filename*
Specifies a filename, relative to the directory in the *SEC_TOKEN_DIRECTORY* configuration variable (for example, on Linux this defaults to `~/.condor/tokens.d`), where the resulting token is stored. If not specified, the token will be sent to `stdout`.

15.61.4 Examples

To create a token for `jane@cs.wisc.edu` with no additional restrictions:

```
$ condor_token_create -identity jane@cs.wisc.edu
eyJhbGciOiJIUzI1NiIsImtpZCI6Il....bnu3NoO9BGM
```

To create a token for `worker-node@cs.wisc.edu` that may advertise either a *condor_startd* or a *condor_master*:

```
$ condor_token_create -identity worker-node@cs.wisc.edu \
                      -authz ADVERTISE_STARTD \
                      -authz ADVERTISE_MASTER
eyJhbGciOiJIUzI1NiIsImtpZC.....8wkstyj_OnM0SHsOdw
```

To create a token for `friend@cs.wisc.edu` that is only valid for 10 minutes, and then to save it to `~/.condor/tokens.d/friend`:

```
$ condor_token_create -identity friend@cs.wisc.edu -lifetime 600 -token friend
```

If the administrator would like to create a specific key for signing tokens, `token_key`, distinct from the default pool password, they would first use *condor_store_cred* to create the key:

```
$ openssl rand -base64 32 | condor_store_cred -f /etc/condor/passwords.d/token_key
```

Note, in this case, we created a random 32 character key using SSL instead of providing a human-friendly password.

Next, the administrator would run run *condor_token_create*:

```
$ condor_token_create -identity frida@cs.wisc.edu -key token_key
eyJhbGciOiJIUzI1NiIsImtpZCI6I.....eyJpYXQiOUzlN6QA
```

If the `token_key` file is deleted from the *SEC_PASSWORD_DIRECTORY*, then all of the tokens issued with that key will be invalidated.

15.61.5 Exit Status

condor_token_create will exit with a non-zero status value if it fails to read the password file, sign the token, write the output, or experiences some other error. Otherwise, it will exit 0.

15.61.6 See also

condor_store_cred(1), *condor_token_fetch(1)*, *condor_token_request(1)*, *condor_token_list(1)*

15.61.7 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.62 *condor_token_fetch*

obtain a token from a remote daemon for the IDTOKENS authentication method

15.62.1 Synopsis

condor_token_fetch [**-authz** *authz* ...] [**-lifetime** *value*] [**-pool** *pool_name*] [**-name** hostname] [**-type** *type*] [**-token** *filename*] [**-key** *signing_key*]

condor_token_fetch [**-help**]

15.62.2 Description

condor_token_fetch will attempt to fetch an authentication token from a remote daemon. If successful, the identity embedded in the token will be the same as client's identity at the remote daemon.

Authentication tokens are a useful mechanism to limit an identity's authorization or to establish an alternate authentication method. For example, an administrator may utilize *condor_token_fetch* to create a token for a monitoring host that is limited to only the `READ` authorization. A user may use *condor_token_fetch* while they are logged in to a submit host then use the resulting token to submit remotely from their personal laptop.

If the **-lifetime** or (one or more) **-authz** options are specified, the token will contain additional restrictions that limit what the client will be authorized to do.

By default, *condor_token_fetch* will query the local *condor_schedd*; by specifying a combination of **-pool**, **-name**, or **-type**, the tool can request tokens in other pools, on other hosts, or different daemon types.

If successful, the resulting token will be sent to `stdout`; by specifying the **-token** option, it will instead be written to the user's token directory.

15.62.3 Options

-authz *authz*
: Adds a restriction to the token so it is only valid to be used for a given authorization level (such as `READ`, `WRITE`, `DAEMON`, `ADVERTISE_STARTD`). If multiple authorizations are needed, then **-authz** must be specified multiple times. If **-authz** is not specified, no authorization restrictions are added and authorization will be solely based on the token's identity. **NOTE** that **-authz** cannot be used to give an identity additional permissions at the remote host. If the server's admin only permits the user `READ` authorization, then specifying `-authz WRITE` in a token will not allow the user to perform writes.

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-help
Display brief usage information and exit.

-lifetime ***value***
Specify the lifetime, in seconds, for the token to be valid (the token validity will start when the token is signed). After the lifetime expires, the token cannot be used for authentication. If not specified, the token will contain no lifetime restrictions.

-name ***hostname***
Request a token from the daemon named *hostname* in the pool. If not specified, the locally-running daemons will be used.

-pool ***pool_name***
Request a token from a daemon in a non-default pool *pool_name*.

-token ***filename***
Specifies a filename, relative to the directory in the *SEC_TOKEN_DIRECTORY* configuration variable (defaulting to `~/.condor/tokens.d`), where the resulting token is stored. If not specified, the token will be sent to `stdout`.

-type ***type***
Request a token from a specific daemon type *type*. If not given, a *condor_schedd* is used.

-key ***signing_key***
Request a token signed by the signing key named *signing_key*. If not given, the daemon's default key will be used.

15.62.4 Examples

To obtain a token with a lifetime of 10 minutes from the default *condor_schedd*:

```
$ condor_token_fetch -lifetime 600
eyJhbGciOiJIUzI1NiIsImtpZCI6IlBPT0wifQ.eyJpYX...ii7lAfCA
```

To request a token from `bird.cs.wisc.edu` which is limited to `READ` and `WRITE`:

```
$ condor_token_fetch -name bird.cs.wisc.edu \
                     -authz READ -authz WRITE
eyJhbGciOiJIUzI1NiIsImtpZCI6IlBPT0wifQ.eyJpYX...lJTj54
```

To create a token from the collector in the `htcondor.cs.wisc.edu` pool and then to save it to `~/.condor/tokens.d/friend`:

```
$ condor_token_fetch -identity friend@cs.wisc.edu -lifetime 600 -token friend
```

15.62.5 Exit Status

condor_token_fetch will exit with a non-zero status value if it fails to request or read the token. Otherwise, it will exit 0.

15.62.6 See also

condor_token_create(1), *condor_token_request(1)*, *condor_token_list(1)*

15.62.7 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.63 *condor_token_list*

list all available tokens for IDTOKENS auth

15.63.1 Synopsis

condor_token_list [**-dir** *directory*]

condor_token_list -help

15.63.2 Description

condor_token_list parses the tokens available to the current user and prints them to `stdout`.

The tokens are stored in files in the directory referenced by *SEC_TOKEN_DIRECTORY*; multiple tokens may be saved in each file (one per line).

The output format is a list of the deserialized contents of each token, along with the file name containing the token, one per line. It should not be considered machine readable and will be subject to change in future release of HTCondor.

15.63.3 Options

-help
: Display brief usage information and exit.

-dir
: Read tokens from an alternate directory.

15.63.4 Examples

To list all tokens as the current user:

```
$ condor_token_list
Header: {"alg":"HS256","kid":"POOL"} Payload: {"exp":1565576872,"iat":1565543872,"iss":
↪"htcondor.cs.wisc.edu","scope":"condor:\/DAEMON","sub":"k8sworker@wisc.edu"} File: /
↪home/bucky/.condor/tokens.d/token1
Header: {"alg":"HS256","kid":"POOL"} Payload: {"iat":1572414350,"iss":"htcondor.cs.wisc.
↪edu","scope":"condor:\/WRITE","sub":"bucky@wisc.edu"} File: /home/bucky/.condor/tokens.
↪d/token2
```

15.63.5 Exit Status

condor_token_list will exit with a non-zero status value if it fails to read the token directory, tokens are improperly formatted, or if it experiences some other error. Otherwise, it will exit 0.

15.63.6 See also

condor_token_create(1), *condor_token_fetch(1)*, *condor_token_request(1)*

15.63.7 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.64 *condor_token_request*

interactively request a token from a remote daemon for the IDTOKENS authentication method

15.64.1 Synopsis

condor_token_request [**-identity** *user@domain*] [**-authz** *authz . . .*] [**-lifetime** *value*] [**-pool** *pool_name*] [**-name** hostname] [**-type** *type*] [**-token** *filename*]

condor_token_request [**-help**]

15.64.2 Description

condor_token_request will request an authentication token from a remote daemon. Token requests must be approved by the daemon's administrator using *condor_token_request_approve*. Unlike *condor_token_fetch*, the user doesn't need an existing identity with the remote daemon when using *condor_token_request* (an anonymous method, such as SSL without a client certificate will suffice).

If the request is successfully enqueued, the request ID will be printed to `stderr`; the administrator will need to know the ID to approve the request. *condor_token_request* will wait until the request is approved, timing out after an hour.

The token request mechanism provides a powerful way to bootstrap authentication in a HTCondor pool - a remote user can request an identity, verify the authenticity of the request out-of-band with the remote daemon's administrator, and then securely recieve their authentication token.

By default, *condor_token_request* will query the local *condor_collector*; by specifying a combination of **-pool**, **-name**, or **-type**, the tool can request tokens in other pools, on other hosts, or different daemon types.

If successful, the resulting token will be sent to `stdout`; by specifying the **-token** option, it will instead be written to the user's token directory.

15.64.3 Options

-authz *authz*
: Adds a restriction to the token so it is only valid to be used for a given authorization level (such as `READ`, `WRITE`, `DAEMON`, `ADVERTISE_STARTD`). If multiple authorizations are needed, then **-authz** must be specified multiple times. If **-authz** is not specified, no authorization restrictions are added and authorization will be solely based on the token's identity. **NOTE** that **-authz** cannot be used to give an identity additional permissions at the remote host. If the server's admin only permits the user `READ` authorization, then specifying `-authz WRITE` in a token will not allow the user to perform writes.

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-help
: Display brief usage information and exit.

-identity *user@domain*
: Request a specific identity from the daemon; a client using the resulting token will authenticate as this identity with a remote server. If not specified, the token will be issued for the `condor` identity.

-lifetime *value*
: Specify the lifetime, in seconds, for the token to be valid (the token validity will start when the token is signed). After the lifetime expires, the token cannot be used for authentication. If not specified, the token will contain no lifetime restrictions.

-name *hostname*
: Request a token from the daemon named *hostname* in the pool. If not specified, the locally-running daemons will be used.

-pool *pool_name*
: Request a token from a daemon in a non-default pool *pool_name*.

-token *filename*
: Specifies a filename, relative to the directory in the *SEC_TOKEN_DIRECTORY* configuration variable (defaulting to `~/.condor/tokens.d`), where the resulting token is stored. If not specified, the token will be sent to `stdout`.

-type *type*
: Request a token from a specific daemon type *type*. If not given, a *condor_collector* is used.

15.64.4 Examples

To obtain a token with a lifetime of 10 minutes from the default *condor_collector* (the token is not returned until the daemon's administrator takes action):

```
$ condor_token_request -lifetime 600
Token request enqueued.  Ask an administrator to please approve request 6108900.
eyJhbGciOiJIUzI1NiIsImtpZCI6IlBPT0wifQ.eyJpYX...ii7lAfCA
```

To request a token from `bird.cs.wisc.edu` which is limited to `READ` and `WRITE`:

```
$ condor_token_request -name bird.cs.wisc.edu \
                       -identity bucky@cs.wisc.edu
                       -authz READ -authz WRITE
```

(continues on next page)

(continued from previous page)

```
Token request enqueued.  Ask an administrator to please approve request 2578154
eyJhbGciOiJIUzI1NiIsImtpZCI6IlBPT0wifQ.eyJpYX...lJTj54
```

To create a token from the collector in the `htcondor.cs.wisc.edu` pool and then to save it to `~/.condor/tokens.d/friend`:

```
$ condor_token_request -pool htcondor.cs.wisc.edu \
                       -identity friend@cs.wisc.edu \
                       -lifetime 600 -token friend
Token request enqueued.  Ask an administrator to please approve request 2720841.
```

15.64.5 Exit Status

condor_token_request will exit with a non-zero status value if it fails to request or recieve the token. Otherwise, it will exit 0.

15.64.6 See also

condor_token_create(1), *condor_token_fetch(1)*, *condor_token_request_approve(1)*, *condor_token_request_auto_approve(1)*, *condor_token_list(1)*

15.64.7 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.65 *condor_token_request_approve*

approve a token request at a remote daemon

15.65.1 Synopsis

condor_token_request_approve [**-reqid** *val*] [**-pool** *pool_name*] [**-name** hostname] [**-type** *type*] [**-debug**]

condor_token_request_approve [**-help**]

15.65.2 Description

condor_token_request_approve will approve an request for an authentication token queued at a remote daemon. Once approved, the requester will be able to fetch a fully signed token from the daemon and use it to authenticate with the IDTOKENS method.

NOTE that any user can request a very powerful token, even allowing them to be the HTCondor administrator; such requests can only be approved by an administrator. Review token requests carefully to ensure you understand what identity you are approving. The only safe way to approve a request is to have the request ID communicated out-of-band and verify it matches the expected, request contents, ensuring the request's authenticity.

By default, users can only approve requests for their own identity (that is, a user authenticating as `bucky@cs.wisc.edu` can only approve token requests for the identity `bucky@cs.wisc.edu`). Users with `ADMINISTRATOR` authorization can approve any request.

If you want to approve multiple requests at once, do not provide the **-reqid** flag; in that case, the utility will iterate through all known requests.

By default, *condor_token_request_approve* will query the local *condor_collector*; by specifying a combination of **-pool**, **-name**, or **-type**, the tool can request tokens in other pools, on other hosts, or different daemon types.

15.65.3 Options

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-help
Display brief usage information and exit.

-name ***hostname***
Request a token from the daemon named *hostname* in the pool. If not specified, the locally-running daemons will be used.

-pool ***pool_name***
Request a token from a daemon in a non-default pool *pool_name*.

-reqid ***val***
Provides the specific request ID to approve. Request IDs should be communicated out of band to the administrator through a trusted channel.

-type ***type***
Request a token from a specific daemon type *type*. If not given, a *condor_collector* is used.

15.65.4 Examples

To approve the tokens at the default *condor_collector*, one-by-one:

```
$ condor_token_request_approve
RequestedIdentity = "bucky@cs.wisc.edu"
AuthenticatedIdentity = "anonymous@ssl"
PeerLocation = "10.0.0.42"
ClientId = "bird.cs.wisc.edu-516"
RequestId = "8414912"

To approve, please type 'yes'
yes
Request 8414912 approved successfully.
```

When a token is approved, the corresponding *condor_token_request* process will complete. Note the printed request includes both the requested identity (which will be written into the issued token) and the authenticated identity of the token requester. In this case, `anonymous@ssl` indicates the connection was established successfully over SSL but the remote side is anonymous (did not contain a client SSL certificate).

15.65.5 Exit Status

condor_token_request_approve will exit with a non-zero status value if it fails to communicate with the remote daemon. Otherwise, it will exit 0.

15.65.6 See also

condor_token_request(1), *condor_token_fetch(1)*, *condor_token_request_auto_approve(1)*

15.65.7 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.66 *condor_token_request_auto_approve*

generate a new rule to automatically approve token requests

15.66.1 Synopsis

condor_token_request_auto_approve -netblock *network* **-lifetime** *val* [**-pool** *pool_name*] [**-name** hostname] [**-type** *type*] [**-debug**]

condor_token_request_auto_approve [**-help**]

15.66.2 Description

condor_token_request_auto_approve will install a temporary auto-approval rule for token requests. Any token request matching the auto-approval rule will be immediately approved instead of requiring administrator approval

Automatic request approval is intended to help administrators initially setup their cluster. To install a new rule, you must specify both a network and a lifetime; requests are only approved if they come from that given source network, are within the rule lifetime, are limited to `ADVERTISE_SCHEDD` or `ADVERTISE_STARTD` permissions, and are for the `condor` identity. When a *condor_startd* or *condor_schedd* is started and cannot communicate with the collector, they will automatically generate token requests that meet the last two conditions.

It is not safe to enable auto-approval when users have access to any of the involved hosts or networks.

To remove auto-approval rules, run `condor_reconfig` against the remote daemon.:

By default, *condor_token_request_auto_approve* will install rules at the local *condor_collector*; by specifying a combination of **-pool**, **-name**, or **-type**, the tool can request tokens in other pools, on other hosts, or different daemon types.

15.66.3 Options

-debug
Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-help
Display brief usage information and exit.

-lifetime ***value***
Specify the lifetime, in seconds, for the auto-request rule to be valid.

-name ***hostname***
Request a token from the daemon named *hostname* in the pool. If not specified, the locally-running daemons will be used.

-netblock ***network***
A netblock of the form *IP_ADDRESS/SUBNET_MASK* specifying the source of authorized requests. Examples may include `129.93.12.0/24` or `10.0.0.0/26`.

-pool ***pool_name***
Request a token from a daemon in a non-default pool *pool_name*.

-type ***type***
Request a token from a specific daemon type *type*. If not given, a *condor_collector* is used.

15.66.4 Examples

To automatically approve token requests to the default *condor_collector* coming from the `10.0.0.0/26` subnet for the next 10 minutes:

```
$ condor_token_request_auto_approve -lifetime 600 -netblock 10.0.0.0/26
Successfully installed auto-approval rule for netblock 10.0.0.0/26 with lifetime of 0.17␣
↪hours
Remote daemon reports no un-approved requests pending.
```

15.66.5 Exit Status

condor_token_request_auto_approve will exit with a non-zero status value if it fails to communicate with the remote daemon or has insufficient authorization. Otherwise, it will exit 0.

15.66.6 See also

condor_token_request(1), *condor_token_request_approve(1)*

15.66.7 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.67 *condor_token_request_list*

list all token requests at a remote daemon

15.67.1 Synopsis

condor_token_request_list [**-pool** *pool_name*] [**-name** hostname] [**-type** *type*] [**-json**] [**-debug**]

condor_token_request_list [**-help**]

15.67.2 Description

condor_token_request_list will list all requests for tokens currently queued at a remote daemon. This allows the administrator to review token requests; these requests may be subsequently approved with an invocation of *condor_token_request_approve*.

An individual with `ADMINISTRATOR` authorization may see all queued token requests; otherwise, users can only see token requests for their own identity.

By default, *condor_token_request_list* will query the local *condor_collector*; by specifying a combination of **-pool**, **-name**, or **-type**, the tool can request tokens in other pools, on other hosts, or different daemon types.

15.67.3 Options

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-help
: Display brief usage information and exit.

-name ***hostname***
: Request a token from the daemon named *hostname* in the pool. If not specified, the locally-running daemons will be used.

-pool ***pool_name***
: Request a token from a daemon in a non-default pool *pool_name*.

-json
: Causes all pending requests to be printed as JSON objects.

-type ***type***
: Request a token from a specific daemon type *type*. If not given, a *condor_collector* is used.

15.67.4 Examples

To list the tokens at the default *condor_collector*:

```
$ condor_token_request_list
RequestId = "4303687"
ClientId = "worker0000.wisc.edu-960"
PeerLocation = "10.0.4.13"
AuthenticatedIdentity = "anonymous@ssl"
RequestedIdentity = "condor@cs.wisc.edu"
LimitAuthorization = "ADVERTISE_STARTD"

RequestedIdentity = "bucky@cs.wisc.edu"
AuthenticatedIdentity = "bucky@cs.wisc.edu"
PeerLocation = "129.93.244.211"
ClientId = "desktop0001.wisc.edu-712"
RequestId = "4413973"
```

15.67.5 Exit Status

condor_token_request_list will exit with a non-zero status value if it fails to communicate with the remote daemon or fails to authenticate. Otherwise, it will exit 0.

15.67.6 See also

condor_token_request(1), *condor_token_request_approve(1)*, *condor_token_list(1)*

15.67.7 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.68 *condor_top*

Display status and runtime statistics of a HTCondor daemon

15.68.1 Synopsis

condor_top [**-h**]

condor_top [**-l**] [**-p** *centralmanagerhostname[:portname]*] [**-n** *name*] [**-d** *delay*] [**-c** *columnset*] [**-s** *sortcolumn*] [–**attrs=<attr1,attr2,...>**] [*daemon options*]

condor_top [**-c** *columnset*] [**-s** *sortcolumn*] [–**attrs=<attr1,attr2,...>**] [*classad-filename classad-filename*]

15.68.2 Description

condor_top displays the status (e.g. memory usage and duty cycle) of a HTCondor daemon and calculates and displays runtime statistics for the daemon's subprocesses.

When no arguments are specified, *condor_top* displays the status for the primary daemon based on the role of the current machine by scanning the `DAEMON_LIST` configuration setting. If multiple daemons are listed, *condor_top* will monitor one of (in decreasing priority): *condor_schedd*, *condor_startd*, *condor_collector*, *condor_negotiator*, *condor_master*.

If the *condor_collector* returns multiple ClassAds for the chosen daemon type, *condor_top* will display stats from the first ClassAd returned. Results can be constrained by passing the `NAME` of a specific daemon with **-n**.

The default *delay* is `STATISTICS_WINDOW_QUANTUM`, which is 4 minutes (240 seconds) in a default HTCondor configuration. Setting the delay smaller can be helpful for finding spikes of activity, but setting the delay too small will lead to poor measurements of the duty cycle and of the runtime statistics.

condor_top can run in a top-like "live" mode by passing **-l**. The live mode is similar to the *nix top command, with stats updating every *delay* seconds. Redirecting stdout will disable live mode even if **-l** is set. To exit *condor_top* while in live mode, issue Ctrl-C.

condor_top can be passed two files containing ClassAds from the same HTCondor daemon, in which case the *condor_collector* will not be queried but rather the statistics will be computed and displayed immediately from the two ClassAds. Only -c, -s, and -attrs options are considered when passing ClassAds via files.

The following subprocess stat columns may be displayed (*default):

Item
: *Name of the subprocess

InstRt
: *Total runtime between the two ClassAds

InstAvg
: *Mean runtime per execution between the two ClassAds

TotalRt
: Total runtime since daemon start

TotAvg
: *Mean runtime per execution since daemon start

TotMax
: *Max runtime per execution since daemon start

TotMin
: Min runtime per execution since daemon start

RtPctAvg
: *Percent of mean runtime per execution. The ratio of InstAvg to TotAvg, expressed as a percentage

RtPctMax
: Percent of max runtime per execution. The ratio of (InstAvg - TotMin) to (TotMax - TotMin), expressed as a percentage

RtSigmas
: Standard deviations from mean runtime. The ratio of (InstAvg - TotAvg) to the standard deviation in runtime per execution since daemon start

InstCt
: Executions between the two ClassAds

InstRate
: *Executions per second between the two ClassAds

TotalCt
Total executions (counts) since daemon start

AvgRate
*Mean count rate. Executions per second since daemon start

CtPctAvg
Percent of mean count rate. The ratio of InstRate to AvgRate, expressed as a percentage.

15.68.3 Options

-h
Displays the list of options.

-l
Puts *condor_top* in to a live, continually updating mode.

-p *centralmanagerhostname[:portname]*
Query the daemon via the specified central manager. If omitted, the value of the configuration variable `COLLECTOR_HOST` is used.

-n *name*
Query the daemon named *name*. If omitted, the value used will depend on the type of daemon queried (see Daemon Options).

-d *delay*
Specifies the *delay* between ClassAd updates, in integer seconds. If omitted, the value of the configuration variable `STATISTICS_WINDOW_QUANTUM` is used.

-c *columnset*
Display *columnset* set of columns. Valid *columnset* s are: default, runtime, count, all.

-s *sortcolumn*
Sort table by *sortcolumn*. Defaults to InstRt.

-attrs=<attr1,attr2,...>

Comma-delimited list of additional ClassAd attributes to monitor.

Daemon Options

-collector
Monitor *condor_collector* ClassAds. If -n is not set, the constraint "Machine == `COLLECTOR_HOST`" will be used.

-negotiator
Monitor *condor_negotiator* ClassAds. If -n is not set, the constraint "Machine == `COLLECTOR_HOST`" will be used.

-master
Monitor *condor_master* ClassAds. If -n is not set, the constraint "Machine == `COLLECTOR_HOST`" will be used.

-schedd
Monitor *condor_schedd* ClassAds. If -n is not set, the constraint "Machine == `FULL_HOSTNAME`" will be tried, otherwise the first *condor_schedd* ClassAd returned from the *condor_collector* will be used.

-startd
Monitor *condor_startd* ClassAds. If -n is not set, the constraint "Machine == `FULL_HOSTNAME`" will be tried, otherwise the first *condor_startd* ClassAd returned from the *condor_collector* will be used.

15.69 *condor_transfer_data*

transfer spooled data

15.69.1 Synopsis

condor_transfer_data [**-help | -version**]

condor_transfer_data [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] *cluster...* | *cluster.process...* | *user...* | **-constraint** *expression ...*

condor_transfer_data [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] **-all**

15.69.2 Description

condor_transfer_data causes HTCondor to transfer spooled data. It is meant to be used in conjunction with the **-spool** option of *condor_submit*, as in

```
$ condor_submit -spool mysubmitfile
```

Submission of a job with the **-spool** option causes HTCondor to spool all input files, the job event log, and any proxy across a connection to the machine where the *condor_schedd* daemon is running. After spooling these files, the machine from which the job is submitted may disconnect from the network or modify its local copies of the spooled files.

When the job finishes, the job has `JobStatus` = 4, meaning that the job has completed. The output of the job is spooled, and *condor_transfer_data* retrieves the output of the completed job.

15.69.3 Options

-help
Display usage information

-version
Display version information

-pool *centralmanagerhostname[:portnumber]*
Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname*
Send the command to a machine identified by *scheddname*

-addr *"<a.b.c.d:port>"*
Send the command to a machine located at *"<a.b.c.d:port>"*

cluster
Transfer spooled data belonging to the specified cluster

cluster.process
Transfer spooled data belonging to a specific job in the cluster

user
Transfer spooled data belonging to the specified user

-constraint ***expression***
Transfer spooled data for jobs which match the job ClassAd expression constraint

-all
Transfer all spooled data

15.69.4 Exit Status

condor_transfer_data will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.70 *condor_transform_ads*

Transform ClassAds according to specified rules, and output the transformed ClassAds.

15.70.1 Synopsis

condor_transform_ads [**-help [rules]**]

condor_transform_ads [**-rules** *rules-file*] [**-jobtransforms** *name-list*] [**-jobroute** *route-name*] [**-in[:<form>]** ** ***in-file**] [-out[:<form>[, nosort]] ** *outfile*] [<*key*>=<*value*>] [**-long**] [**-json**] [**-xml**] [**-verbose**] [**-terse**] [**-debug**] [**-unit-test**] [**-testing**] [**-convertoldroutes**] [*infile1* ... *infileN*]

Note that one or more transforms must be specified in the form of a rules file or a `JOB_TRANSFORM_` or `JOB_ROUTER_ROUTE_` name and at least one input file must be specified. Transforms will be applied in the order they are given on the command line. If a rules file has a TRANSFORM statement with arguments it must be the last rules file. If no output file is specified, output will be written to `stdout`.

15.70.2 Description

condor_transform_ads reads ClassAds from a set of input files, transforms them according to rules defined in a rules files or read from configuration, and outputs the resulting transformed ClassAds.

See the *ClassAd Transforms* section for a description of the transform language.

15.70.3 Options

-help [rules]
Display usage information and exit. **-help rules** displays information about the available transformation rules.

-rules ***rules-file***
Specifies the file containing definitions of the transformation rules, or configuration that declares a `JOB_TRANSFORM_<name>` or `JOB_ROUTER_ROUTE_<name>` variable for use in a subsequent `-jobtransforms <name>` or `-jobroute <name>` argument.

-jobtransforms ***name-list***
A comma-separated list of more transform names. The transform rules will be read from a previous rules file or the configured `JOB_TRANSFORM_<name>` values

-jobroute *name*
A job route. The transform rules will be read from a previous rules file or the configured JOB_ROUTER_ROUTE_<name> values

-in[:<form>] *infile*
Specifies an input file containing ClassAd(s) to be transformed. **<form>**, if specified, is one of:

- **long**: traditional long form (default)
- **xml**: XML form
- **json**: JSON ClassAd form
- **new**: "new" ClassAd form without newlines
- **auto**: guess format by reading the input

If - is specified for *infile*, input is read from stdin.

-out[:<form>[, nosort] *outfile*
Specifies an output file to receive the transformed ClassAd(s). **<form>**, if specified, is one of:

- **long**: traditional long form (default)
- **xml**: XML form
- **json**: JSON ClassAd form
- **new**: "new" ClassAd form without newlines
- **auto**: use the same format as the first input

ClassAds are storted by attribute unless **nosort** is specified.

[<*key*>=<*value*>]
Assign key/value pairs before rules file is parsed; can be used to pass arguments to rules. (More detail needed here.)

-long
Use long form for both input and output ClassAd(s). (This is the default.)

-json
Use JSON form for both input and output ClassAd(s).

-xml
Use XML form for both input and output ClassAd(s).

-verbose
Verbose mode, echo to stderr the transform names as they are applied and individual transform rules as they are executed.

-terse
Disable the **-verbose** option.

-debug
Causes debugging information to be sent to stderr, based on the value of the configuration variable TOOL_DEBUG.

15.70.4 Exit Status

condor_transform_ads will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.70.5 Examples

Here's a simple example that transforms the given input ClassAds according to the given rules:

```
# File: my_input
ResidentSetSize = 500
DiskUsage = 2500000
NumCkpts = 0
TransferrErr = false
Err = "/dev/null"

# File: my_rules
EVALSET MemoryUsage ( ResidentSetSize / 100 )
EVALMACRO WantDisk = ( DiskUsage * 2 )
SET RequestDisk ( $(WantDisk) / 1024 )
RENAME NumCkpts NumCheckPoints
DELETE /(.+)Err/

# Command:
condor_transform_ads -rules my_rules -in my_input

# Output:
DiskUsage = 2500000
Err = "/dev/null"
MemoryUsage = 5
NumCheckPoints = 0
RequestDisk = ( 5000000 / 1024 )
ResidentSetSize = 500
```

15.71 *condor_update_machine_ad*

update a machine ClassAd

15.71.1 Synopsis

condor_update_machine_ad [**-help | -version**]

condor_update_machine_ad [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *startdname*] *path/to/update-ad*

15.71.2 Description

condor_update_machine_ad modifies the specified *condor_startd* daemon's machine ClassAd. The ClassAd in the file given by `path/to/update-ad` represents the changed attributes. The changes persists until the *condor_startd* restarts. If no file is specified on the command line, *condor_update_machine_ad* reads the update ClassAd from `stdin`.

Contents of the file or `stdin` must contain a complete ClassAd. Each line must be terminated by a newline character, including the last line of the file. Lines are of the form

```
<attribute> = <value>
```

Changes to certain ClassAd attributes will cause the *condor_startd* to regenerate values for other ClassAd attributes. An example of this is setting `HasVM`. This will cause `OfflineUniverses`, `VMOfflineTime`, and `VMOfflineReason` to change.

15.71.3 Options

-help
Display usage information and exit

-version
Display the HTCondor version and exit

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number

-name ***startdname***
Send the command to a machine identified by *startdname*

15.71.4 General Remarks

This tool is intended for the use of system administrators when dealing with offline universes.

15.71.5 Examples

To re-enable matching with the VM universe jobs, place on `stdin` a complete ClassAd (including the ending newline character) to change the value of ClassAd attribute `HasVM`:

```
$ echo "HasVM = True
" | condor_update_machine_ad
```

To prevent vm universe jobs from matching with the machine:

```
$ echo "HasVM = False
" | condor_update_machine_ad
```

To prevent vm universe jobs from matching with the machine and specify a reason:

```
$ echo "HasVM = False
VMOfflineReason = \"Cosmic rays.\"
" | condor_update_machine_ad
```

Note that the quotes around the reason are required by ClassAds, and they must be escaped because of the shell. Using a file instead of `stdin` may be preferable in these situations, because neither quoting nor escape characters are needed.

15.71.6 Exit Status

condor_update_machine_ad will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.72 *condor_updates_stats*

Display output from *condor_status*

15.72.1 Synopsis

condor_updates_stats [**–help** | **-h**] | [**–version**]

condor_updates_stats [**–long** | **-l**] [**–history=<min>-<max>**] [**–interval=<seconds>**] [**–notime**] [**–time**] [**–summary** | **-s**]

15.72.2 Description

condor_updates_stats parses the output from *condor_status*, and it displays the information relating to update statistics in a useful format. The statistics are displayed with the most recent update first; the most recent update is numbered with the smallest value.

The number of historic points that represent updates is configurable on a per-source basis by configuration variable `COLLECTOR_DAEMON_HISTORY_SIZE` .

15.72.3 Options

-help
: Display usage information and exit.

-h
: Same as **-help**.

-version
: Display HTCondor version information and exit.

-long
: All update statistics are displayed. Without this option, the statistics are condensed.

-l
: Same as **-long**.

-history=<min>-<max>
: Sets the range of update numbers that are printed. By default, the entire history is displayed. To limit the range, the minimum and/or maximum number may be specified. If a minimum is not specified, values from 0 to the maximum are displayed. If the maximum is not specified, all values after the minimum are displayed. When both minimum and maximum are specified, the range to be displayed includes the endpoints as well as all values in between. If no = sign is given, command-line parsing fails, and usage information is displayed. If an = sign is given, with no minimum or maximum values, the default of the entire history is displayed.

-interval=<seconds>
: The assumed update interval, in seconds. Assumed times for the the updates are displayed, making the use of the **-time** option together with the **-interval** option redundant.

-notime
Do not display assumed times for the the updates. If more than one of the options **-notime** and **-time** are provided, the final one within the command line parsed determines the display.

-time
Display assumed times for the the updates. If more than one of the options **-notime** and **-time** are provided, the final one within the command line parsed determines the display.

-summary
Display only summary information, not the entire history for each machine.

-s
Same as **-summary**.

15.72.4 Exit Status

condor_updates_stats will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

15.72.5 Examples

Assuming the default of 128 updates kept, and assuming that the update interval is 5 minutes, *condor_updates_stats* displays:

```
$ condor_status -l host1 | condor_updates_stats -\-interval=300
(Reading from stdin)
*** Name/Machine = 'HOST1.cs.wisc.edu' MyType = 'Machine' ***
 Type: Main
   Stats: Total=2277, Seq=2276, Lost=3 (0.13%)
     0 @ Mon Feb 16 12:55:38 2004: Ok
  ...
    28 @ Mon Feb 16 10:35:38 2004: Missed
    29 @ Mon Feb 16 10:30:38 2004: Ok
  ...
   127 @ Mon Feb 16 02:20:38 2004: Ok
```

Within this display, update numbered 27, which occurs later in time than the missed update numbered 28, is Ok. Each change in state, in reverse time order, displays in this condensed version.

15.73 *condor_upgrade_check*

Check a current install of HTCondor for incompatibilites that may cause issues when upgrading to a new major version.

15.73.1 Synopsis

condor_upgrade_check [**–help**]

condor_upgrade_check [**–CE**] [**–all**] [**–ignore** *TAG {TAG...}*] [**–only** *TAG {TAG...}*] [**–tags**] [**–warnings**] [**–no-warnings**] [**–dump**] [**–verbose**]

15.73.2 Description

condor_upgrade_check is a tool intended to be used by administrators before upgrading an HTCondor install to a new major version. This tool will perform various checks for the current installation against known incompatibilities introduced in the Feature series of HTCondor for a given major version. If a check fails, indicating the current install will have issues with an upgrade, then the tool will do its best to suggest a course of action to take.

condor_upgrade_check is intended to be ran on a per-host basis for upgrading a system. Since the CHTC recommends upgrading between major versions in steps (i.e. V23 -> V24 -> V25), the available checks change between major versions. New ones will be added and old ones removed.

Some checks ran by **condor_upgrade_check** are classified as warnings. These checks output warnings about incompatibilites that the tool is not capable of testing and thus give accurrate feedback. Warnings tend to unconditionally output information.

Note: Some checks ran by this tool require the tool to be ran as `root`. If this tool is executed with out `root` privileges or on a Windows host then any checks that require `root` will be skipped.

15.73.3 Options

-h/–help
: Display **condor_upgrade_checks** usage to the terminal

–CE
: Run available checks for installed HTCondor-CE on host

-a/–all
: Run all available checks ignoring the version check to run

-i/–ignore ***TAG [TAG ...]***
: Ignore checks with a matching *TAG*. Takes precendence over **-only**

-o/–only ***TAG [TAG ...]***
: Only run checks with a matching *TAG*. If given the *TAG* `WARNINGS` then only checks classified as warnings will be ran.

-w/–warnings/–no-warnings
: Enable/disable output of checks classified as warnings. Default is enabled.

-t/–tags
: Display all available check *TAGS* to be used by **–only** and **–ignore**

-d/–dump
: Display information about all available checks.

-v/–verbose
: Increase tool verbosity

15.73.4 Examples

Check hosts installed HTCondor for potential issues caused by incompatibilities when upgrading between major versions.

```
condor_upgrade_check
```

Check hosts installed HTCondor-CE for potential issues caused by incompatibilites when upgrading between major versions.

```
condor_upgrade_check -ce
```

List all available check *TAGS*

```
condor_upgrade_check --tags
```

List information about all available checks

```
condor_upgrade_check --dump
```

Run checks while ignoring specific checks for a host installed HTCondor

```
condor_upgrade_check --ignore BAR BAZ
```

Run only checks classified as warnings for a host installed HTCondor

```
condor_upgrade_check --only warnings
```

15.73.5 Exit Status

Returns `0` when tool is finished running. Returns `1` for fatal internal errors.

15.74 *condor_urlfetch*

fetch configuration given a URL

15.74.1 Synopsis

condor_urlfetch [**-<daemon>**] *url local-url-cache-file*

15.74.2 Description

Depending on the command line arguments, *condor_urlfetch* sends the result of a query from the *url* to both standard output and to a file specified by *local-url-cache-file*, or it sends the contents of the file specified by *local-url-cache-file* to standard output.

condor_urlfetch is intended to be used as the program to run when defining configuration, such as in the nonfunctional example:

```
LOCAL_CONFIG_FILE = $(LIBEXEC)/condor_urlfetch -$(SUBSYSTEM) \
  http://www.example.com/htcondor-baseconfig  local.config |
```

The pipe character (|) at the end of this definition of the location of a configuration file changes the use of the definition. It causes the command listed on the right hand side of this assignment statement to be invoked, and standard output becomes the configuration. The value of `$(SUBSYSTEM)` becomes the daemon that caused this configuration to be read. If `$(SUBSYSTEM)` evaluates to `MASTER`, then the URL query always occurs, and the result is sent to standard output as well as written to the file specified by argument *local-url-cache-file*. When `$(SUBSYSTEM)` evaluates to a daemon other than `MASTER`, then the URL query only occurs if the file specified by *local-url-cache-file* does not exist. If the file specified by *local-url-cache-file* does exist, then the contents of this file is sent to standard output.

Note that if the configuration kept at the URL site changes, and reconfiguration is requested, the **-<daemon>** argument needs to be `-MASTER`. This is the only way to guarantee that there will be a query of the changed URL contents, such that they will make their way into the configuration.

15.74.3 Options

-<daemon>
: The upper case name of the daemon issuing the request for the configuration output. If `-MASTER`, then the URL query always occurs. If a daemon other than `-MASTER`, for example `STARTD` or `SCHEDD`, then the URL query only occurs if the file defined by *local-url-cache-file* does not exist.

15.74.4 Exit Status

condor_urlfetch will exit with a status value of 0 (zero) upon success and non zero otherwise.

15.75 *condor_userlog*

Display and summarize job statistics from job log files.

15.75.1 Synopsis

condor_userlog [**-help**] [**-total | -raw**] [**-debug**] [**-evict**] [**-j** *cluster* | *cluster.proc*] [**-all**] [**-hostname**] *logfile* ...

15.75.2 Description

condor_userlog parses the information in job log files and displays summaries for each workstation allocation and for each job. See the *condor_submit* manual page for instructions for specifying that HTCondor write a log file for your jobs.

If **-total** is not specified, *condor_userlog* will first display a record for each workstation allocation, which includes the following information:

Job
: The cluster/process id of the HTCondor job.

Host
: The host where the job ran. By default, the host's IP address is displayed. If **-hostname** is specified, the host name will be displayed instead.

Start Time
: The time (month/day hour:minute) when the job began running on the host.

Evict Time
: The time (month/day hour:minute) when the job was evicted from the host.

Wall Time
The time (days+hours:minutes) for which this workstation was allocated to the job.

Good Time
The allocated time (days+hours:min) which contributed to the completion of this job. If the job exited during the allocation, then this value will equal "Wall Time." Otherwise, it will 0+00:00; self-checkpoint are presently ignored.

CPU Usage
The CPU time (days+hours:min) which contributed to the completion of this job.

condor_userlog will then display summary statistics per host:

Host/Job
The IP address or host name for the host.

Wall Time
The workstation time (days+hours:minutes) allocated by this host to the jobs specified in the query. By default, all jobs in the log are included in the query.

Good Time
The time (days+hours:minutes) allocated on this host which contributed to the completion of the jobs specified in the query.

CPU Usage
The CPU time (days+hours:minutes) obtained from this host which contributed to the completion of the jobs specified in the query.

Avg Alloc
The average length of an allocation on this host (days+hours:minutes).

Avg Lost
The average amount of work lost (days+hours:minutes) when a job was evicted from this host.

Goodput
This percentage is computed as Good Time divided by Wall Time.

Util.
This percentage is computed as CPU Usage divided by Good Time.

condor_userlog will then display summary statistics per job:

Host/Job
The cluster/process id of the HTCondor job.

Wall Time
The total workstation time (days+hours:minutes) allocated to this job.

Good Time
The total time (days+hours:minutes) allocated to this job which contributed to the job's completion.

CPU Usage
The total CPU time (days+hours:minutes) which contributed to this job's completion.

Avg Alloc
The average length of a workstation allocation obtained by this job in minutes (days+hours:minutes).

Avg Lost
The average amount of work lost (days+hours:minutes) when this job was evicted from a host; self-checkpoints are presently ignored.

Goodput
This percentage is computed as Good Time divided by Wall Time.

Util.
This percentage is computed as CPU Usage divided by Good Time.

Finally, *condor_userlog* will display a summary for all hosts and jobs.

15.75.3 Options

-help
Get a brief description of the supported options

-total
Only display job totals

-raw
Display raw data only

-debug
Debug mode

-j
Select a specific cluster or cluster.proc

-evict
Select only allocations which ended due to eviction

-all
Select all clusters and all allocations

-hostname
Display host name instead of IP address

15.75.4 General Remarks

Since the HTCondor job log file format does not contain a year field in the timestamp, all entries are assumed to occur in the current year. Allocations which begin in one year and end in the next will be silently ignored.

15.75.5 Exit Status

condor_userlog will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.76 *condor_userprio*

Manage user priorities

15.76.1 Synopsis

condor_userprio -help

condor_userprio [**-name** *negotiatorname*] [**-pool** *centralmanagerhostname[:portnumber]*] [**Edit option**] | [**Display options** [*username*]] [**-inputfile** *filename*]

15.76.2 Description

condor_userprio either modifies priority-related information or displays priority-related information. Displayed information comes from the accountant log, where the *condor_negotiator* daemon stores historical usage information in the file at `$(SPOOL)`/Accountantnew.log. Which fields are displayed changes based on command line arguments. *condor_userprio* with no arguments, lists the active users along with their priorities, in increasing priority order. The **-all** option can be used to display more detailed information about each user, resulting in a rather wide display, and includes the following columns:

Effective Priority
: The effective priority value of the user, which is used to calculate the user's share when allocating resources. A lower value means a higher priority, and the minimum value (highest priority) is 0.5. The effective priority is calculated by multiplying the real priority by the priority factor.

Real Priority
: The value of the real priority of the user. This value follows the user's resource usage.

Priority Factor
: The system administrator can set this value for each user, thus controlling a user's effective priority relative to other users. This can be used to create different classes of users.

Res Used
: The number of resources currently used.

Accumulated Usage
: The accumulated number of resource-hours used by the user since the usage start time.

Usage Start Time
: The time since when usage has been recorded for the user. This time is set when a user job runs for the first time. It is reset to the present time when the usage for the user is reset.

Last Usage Time
: The most recent time a resource usage has been recorded for the user.

By default only users for whom usage was recorded in the last 24 hours, or whose priority is greater than the minimum are listed.

The **-pool** option can be used to contact a different central manager than the local one (the default).

Options that do not begin with a - are treated as a username and results will restricted to users that match the given name. More than one username can be specified.

For security purposes of authentication and authorization, specifying an Edit Option requires the ADMINISTRATOR level of access.

15.76.3 Options

-help
Display usage information and exit.

-name ***negotiatorname***
When querying ads from the *condor_collector*, only retrieve ads that came from the negotiator with the given name.

-pool ***centralmanagerhostname[:portnumber]***
Contact the specified *centralmanagerhostname* with an optional port number, instead of the local central manager. This can be used to check other pools. NOTE: The host name (and optional port) specified refer to the host name (and port) of the *condor_negotiator* to query for user priorities. This is slightly different than most HTCondor tools that support a **-pool** option, and instead expect the host name (and port) of the *condor_collector*.

-inputfile ***filename***
Introduced for debugging purposes, read priority information from *filename*. The contents of *filename* are expected to be the same as captured output from running a `condor_userprio -long` command.

-delete ***username***
(Edit option) Remove the specified *username* from HTCondor's accounting.

-resetall
(Edit option) Reset the accumulated usage of all the users to zero.

-resetusage ***username***
(Edit option) Reset the accumulated usage of the user specified by *username* to zero.

-setaccum ***username value***
(Edit option) Set the accumulated usage of the user specified by *username* to the specified floating point *value*.

-setbegin ***username value***
(Edit option) Set the begin usage time of the user specified by *username* to the specified *value*.

-setfactor ***username value***
(Edit option) Set the priority factor of the user specified by *username* to the specified *value*.

-setlast ***username value***
(Edit option) Set the last usage time of the user specified by *username* to the specified *value*.

-setprio ***username value***
(Edit option) Set the real priority of the user specified by *username* to the specified *value*.

-setfloor ***username value***
(Edit option) Set the floor for the user specified by *username* to the specified *value*. This value is the sum of the SlotWeight (See: SLOT_WEIGHT in *condor_startd Configuration File Macros*) of all running jobs. By default, the slot weight of a running job is the number of cores allocated to that job.

-setceil ***username value***
(Edit option) Set the ceiling for the user specified by *username* to the specified *value*. This value is the sum of the SlotWeight (See: SLOT_WEIGHT in *condor_startd Configuration File Macros*) of all running jobs. By default, the slot weight of a running job is the number of cores allocated to that job.

-activefrom ***month day year***
(Display option) Display information for users who have some recorded accumulated usage since the specified date.

-all
(Display option) Display all available fields about each group or user.

-allusers
(Display option) Display information for all the users who have some recorded accumulated usage.

-negotiator
(Display option) Force the query to come from the negotiator instead of the collector.

-autoformat[:jlhVr,tng] ***attr1 [attr2 ...]*** **or -af[:jlhVr,tng]** ***attr1 [attr2 ...]***
(Display option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

j print the job ID as the first field,

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may not be used together. The **l** and **h** characters may not be used together.

-constraint <*expr*>
(Display option) To be used in conjunction with the **-long -modular** or the **-autoformat** options. Displays users and groups that match the <`expr`>.

-debug[:<opts>]
(Display option) Without **:<opts>** specified, use configured debug level to send debugging output to `stderr`. With **:<opts>** specified, these options are debug levels that override any configured debug levels for this command's execution to send debugging output to `stderr`.

-flat
(Display option) Display information such that users within hierarchical groups are not listed with their group.

-getreslist ***username***
(Display option) Display all the resources currently allocated to the user specified by *username*.

-grouporder
(Display option) Display submitter information with accounting group entries at the top of the list, and in breadth-first order within the group hierarchy tree.

-grouprollup
(Display option) For hierarchical groups, the display shows sums as computed for groups, and these sums include sub groups.

-hierarchical
(Display option) Display information such that users within hierarchical groups are listed with their group.

-legacy
(Display option) For use with the **-long** option, displays attribute names and values as a single ClassAd.

-long
(Display option) A verbose output which displays entire ClassAds.

-modular
(Display option) Modifies the display when using the **-long** option, such that attribute names and values are shown as distinct ClassAds.

-most
(Display option) Display fields considered to be the most useful. This is the default set of fields displayed.

-priority
(Display option) Display fields with user priority information.

-quotas
(Display option) Display fields relevant to hierarchical group quotas.

-usage
(Display option) Display usage information for each group or user.

15.76.4 Examples

Example 1 Since the output varies due to command line arguments, here is an example of the default output for a pool that does not use Hierarchical Group Quotas. This default output is the same as given with the **-most** Display option.

```
Last Priority Update:  1/19 13:14
                      Effective   Priority   Res   Total Usage  Time Since
User Name              Priority    Factor   In Use (wghted-hrs) Last Usage
---------------------- ---------- --------- ------ ------------ ----------
www-cndr@cs.wisc.edu         0.56      1.00      0    591998.44    0+16:30
joey@cs.wisc.edu             1.00      1.00      1       990.15 <now>
suzy@cs.wisc.edu             1.53      1.00      0       261.78    0+09:31
leon@cs.wisc.edu             1.63      1.00      2     12597.82 <now>
raj@cs.wisc.edu              3.34      1.00      0      8049.48    0+01:39
jose@cs.wisc.edu             3.62      1.00      4     58137.63 <now>
betsy@cs.wisc.edu           13.47      1.00      0      1475.31    0+22:46
petra@cs.wisc.edu          266.02    500.00      1    288082.03 <now>
carmen@cs.wisc.edu         329.87     10.00    634   2685305.25 <now>
carlos@cs.wisc.edu         687.36     10.00      0     76555.13    0+14:31
ali@proj1.wisc.edu        5000.00  10000.00      0      1315.56    0+03:33
apu@nnland.edu            5000.00  10000.00      0       482.63    0+09:56
pop@proj1.wisc.edu       26688.11  10000.00      1     49560.88 <now>
franz@cs.wisc.edu        29352.06    500.00    109    600277.88 <now>
martha@nnland.edu        58030.94  10000.00      0     48212.79    0+12:32
```

(continues on next page)

(continued from previous page)

```
izzi@nnland.edu              62106.40   10000.00        0      6569.75     0+02:26
marta@cs.wisc.edu            62577.84     500.00       29    193706.30 <now>
kris@proj1.wisc.edu         100597.94   10000.00        0     20814.24     0+04:26
boss@proj1.wisc.edu         318229.25   10000.00        3    324680.47 <now>
---------------------- ------------ --------- ------ ------------ ----------
Number of users: 19                                784   4969073.00     0+23:59
```

Example 2 This is an example of the default output for a pool that uses hierarchical groups, and the groups accept surplus. This leads to a very wide display.

```
$ condor_userprio -pool crane.cs.wisc.edu -allusers
Last Priority Update:  1/19 13:18
Group                                 Config     Use    Effective   Priority   Res␣
↪Total Usage  Time Since
  User Name                            Quota   Surplus   Priority    Factor   In Use␣
↪(wghted-hrs) Last Usage
------------------------------------ --------- ------- ------------ --------- ------ ----
↪-------- ----------
<none>                                   0.00     yes                   1.00      0    ␣
↪    6.78    9+03:52
  johnsm@crane.cs.wisc.edu                                    0.50       1.00      0    ␣
↪    6.62    9+19:42
  John.Smith@crane.cs.wisc.edu                                0.50       1.00      0    ␣
↪    0.02    9+03:52
  Sedge@crane.cs.wisc.edu                                     0.50       1.00      0    ␣
↪    0.05   13+03:03
  Duck@crane.cs.wisc.edu                                      0.50       1.00      0    ␣
↪    0.02   31+00:28
  other@crane.cs.wisc.edu                                     0.50       1.00      0    ␣
↪    0.04   16+03:42
Duck                                     2.00      no                   1.00      0    ␣
↪    0.02   13+02:57
  goose@crane.cs.wisc.edu                                     0.50       1.00      0    ␣
↪    0.02   13+02:57
Sedge                                    4.00      no                   1.00      0    ␣
↪    0.17    9+03:07
  johnsm@crane.cs.wisc.edu                                    0.50       1.00      0    ␣
↪    0.13    9+03:08
  Half@crane.cs.wisc.edu                                      0.50       1.00      0    ␣
↪    0.02   31+00:02
  John.Smith@crane.cs.wisc.edu                                0.50       1.00      0    ␣
↪    0.05    9+03:07
  other@crane.cs.wisc.edu                                     0.50       1.00      0    ␣
↪    0.01   28+19:34
------------------------------------ --------- ------- ------------ --------- ------ ----
↪-------- ----------
Number of users: 10                                ByQuota                        0    ␣
↪    6.97
```

15.76.5 Exit Status

condor_userprio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.77 *condor_vacate*

Vacate jobs that are running on the specified hosts

15.77.1 Synopsis

condor_vacate [**-help | -version**]

condor_vacate [**-graceful | -fast**] [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

15.77.2 Description

condor_vacate causes HTCondor force jobs to vacate from a given set of machines. The job(s) remains in the submitting machine's job queue.

Given the (default) **-graceful** option, jobs are killed and HTCondor restarts the job from the beginning somewhere else. *condor_vacate* has no effect on a machine with no HTCondor job currently running.

There is generally no need for the user or administrator to explicitly run *condor_vacate*. HTCondor takes care of jobs in this way automatically following the policies given in configuration files.

15.77.3 Options

-help
: Display usage information

-version
: Display version information

-graceful
: Give the job a change to shut down cleanly, then soft-kill it.

-fast
: Hard-kill jobs instead of giving them to shut down cleanly.

-debug
: Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-pool *centralmanagerhostname[:portnumber]*
: Specify a pool by giving the central manager's host name and an optional port number

-name *hostname*
: Send the command to a machine identified by *hostname*

hostname
: Send the command to a machine identified by *hostname*

-addr *"<a.b.c.d:port>"*
Send the command to a machine's master located at *"<a.b.c.d:port>"*

"<a.b.c.d:port>"
Send the command to a machine located at *"<a.b.c.d:port>"*

-constraint *expression*
Apply this command only to machines matching the given ClassAd *expression*

-all
Send the command to all machines in the pool

15.77.4 Exit Status

condor_vacate will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.77.5 Examples

To send a *condor_vacate* command to two named machines:

```
$ condor_vacate  robin cardinal
```

To send the *condor_vacate* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
$ condor_vacate -pool condor.cae.wisc.edu -name cae17
```

15.78 *condor_vacate_job*

vacate jobs in the HTCondor queue from the hosts where they are running

15.78.1 Synopsis

condor_vacate_job [**-help | -version**]

condor_vacate_job [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] [**-fast**] *cluster...* | *cluster.process...* | *user...* | **-constraint** *expression ...*

condor_vacate_job [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*] | [**-addr** *"<a.b.c.d:port>"*] [**-fast**] **-all**

15.78.2 Description

condor_vacate_job finds one or more jobs from the HTCondor job queue and vacates them from the host(s) where they are currently running. The jobs remain in the job queue and return to the idle state.

A running job running will be sent a soft kill signal (SIGTERM by default, or whatever is defined as the `SoftKillSig` in the job ClassAd), and HTCondor will restart the job from the beginning somewhere else.

If the **-fast** option is used, the job(s) will be immediately killed.

If the **-name** option is specified, the named *condor_schedd* is targeted for processing. If the **-addr** option is used, the *condor_schedd* at the given address is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be vacated are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can vacate the job.

Using *condor_vacate_job* on jobs which are not currently running has no effect.

15.78.3 Options

-help
Display usage information

-version
Display version information

-pool ***centralmanagerhostname[:portnumber]***
Specify a pool by giving the central manager's host name and an optional port number

-name ***scheddname***
Send the command to a machine identified by *scheddname*

-addr ***"<a.b.c.d:port>"***
Send the command to a machine located at *"<a.b.c.d:port>"*

cluster
Vacate all jobs in the specified cluster

cluster.process
Vacate the specific job in the cluster

user
Vacate jobs belonging to specified user

-constraint ***expression***
Vacate all jobs which match the job ClassAd expression constraint

-all
Vacate all the jobs in the queue

-fast
Perform a fast vacate and hard kill the jobs

15.78.4 General Remarks

Do not confuse *condor_vacate_job* with *condor_vacate*. *condor_vacate* is given a list of hosts to vacate, regardless of what jobs happen to be running on them. Only machine owners and administrators have permission to use *condor_vacate* to evict jobs from a given host. *condor_vacate_job* is given a list of job to vacate, regardless of which hosts they happen to be running on. Only the owner of the jobs or queue super users have permission to use *condor_vacate_job*.

15.78.5 Examples

To vacate job 23.0:

```
$ condor_vacate_job 23.0
```

To vacate all jobs of a user named Mary:

```
$ condor_vacate_job mary
```

To vacate all vanilla universe jobs owned by Mary:

```
$ condor_vacate_job -constraint 'JobUniverse == 5 && Owner == "mary"'
```

Note that the entire constraint, including the quotation marks, must be enclosed in single quote marks for most shells.

15.78.6 Exit Status

condor_vacate_job will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.79 *condor_version*

print HTCondor version and platform information

15.79.1 Synopsis

condor_version [**-help**]

condor_version [**-arch**] [**-opsys**] [**-syscall**]

15.79.2 Description

With no arguments, *condor_version* prints the currently installed HTCondor version number and platform information. The version number includes a build identification number, as well as the date built.

15.79.3 Options

-help
Print usage information

-arch
Print this machine's ClassAd value for `Arch`

-opsys
Print this machine's ClassAd value for `OpSys`

-syscall
Get any requested version and/or platform information from the `libcondorsyscall.a` that this HTCondor pool is configured to use, instead of using the values that are compiled into the tool itself. This option may be used in combination with any other options to modify where the information is coming from.

15.79.4 Exit Status

condor_version will exit with a status value of 0 (zero) upon success, and it should never exit with a failing value.

15.80 *condor_wait*

Wait for jobs to finish

15.80.1 Synopsis

condor_wait [**-help | -version**]

condor_wait [**-debug**] [**-status**] [**-echo**] [**-wait** *seconds*] [**-num** *number-of-jobs*] *log-file* [**job ID**]

15.80.2 Description

condor_wait watches a job event log file (created with the **log** command within a submit description file) and returns when one or more jobs from the log have completed or aborted.

Because *condor_wait* expects to find at least one job submitted event in the log file, at least one job must have been successfully submitted with *condor_submit* before *condor_wait* is executed.

condor_wait will wait forever for jobs to finish, unless a shorter wait time is specified.

15.80.3 Options

-help
Display usage information

-version
Display version information

-debug
Show extra debugging information.

-status
Show job start and terminate information.

-echo
Print the events out to `stdout`.

-wait ***seconds***
Wait no more than the integer number of *seconds*. The default is unlimited time.

-num ***number-of-jobs***
Wait for the integer *number-of-jobs* jobs to end. The default is all jobs in the log file.

log file
The name of the log file to watch for information about the job.

job ID
A specific job or set of jobs to watch. If the **job ID** is only the job ClassAd attribute `ClusterId`, then *condor_wait* waits for all jobs with the given `ClusterId`. If the **job ID** is a pair of the job ClassAd attributes, given by `ClusterId.ProcId`, then *condor_wait* waits for the specific job with this **job ID**. If this option is not specified, all jobs that exist in the log file when *condor_wait* is invoked will be watched.

15.80.4 General Remarks

condor_wait is an inexpensive way to test or wait for the completion of a job or a whole cluster, if you are trying to get a process outside of HTCondor to synchronize with a job or set of jobs.

It can also be used to wait for the completion of a limited subset of jobs, via the **-num** option.

15.80.5 Examples

```
$ condor_wait logfile
```

This command waits for all jobs that exist in `logfile` to complete.

```
$ condor_wait logfile 40
```

This command waits for all jobs that exist in `logfile` with a job ClassAd attribute `ClusterId` of 40 to complete.

```
$ condor_wait -num 2 logfile
```

This command waits for any two jobs that exist in `logfile` to complete.

```
$ condor_wait logfile 40.1
```

This command waits for job 40.1 that exists in `logfile` to complete.

```
$ condor_wait -wait 3600 logfile 40.1
```

This waits for job 40.1 to complete by watching `logfile`, but it will not wait more than one hour (3600 seconds).

15.80.6 Exit Status

condor_wait exits with 0 if and only if the specified job or jobs have completed or aborted. *condor_wait* returns 1 if unrecoverable errors occur, such as a missing log file, if the job does not exist in the log file, or the user-specified waiting time has expired.

15.81 *condor_watch_q*

Track the status of jobs over time.

15.81.1 Synopsis

condor_watch_q [**-help**]

condor_watch_q [*general options*] [*display options*] [*behavior options*] [*tracking options*]

15.81.2 Description

condor_watch_q is a tool for tracking the status of jobs over time without repeatedly querying the *condor_schedd*. It does this by reading job event log files. These files may be specified directly (the `-files` option), or indirectly via a single query to the *condor_schedd* when **condor_watch_q** starts up (options like `-users` or `-clusters`).

condor_watch_q provides a variety of options for output formatting, including: colorized output, tabular information, progress bars, and text summaries. These display options are highly-customizable via command line options.

condor_watch_q also provides a minimal language for exiting when certain conditions are met by the tracked jobs. For example, it can be configured to exit when all of the tracked jobs have terminated.

15.81.3 Examples

If no users, cluster ids, or event logs are given, **condor_watch_q** will default to tracking all of the current user's jobs. Thus, with no arguments,

```
condor_watch_q
```

will track all of your currently-active clusters.

To track jobs from a specific cluster, use the `-clusters` option, passing the cluster ID:

```
condor_watch_q -clusters 12345
```

To track jobs from a specific user, use the `-users` option, passing the user's name the actual query will be the for the `Owner` job ad attribute):

```
condor_watch_q -users jane
```

To track jobs from a specific event log file, use the `-files` option, passing the path to the event log:

```
condor_watch_q -users /home/jane/events.log
```

To track jobs from a specific batch, use the `-batches` option, passing the batch name:

```
condor_watch_q -batches BatchOfJobsFromTuesday
```

All of the above "tracking" options can be used together, and multiple values may be passed to each one. For example, to track all of the jobs that are: owned by `jane` or `jim`, in cluster `12345`, or in the event log `/home/jill/events.log`, run

```
condor_watch_q -users jane jim -clusters 12345 -files /home/jill/events.log
```

By default, **condor_watch_q** will never exit on its own (unless it encounters an error or it is not tracking any jobs). You can tell it to exit when certain conditions are met. For example, to exit with status 0 when all of the jobs it is tracking are done or with status 1 when any job is held, you could run

```
condor_watch_q -exit all,done,0 -exit any,held,1
```

15.81.4 Options

General Options

-help
Display the help message and exit.

-debug
Causes debugging information to be sent to `stderr`.

Tracking Options

These options control which jobs **condor_watch_q** will track, and how it discovers them.

-users USER [USER ...]
Choose which users to track jobs for. All of the user's jobs will be tracked. One or more user names may be passed.

-clusters CLUSTER_ID [CLUSTER_ID ...]
Which cluster IDs to track jobs for. One or more cluster ids may be passed.

-files FILE [FILE ...]
Which job event log files (i.e., the `log` file from `condor_submit`) to track jobs from. One or more file paths may be passed.

-batches BATCH_NAME [BATCH_NAME ...]
Which job batch names to track jobs for. One or more batch names may be passed.

-collector COLLECTOR
Which collector to contact to find the schedd, if needed. Defaults to the local collector.

-schedd SCHEDD
Which schedd to contact for queries, if needed. Defaults to the local schedd.

Behavior Options

-exit GROUPER,JOB_STATUS[,EXIT_STATUS]
Specify conditions under which condor_watch_q should exit. `GROUPER` is one of `all`, `any` or `none`. `JOB_STATUS` is one of `active`, `done`, `idle`, or `held`. The "active" status means "in the queue", and includes jobs in the idle, running, and held states. `EXIT_STATUS` may be any valid exit status integer. To specify multiple exit conditions, pass this option multiple times. **condor_watch_q** will exit when any of the conditions are satisfied.

Display Options

These options control how **condor_watch_q** formats its output. Many of them are "toggles": `-x` enables option "x", and `-no-x` disables it.

-groupby {batch, log, cluster}
How to group jobs into rows for display in the table. Must be one of `batch` (group by job batch name), `log` (group by event log file path), or `cluster` (group by cluster ID). Defaults to `batch`.

-table/-no-table
Enable/disable the table. Enabled by default.

-progress/-no-progress
Enable/disable the progress bar. Enabled by default.

-row-progress/-no-row-progress
Enable/disable the progress bar for each row. Enabled by default.

-summary/-no-summary
Enable/disable the summary line. Enabled by default.

-summary-type {totals, percentages}
Choose what to display on the summary line, `totals` (the number of each jobs in each state), or `percentages` (the percentage of jobs in each state, of the total number of tracked jobs) By default, show `totals`.

-updated-at/-no-updated-at
Enable/disable the "updated at" line. Enabled by default.

-abbreviate/-no-abbreviate
Enable/disable abbreviating path components to the shortest somewhat-unique prefix. Disabled by default.

-color/-no-color
Enable/disable colored output. Enabled by default if connected to a tty. Disabled on Windows if colorama is not available (https://pypi.org/project/colorama/).

-refresh/-no-refresh
Enable/disable refreshing output. If refreshing is disabled, output will be appended instead. Enabled by default if connected to a tty.

15.81.5 Exit Status

Returns `0` when sent a SIGINT (keyboard interrupt).

Returns `0` if no jobs are found to track.

Returns `1` for fatal internal errors.

Can be configured via the `-exit` option to return any valid exit status when a certain condition is met.

15.81.6 Author

Center for High Throughput Computing, University of Wisconsin-Madison

15.82 *condor_who*

Display information about owners of jobs and jobs running on an execute machine

15.82.1 Synopsis

condor_who [*help options*] [*address options*] [*display options*]

15.82.2 Description

condor_who queries and displays information about the user that owns the jobs running on a machine. It is intended to be run on an execute machine.

The options that may be supplied to *condor_who* belong to three groups:

- **Help options** provide information about the *condor_who* tool.
- **Address options** allow destination specification for query.
- **Display options** control the formatting and which of the queried information to display.

At any time, only one **help option** and one **address option** may be specified. Any number of **display options** may be specified.

condor_who obtains its information about jobs by talking to one or more *condor_startd* daemons. So, *condor_who* must identify the command port of any *condor_startd* daemons. An **address option** provides this information. If no **address option** is given on the command line, then *condor_who* searches using this ordering:

1. A defined value of the environment variable `CONDOR_CONFIG` specifies the directory where log and address files are to be scanned for needed information.
2. With the aim of finding all *condor_startd* daemons, *condor_who* utilizes the same algorithm it would using the **-allpids** option. The Linux *ps* or the Windows *tasklist* program obtains all PIDs. As Linux root or Windows administrator, the Linux *lsof* or the Windows *netstat* identifies open sockets and from there the PIDs of listen sockets. Correlating the two lists of PIDs results in identifying the command ports of all *condor_startd* daemons.

15.82.3 Options

-help
(help option) Display usage information

-daemons
(help option) Display information about the daemons running on the specified machine, including the daemon's PID, IP address and command port

-diagnostic
(help option) Display extra information helpful for debugging

-verbose
(help option) Display PIDs and addresses of daemons

-address ***hostaddress***
(address option) Identify the *condor_startd* host address to query

-allpids
(address option) Query all local *condor_startd* daemons

-logdir ***directoryname***
(address option) Specifies the directory containing log and address files that *condor_who* will scan to search for command ports of *condor_start* daemons to query

-pid ***PID***
(address option) Use the given *PID* to identify the *condor_startd* daemon to query

-long
(display option) Display entire ClassAds

-wide
(display option) Displays fields without truncating them in order to fit screen width

-format ***fmt attr***
(display option) Display attribute *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)`-style conversion specifier. Attributes must be from the resource ClassAd. Expressions are ClassAd expressions and may refer to attributes in the resource ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. %r prints the unevaluated, or raw values. The conversion specifier must match the type of the attribute or expression. %s is suitable for strings such as `Name`, %d for integers such as `LastHeardFrom`, and %f for floating point numbers such as `LoadAvg`. %v identifies the type of the attribute, and then prints the value in an appropriate format. %V identifies the type of the attribute, and then prints the value in an appropriate format as it would appear in the **-long** format. As an example, strings used with %V will have quote marks. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)`-style formats, one may include other text that will be reproduced directly. A format without any conversion specifiers may be specified, but an attribute is still required. Include a backslash followed by an 'n' to specify a line break.

-autoformat[:lhVr,tng] ***attr1 [attr2 ...]*** **or -af[:lhVr,tng]** ***attr1 [attr2 ...]***
(display option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character

and formatting qualifiers to deviate the output formatting from the default:

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may not be used together. The **l** and **h** characters may not be used together.

15.82.4 Examples

Example 1 Sample output from the local machine, which is running a single HTCondor job. Note that the output of the `PROGRAM` field will be truncated to fit the display, similar to the artificial truncation shown in this example output.

```
$ condor_who

OWNER                    CLIENT            SLOT JOB RUNTIME    PID    PROGRAM
smith1@crane.cs.wisc.edu crane.cs.wisc.edu    2 320.0 0+00:00:08 7776 D:\scratch\condor\
↪execut
```

Example 2 Verbose sample output.

```
$ condor_who -verbose

LOG directory "D:\scratch\condor\master\test/log"

Daemon       PID      Exit       Addr                     Log, Log.Old
------       ---      ----       ----                     ---, -------
Collector    6788               <128.105.136.32:7977> CollectorLog, CollectorLog.old
Credd        8148               <128.105.136.32:9620> CredLog, CredLog.old
Master       5976               <128.105.136.32:64980> MasterLog,
Match MatchLog, MatchLog.old
Negotiator   6600 NegotiatorLog, NegotiatorLog.old
Schedd       6336               <128.105.136.32:64985> SchedLog, SchedLog.old
Shadow ShadowLog,
Slot1 StarterLog.slot1,
Slot2        7272               <128.105.136.32:65026> StarterLog.slot2,
Slot3 StarterLog.slot3,
Slot4 StarterLog.slot4,
SoftKill SoftKillLog,
Startd       7416               <128.105.136.32:64984> StartLog, StartLog.old
Starter StarterLog,
```

(continues on next page)

(continued from previous page)

```
TOOL                                                    TOOLLog,

OWNER                    CLIENT            SLOT JOB RUNTIME    PID    PROGRAM
smith1@crane.cs.wisc.edu crane.cs.wisc.edu    2 320.0 0+00:01:28 7776 D:\scratch\condor\
↪execut
```

15.82.5 Exit Status

condor_who will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.83 *get_htcondor*

Install and configure HTCondor on Linux machines.

15.83.1 Synopsis

get_htcondor <-h | --help>

get_htcondor [**–[no-]dry-run**] [**--channel** *name*] [**--minicondor** | [**--central-manager** | **--submit** | **--execute**] *central-manager-name*] [**--shared-filesystem-domain** *filesystem-domain-name*]

get_htcondor --dist

15.83.2 Description

This tool installs and configure HTCondor on Linux machines. See https://htcondor.readthedocs.io/en/latest/getting-htcondor for detailed instructions. This page is intended as a quick reference to its options; it also includes a section about the reasons for the configurations it installs.

15.83.3 Options

-help
: Print a usage reminder.

--dry-run
: Do not issue commands, only print them. [default]

--no-dry-run
: Issue all the commands needed to install HTCondor.

--channel *name*
: Specify channel *name* to install; *name* may be `current`, the most recent release with new features [default] or `stable`, the most recent release with only bug-fixes

--dist
: Display the detected operating system and exit.

--minicondor
Configure as a single-machine ("mini") HTCondor. [default]

--central-manager *central-manager-name*

--submit *central-manager-name*

--execute *central-manager-name*

Configure this installation with the central manager, submit, or execute role.

--shared-filesystem-domain *filesystem-domain-name*

Configure this installation to assume that machines specifying the same *filesystem-domain-name* share a filesystem.

15.83.4 Exit Status

On success, exits with code 0. Failures detected by **get_htcondor** will result in exit code 1. Other failures may have other exit codes.

15.83.5 Installed Configuration

This tool may install four different configurations. We discuss the single-machine configuration first, and then the three parts of the multi-machine configuration as a group. Our goal is to document the reasoning behind the details, because the details can obscure that reasoning, and because the details will change as we continue to improve HTCondor.

As a general note, the configurations this tool installs make extensive use of metaknobs, lines in HTCondor configuration files that look like `use x : y`. To determine exactly what configuration a metaknob sets, run `condor_config_val use x:y`.

Single-Machine Installation

The single-machine installation performed by *get_htcondor* uses the `minicondor` package. (A "mini" HTCondor is a single-machine HTCondor system installed with administrative privileges.) Because the different roles in the HTCondor system are all on the same machine, we configure all network communications to occur over the loopback device, where we don't have to worry about eavesdropping or requiring encryption. We use the `FS` method, which depends on the local filesytem, to identify which user is attempting to connect, and restrict access correspondingly.

The *get_htcondor* tool installs the standard minicondor package from the HTCondor repositories; see the file it creates, `/etc/condor/config.d/00-minicondor`, for details.

Multi-Machine Installation

Because the three roles must communicate over the network to form a complete pool in this case,, security is a much bigger concern; we therefore require authentication and encryption on every connection. Thankfully, almost all of the network communication is daemon-to-daemon, so we don't have to burden individual users with that aspect of security. Instead, users submit jobs on the submit-role machine, using `FS` to authenticate. Users may also need to contact the central manager (when running `condor_status`, for example), but they never need to write anything to it, so we've configured authentication for read-only commands to be optional.

Daemon-to-daemon communication is authenticated with the IDTOKENS method. (If a user needs to submit jobs remotely, they can also use the IDTOKENS method, it's just more work; see `condor_token_fetch`.) Each role installed by this tool has a copy of the password, which is used to generate an IDTOKEN, which is used for all daemon-to-daemon

authentication; both the password and the IDTOKEN can only be read by privileged processes. An IDTOKEN can only be validated by the holder of the corresponding password, so each daemon in the pool has to have both.

This tool installs the role-specific configuration in the files `/etc/condor/config.d/01-central-manager.config`, `/etc/condor/config.d/01-submit.config`, and `/etc/condor/config.d/01-execute.config`; consult them for details.

15.84 gidd_alloc

find a GID within the specified range which is not used by any process

15.84.1 Synopsis

gidd_alloc *min-gid max-gid*

15.84.2 Description

This program will scan the alive PIDs, looking for which GID is unused in the supplied, inclusive range specified by the required arguments *min-gid* and *max-gid*. Upon finding one, it will add the GID to its own supplementary group list, and then scan the PIDs again expecting to find only itself using the GID. If no collision has occurred, the program exits, otherwise it retries.

15.84.3 General Remarks

This is a program only available for the Linux ports of HTCondor.

15.84.4 Exit Status

gidd_alloc will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

15.85 *htcondor*

Manage HTCondor jobs, job sets, dags, event logs, and resources

15.85.1 Synopsis

htcondor [**-h** | **--help**] [**-v** | **-q**]

htcondor job *submit* [**--resource** *resource-type*] [**--runtime** *time-seconds*] [**--email** *email-address*] submit_file
htcondor job *status* [**--resource** *resource-type*] [**--skip-history**] job_id
htcondor job *resources* [**--resource** *resource-type*] [**--skip-history**] job_id

htcondor jobset *submit* description-file
htcondor jobset *list* [**--allusers**]

htcondor jobset *status* job-set-name [**--owner** *user-name*] [**--nobatch**] [**--skip-history**]
htcondor jobset *remove* job-set-name [**--owner** *user-name*]

htcondor dag *submit* dag-file
htcondor dag *status* dagman-job-id

htcondor eventlog *read* [**-csv** | **-json**] [**--groupby attribute**] eventlog
htcondor eventlog *follow* [**-csv** | **-json**] [**--groupby attribute**] eventlog

15.85.2 Description

htcondor is a tool for managing HTCondor jobs, job sets, resources, event logs, and DAGs. It can replace *condor_submit*, *condor_submit_dag*, *condor_q*, *condor_status*, and *condor_userlog*, as well as all-new functionality and features. The user interface is more consistent than its predecessor tools.

The first argument of the *htcondor* command (ignoring any global options) is the *noun* representing an object in the HTCondor system to be operated on. The nouns include an individual *job*, *jobset*, *eventlog*, or a *dag*. Each noun is then followed by a noun-specific *verb* that describe the operation on that noun.

One of the following optional global option may appear before the noun:

15.85.3 Global Options

htcondor -h, htcondor --help
Display the help message. Can also be specified after any verb to display the options available for each verb.

htcondor -q ...
Reduce verbosity of log messages.

htcondor -v ...
Increase verbosity of log messages.

A noun-specific verb appears after each noun; the verbs are sorted by noun in the list, which includes with their individual option flags.

15.85.4 Job Verbs

htcondor job submit *submit_file*
Takes as an argument a submit file in the *condor_submit* job submit description language, and places a new job in an Access Point

htcondor job submit options

htcondor job submit --resource *resource_type submit_file*
Resource type used to run this job. Currently supports `Slurm` and `EC2`. Assumes the necessary setup is complete and security tokens available.

htcondor job submit --runtime *runtime_in_seconds submit_file*
Amount of time in seconds to allocate resources. Used in conjunction with the *--resource* flag.

htcondor job submit --email *address submit_file*
Email address to receive notification messages. Used in conjunction with the *--resource* flag.

htcondor job status
Takes as an argument a job id in the form of clusterid.procid, and returns a human readable presentation of the status of that job.

job status option

htcondor job status --skip-history *job.id*

Passed to the *status* verb to skip checking history if job not found in the active job queue.

htcondor job resources
Takes as an argument a job id in the form of clusterid.procid, and returns a human readable presentation the machine resource used by this job.

15.85.5 Jobset Verbs

htcondor jobset submit ***submit_file***
Takes as an argument a submit file in the *condor_submit* job submit description language, and places a new job set in an Access Point

htcondor jobset list
Succinctly lists all the jobsets in the queue which are owned by the current user.

htcondor jobset list options

htcondor jobset list --allusers
Shows jobs from all users, not just those owned by the current user.

htcondor jobset status ***submit_file***
Takes as an argument a job set name, and shows detailed information about that job set.

htcondor jobset status options

htcondor jobset status --nobatch
Shows jobs in a more detailed view, one line per job

htcondor jobset status --owner ***ownername***
Shows jobs from the specified job owner.

htcondor jobset status --skiphistory
Shows detailed information only about active jobs in the queue, and ignore historical jobs which have left the queue. This runs much faster.

htcondor jobset remove ***job_name***
Takes as an argument a *job_name* in the queue, and removes it from the Access Point.

htcondor jobsets remove options

htcondor jobset remove --owner=owner_name Removes all jobs owned by the given owner.

15.85.6 Eventlog Verbs

htcondor eventlog read *logfile*

Takes as an argument an event log to process. It may be the per-job or per-jobset eventlog, which was specified by the *log = some_file* in the submit description language. For a dag, it may also be the *nodes.log* file that all dags generate. Or, if the global event log is enabled by an administrator with the *EVENT_LOG* configuration knob, it may be the global event log, containing information about all jobs on the Access point.

Given this file, *htcondor eventlog read* returns information about all the contained jobs, and their status. It runs much faster than *condor_history*, because these logs are more concise than the history files. Unlike *condor_history*, it will also show information about jobs that have not yet left the queue.

htcondor eventlog follow *logfile*

Takes as an argument an event log to process, as above, but instead of processing that file to completion, it does the equivalent of *tail -f*, and runs until interruption, emitting information about jobs as it appears in the file.

Eventlog Options

--csv
By default, *htcondor eventlog read* emits a table of information in human readable format. With this option, the output is in a command separated value format, suitable for injestion by a spreadsheet or database.

--json
Emits output in the json format. Only one of **-csv** or **-json** should be given.

--group-by attributeName
With a job ad attribute name, instead of one line per job, emit one line summarizing all jobs that share the same value for the attribute name given. In the OSG, the GLIDEIN_SITE attribute is injected into all jobs, so one can quickly get a count of all jobs running, idle and exitted per site by using this option.

15.85.7 Examples

```
$ htcondor eventlog read logfile

Job       Host            Start Time   Evict Time   Evictions   Wall Time     Good Time  ␣
↪   CPU Usage
19989.0   slot1_1@speedy  5/18 12:34   5/18 12:54   0           0+00:20:00    0+00:20:00␣
↪   0+00:00:00
19990.0   slot1_1@lumpy   5/22 18:51   5/22 18:51   1           0+00:02:00    0+00:00:00␣
↪   0+00:00:43
20003.0   slot1_1@chtc    8/9 23:33    8/9 23:37    1           0+00:04:00    0+00:00:00␣
↪   0+00:00:00
20004.0   slot1_1@wisc    8/9 23:38    8/9 23:58    0           0+00:20:00    0+00:20:00␣
↪   0+00:00:00
```

15.85.8 Exit Status

htcondor will exit with a non-zero status value if it fails and zero status if it succeeds.

15.86 procd_ctl

command line interface to the *condor_procd*

15.86.1 Synopsis

procd_ctl -h

procd_ctl -A *address-file* [**command**]

15.86.2 Description

This is a programmatic interface to the *condor_procd* daemon. It may be used to cause the *condor_procd* to do anything that the *condor_procd* is capable of doing, such as tracking and managing process families.

This is a program only available for the Linux ports of HTCondor.

The **-h** option prints out usage information and exits. The *address-file* specification within the **-A** argument specifies the path and file name of the address file which the named pipe clients must use to speak with the *condor_procd*.

One command is given to the *condor_procd*. The choices for the command are defined by the Options.

15.86.3 Options

TRACK_BY_ASSOCIATED_GID *GID* [*PID*]
Use the specified *GID* to track the specified family rooted at *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

GET_USAGE [*PID*]
Get the total usage information about the PID family at *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

DUMP [*PID*]
Print out information about both the root *PID* being watched and the tree of processes under this root *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

LIST [*PID*]
With no *PID* given, print out information about all the watched processes. If the optional *PID* is specified, print out information about the process specified by *PID* and all its child processes.

SIGNAL_PROCESS *signal* [*PID*]
Send the *signal* to the process specified by *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

SUSPEND_FAMILY *PID*
Suspend the process family rooted at *PID*.

CONTINUE_FAMILY *PID*
Continue execution of the process family rooted at *PID*.

KILL_FAMILY *PID*
Kill the process family rooted at *PID*.

UNREGISTER_FAMILY *PID*
Stop tracking the process family rooted at *PID*.

SNAPSHOT
Perform a snapshot of the tracked family tree.

QUIT
Disconnect from the *condor_procd* and exit.

15.86.4 General Remarks

This program may be used in a standalone mode, independent of HTCondor, to track process families. The programs *procd_ctl* and *gidd_alloc* are used with the *condor_procd* in standalone mode to interact with the daemon and inquire about certain state of running processes on the machine, respectively.

15.86.5 Exit Status

procd_ctl will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

CHAPTER

SIXTEEN

CLASSAD ATTRIBUTES

16.1 ClassAd Types

ClassAd attributes vary, depending on the entity producing the ClassAd. Therefore, each ClassAd has an attribute named `MyType`, which describes the type of ClassAd. In addition, the *condor_collector* appends attributes to any daemon's ClassAd, whenever the *condor_collector* is queried. These additional attributes are listed in the unnumbered subsection labeled ClassAd Attributes Added by the *condor_collector* on the *ClassAd Attributes Added by the condor_collector* page.

Here is a list of defined values for `MyType`, as well as a reference to a list attributes relevant to that type.

`Accounting`
: The *condor_negotiator* keeps persistent records for every submitter who has every submitted a job to the pool, containing total usage and priority information. Attributes in the accounting ad are listed and described in *Accounting ClassAd Attributes* The accounting ads for active users can be queried with the *condor_userprio* command, or the accounting ads for all users, including historical ones can be queried with *condor_userprio* -negotiator. Accounting ads hold information about total usage over the user's HTCondor lifetime, but submitter ads hold instantaneous information.

`Collector`
: Each *condor_collector* daemon describes its state. ClassAd attributes that appear in a Collector ClassAd are listed and described in the unnumbered subsection labeled Collector ClassAd Attributes on the *Collector ClassAd Attributes* page. These ads can be shown by running condor_status -collector.

`DaemonMaster`
: Each *condor_master* daemon describes its state. ClassAd attributes that appear in a DaemonMaster ClassAd are listed and described in the unnumbered subsection labeled DaemonMaster ClassAd Attributes on the *DaemonMaster ClassAd Attributes*. These ads can be shown by running condor_status -master.

`Defrag`
: Each *condor_defrag* daemon describes its state. ClassAd attributes that appear in a Defrag ClassAd are listed and described in the unnumbered subsection labeled Defrag ClassAd Attributes on the *Defrag ClassAd Attributes* page. This ad can be shown by running condor_status -defrag.

`Grid`
: The *condor_gridmanager* describes the state of each remote service to which it submits grid universe jobs. ClassAd attributes that appear in a Grid ClassAd are listed and described in the unnumbered subsection labeled

Grid ClassAd Attributes on the *Grid ClassAd Attributes* page. These ad can be shown by running condor_status -grid.

Job
: Each submitted job describes its state, for use by the *condor_negotiator* daemon in finding a machine upon which to run the job. ClassAd attributes that appear in a job ClassAd are listed and described in the unnumbered subsection labeled Job ClassAd Attributes on the *Job ClassAd Attributes* page. These ads can be shown by running condor_q.

Machine
: Each machine in the pool (and hence, the *condor_startd* daemon running on that machine) describes its state. ClassAd attributes that appear in a machine ClassAd are listed and described in the unnumbered subsection labeled Machine ClassAd Attributes on the *Machine ClassAd Attributes* page. These ads can be shown by running condor_status.

Negotiator
: Each *condor_negotiator* daemon describes its state. ClassAd attributes that appear in a Negotiator ClassAd are listed and described in the unnumbered subsection labeled Negotiator ClassAd Attributes on the *Negotiator ClassAd Attributes* page. This ad can be shown by running condor_status -negotiator.

Scheduler
: Each *condor_schedd* daemon describes its state. ClassAd attributes that appear in a Scheduler ClassAd are listed and described in the unnumbered subsection labeled Scheduler ClassAd Attributes on the *Scheduler ClassAd Attributes* page. These ads can be shown by running condor_status -scheduler.

Submitter
: Each submitter is described by a ClassAd. ClassAd attributes that appear in a Submitter ClassAd are listed and described in the unnumbered subsection labeled Submitter ClassAd Attributes on the *Submitter ClassAd Attributes* page. These ads can be shown run running condor_status -submitter.

In addition, statistics are published for each DaemonCore daemon. These attributes are listed and described in the unnumbered subsection labeled DaemonCore Statistics Attributes on the :doc:/classad-attributes/daemon-core-statistics-attributes` page.

16.2 Accounting ClassAd Attributes

The *condor_negotiator* keeps information about each submitter and group in accounting ads that are also sent to the *condor_collector*. Th *condor_userprio* command queries and displays these ads. For example, to see the full set of raw accounting ads, run the command:

```
$ condor_userprio -l
```

¶
: If this record is for an accounting group with quota, the name of the group.

¶
: The total number of seconds this submitter has used since they first arrived in the pool. Note this is not weighted by cpu cores – an eight core job running for one hour has a usage of 3600, compare with WeightedAccumulatedUsage

¶
The Unix epoch time in seconds when this user claimed resources in the system. This is persistent and survives reboots and HTCondor upgrades.

¶
If this record is for an accounting group with quota, the amount of quota statically configured.

¶
A boolean which is true if this record represents an accounting group

¶
The unix epoch time, in seconds, when this submitter last had claimed resources.

¶
The fully qualified name of the user or accounting group. It will be of the form `name@submit.domain`.

¶
The current effective priority of this user.

¶
The priority factor of this user.

¶
The current number of slots claimed.

¶
When the negotiator computes the fair share of the pool that each user should get, assuming they have infinite jobs and every job matches every slot, the SubmitterShare is the fraction of the pool this user should get. A floating point number from 0 to 1.0.

¶
When the negotiator computes the fair share of the pool that each user should get, assuming they have infinite jobs and every job matches every slot, the SubmitterLimit is the absolute number of cores this user should get.

¶
The total amount of core-seconds used by this user since they arrived in the system, assuming `SLOT_WEIGHT = CPUS`

¶
A total number of requested cores across all running jobs from the submitter.

16.3 Job ClassAd Attributes

Both active HTCondor jobs (those in a *condor_schedd*) and historical jobs (those in the history file), are described by classads. Active jobs can be queried and displayed with the *condor_q* command, and historical jobs are queried with the *condor_history* command, as in the examples below. Note that not all job attributes are described here, some are for internal HTCondor use, and are subject to change. Also, not all jobs contain all attributes.

```
$ condor_history -l username
$ condor_q -l
```

¶
Boolean set to true `True` if the ad is absent.

¶
The accounting group name, as set in the submit description file via the **accounting_group** command. This attribute is only present if an accounting group was requested by the submission. See the *User Priorities and Negotiation* section for more information about accounting groups.

¶

The user name associated with the accounting group. This attribute is only present if an accounting group was requested by the submission.

¶

Formally, the length of time in seconds from when the shadow sends a claim activation to when the shadow receives a claim deactivation.

Informally, this is how much time HTCondor's fair-share mechanism will charge the job for, plus one round-trip over the network.

This attribute may not be used in startd policy expressions and is not computed until complete.

¶

Formally, the length of time in seconds from when the shadow received notification that the job had been spawned to when the shadow received notification that the spawned process has exited.

Informally, this is the duration limited by `AllowedExecuteDuration`.

This attribute may not be used in startd policy expressions and is not computed until complete.

¶

Formally, the length of time in seconds from when the shadow sends a claim activation to when the shadow it notified that the job was spawned.

Informally, this is how long it took the starter to prepare to execute the job. That includes file transfer, so the difference between this duration and the duration of input file transfer is (roughly) the execute-side overhead of preparing to start the job.

This attribute may not be used in startd policy expressions and is not computed until complete.

¶

Formally, the length of time in seconds from when the shadow received notification that the spawned process exited to when the shadow received a claim deactivation.

Informally, this is how long it took the starter to finish up after the job. That includes file transfer, so the difference between this duration and the duration of output file transfer is (roughly) the execute-side overhead of handling job termination.

This attribute may not be used in startd policy expressions and is not computed until complete.

¶

The longest time for which a job may be executing. Jobs which exceed this duration will go on hold. This time does not include file-transfer time. Jobs which self-checkpoint have this long to write out each checkpoint.

This attribute is intended to help minimize the time wasted by jobs which may erroneously run forever.

¶

The longest time for which a job may continuously be in the running state. Jobs which exceed this duration will go on hold. Exiting the running state resets the job duration measured by this attribute.

This attribute is intended to help minimize the time wasted by jobs which may erroneously run forever.

¶

String containing a comma-separated list of all the remote machines running a parallel or mpi universe job.

¶

A string representing the command line arguments passed to the job, when those arguments are specified using the old syntax, as specified in the *condor_submit* section.

¶

A string representing the command line arguments passed to the job, when those arguments are specified using the new syntax, as specified in the *condor_submit* section.

¶
A string recording the subject in the authentication token (IDTOKENS or SCITOKENS) used to submit the job.

¶
A string recording the issuer in the authentication token (IDTOKENS or SCITOKENS) used to submit the job.

¶
A string recording the groups in the authentication token (IDTOKENS or SCITOKENS) used to submit the job.

¶
A string recording the scopes in the authentication token (IDTOKENS or SCITOKENS) used to submit the job.

¶
A string recording the unique identifier of the authentication token (IDTOKENS or SCITOKENS) used to submit the job.

¶
For **batch** grid universe jobs, additional command-line arguments to be given to the target batch system's job submission command.

¶
For **batch** grid universe jobs, the name of the project/account/allocation that should be charged for the job's resource usage.

¶
For **batch** grid universe jobs, the name of the queue in the remote batch system.

¶
For **batch** grid universe jobs, a limit in seconds on the job's execution time, enforced by the remote batch system.

¶
The integer number of KiB read from disk for this job.

¶
The integer number of disk blocks read for this job.

¶
The integer number of KiB written to disk for this job.

¶
The integer number of blocks written to disk for this job.

¶
Used for grid type gce jobs; a string taken from the definition of the submit description file command **cloud_label_names** . Defines the set of labels associated with the GCE instance.

¶
Integer cluster identifier for this job. A cluster is a group of jobs that were submitted together. Each job has its own unique job identifier within the cluster, but shares a common cluster identifier. The value changes each time a job or set of jobs are queued for execution under HTCondor.

¶
The path to and the file name of the job to be executed.

¶
The number of seconds of wall clock time that the job has been allocated a machine, excluding the time spent on run attempts that were evicted. Like `RemoteWallClockTime`, this includes time the job spent in a suspended state, so the total committed wall time spent running is

```
CommittedTime - CommittedSuspensionTime
```

¶

This attribute is identical to CommittedTime except that the time is multiplied by the SlotWeight of the machine(s) that ran the job. This relies on SlotWeight being listed in SYSTEM_JOB_MACHINE_ATTRS

¶

A running total of the number of seconds the job has spent in suspension during time in which the job was not evicted. This number is updated when the job exits.

¶

The time when the job completed, or undefined if the job has not yet completed. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970). Note that older versions of HTCondor initialzed CompletionDate to the integer 0, so job ads from older versions of HTCondor might have a 0 CompletionDate for jobs which haven't completed.

¶

A string list, delimited by commas and space characters. The items in the list identify named resources that the job requires. The value can be a ClassAd expression which, when evaluated in the context of the job ClassAd and a matching machine ClassAd, results in a string list.

¶

A string that describes the operating system version that the *condor_submit* command that submitted this job was built for. Note this may be different that the operating system that is actually running.

¶

A string that describes the HTCondor version of the *condor_submit* command that created this job. Note this may be different than the version of the HTCondor daemon that runs the job.

¶

For Container universe jobs, the string that names the container image to be run the job in.

¶

For Container universe jobs, a filename that becomes the working directory of the job. Mapped to the scratch directory.

¶

This attribute is identical to RemoteWallClockTime except that the time is multiplied by the SlotWeight of the machine(s) that ran the job. This relies on SlotWeight being listed in SYSTEM_JOB_MACHINE_ATTRS

¶

A running total of the number of seconds the job has spent in suspension for the life of the job.

¶

The total time, in seconds, that condor has spent transferring the input and output sandboxes for the life of the job.

¶

The number of hosts in the claimed state, due to this job.

¶

For a DAGMan node job only, the ClusterId job ClassAd attribute of the *condor_dagman* job which is the parent of this node job. For nested DAGs, this attribute holds only the ClusterId of the job's immediate parent.

¶

For a DAGMan node job only, a comma separated list of each *JobName* which is a parent node of this job's node. This attribute is passed through to the job via the *condor_submit* command line, if it does not exceed the line length defined with _POSIX_ARG_MAX. For example, if a node job has two parents with *JobName* s B and C, the *condor_submit* command line will contain

```
-append +DAGParentNodeNames="B,C"
```

¶

For a DAGMan node job only, gives the path to an event log used exclusively by DAGMan to monitor the state of the DAG's jobs. Events are written to this log file in addition to any log file specified in the job's submit description file.

¶

For a DAGMan node job only, a comma-separated list of the event codes that should be written to the log specified by DAGManNodesLog, known as the auxiliary log. All events not specified in the DAGManNodesMask string are not written to the auxiliary event log. The value of this attribute is determined by DAGMan, and it is passed to the job via the *condor_submit* command line. By default, the following events are written to the auxiliary job log:

- Submit, event code is 0
- Execute, event code is 1
- Executable error, event code is 2
- Job evicted, event code is 4
- Job terminated, event code is 5
- Shadow exception, event code is 7
- Job aborted, event code is 9
- Job suspended, event code is 10
- Job unsuspended, event code is 11
- Job held, event code is 12
- Job released, event code is 13
- Post script terminated, event code is 16
- Grid submit, event code is 27

If DAGManNodesLog is not defined, it has no effect. The value of DAGManNodesMask does not affect events recorded in the job event log file referred to by UserLog.

¶

For a DAGMan node job only, the current retry attempt number for the node that this job belongs. This attribute is only included if specified by configuration option.

¶

An integer representing the number of seconds before the jobs DeferralTime to which the job may be matched with a machine.

¶

A Unix Epoch timestamp that represents the exact time HTCondor should attempt to begin executing the job.

¶

An integer representing the number of seconds after the jobs DeferralTime to allow the job to arrive at the execute machine before automatically being evicted due to missing its DeferralTime.

¶

An integer that specifies the maximum number of seconds for which delegated proxies should be valid. The default behavior is determined by the configuration setting DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME which defaults to one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the

credential used to create the proxy. This setting currently only applies to proxies delegated for non-grid jobs and HTCondor-C jobs. This setting has no effect if the configuration setting DELEGATE_JOB_GSI_CREDENTIALS is false, because in that case the job proxy is copied rather than delegated.

Amount of disk space (KiB) in the HTCondor execute directory on the execute machine that this job has used. An initial value may be set at the job's request, placing into the job's submit description file a setting such as

```
# 1 megabyte initial value
+DiskUsage = 1024
```

vm universe jobs will default to an initial value of the disk image size. If not initialized by the job, non-**vm** universe jobs will default to an initial value of the sum of the job's executable and all input files.

For Docker and Container universe jobs, a string that names the docker image to run inside the container.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_access_key_id** . Defines the path and file name of the file containing the EC2 Query API's access key.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_ami_id** . Identifies the machine image of the instance.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_block_device_mapping** . Defines the map from block device names to kernel device names for the instance.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_elastic_ip** . Specifies an Elastic IP address to associate with the instance.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_iam_profile_arn** . Specifies the IAM (instance) profile to associate with this instance.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_iam_profile_name** . Specifies the IAM (instance) profile to associate with this instance.

Used for grid type ec2 jobs; a string set for the job once the instance starts running, as assigned by the EC2 service, that represents the unique ID assigned to the instance by the EC2 service.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_instance_type** . Specifies a service-specific instance type.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_keypair** . Defines the key pair associated with the EC2 instance.

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_parameter_names** . Contains a space or comma separated list of the names of additional parameters to pass when instantiating an instance.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_spot_price** . Defines the maximum amount per hour a job submitter is willing to pay to run this job.

¶

Used for grid type ec2 jobs; identifies the spot request HTCondor made on behalf of this job.

¶

Used for grid type ec2 jobs; reports the reason for the most recent EC2-level state transition. Can be used to determine if a spot request was terminated due to a rise in the spot price.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_tag_names** . Defines the set, and case, of tags associated with the EC2 instance.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_keypair_file** . Defines the path and file name of the file into which to write the SSH key used to access the image, once it is running.

¶

Used for grid type ec2 jobs; a string set for the job once the instance starts running, as assigned by the EC2 service, that represents the host name upon which the instance runs, such that the user can communicate with the running instance.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_secret_access_key** . Defines that path and file name of the file containing the EC2 Query API's secret access key.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_security_groups** . Defines the list of EC2 security groups which should be associated with the job.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_security_ids** . Defines the list of EC2 security group IDs which should be associated with the job.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_user_data** . Defines a block of data that can be accessed by the virtual machine.

¶

Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_user_data_file** . Specifies a path and file name of a file containing data that can be accessed by the virtual machine.

¶

A string containing a comma-separated list of job ClassAd attributes. For each attribute name in the list, its value will be included in the e-mail notification upon job completion.

¶

A boolean value taken from the submit description file command **encrypt_execute_directory** . It specifies if HTCondor should encrypt the remote scratch directory on the machine where the job executes.

¶

An integer containing the epoch time of when the job entered into its current status So for example, if the job is on hold, the ClassAd expression

```
time() - EnteredCurrentStatus
```

will equal the number of seconds that the job has been on hold.

¶

A string representing the environment variables passed to the job, when those arguments are specified using the old syntax, as specified in the *condor_submit* section.

¶

A string representing the environment variables passed to the job, when those arguments are specified using the new syntax, as specified in the *condor_submit* section.

¶

A boolean. If missing or true, HTCondor will erase (truncate) the error and output logs when the job restarts. If this attribute is false, and `when_to_transfer_output` is `ON_EXIT_OR_EVICT`, HTCondor will instead append to those files.

¶

Size of the executable in KiB.

¶

An attribute that is `True` when a user job exits via a signal and `False` otherwise. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitBySignal` is set to `False`.

¶

When a user job exits by means other than a signal, this is the exit return code of the user job. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitCode` is set to 0.

¶

When a user job exits by means of an unhandled signal, this attribute takes on the numeric value of the signal. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitSignal` will be undefined.

¶

The way that HTCondor previously dealt with a job's exit status. This attribute should no longer be used. It is not always accurate in heterogeneous pools, or if the job exited with a signal. Instead, see the attributes: `ExitBySignal`, `ExitCode`, and `ExitSignal`.

¶

Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_auth_file** . Defines the path and file name of the file containing authorization credentials to use the GCE service.

¶

Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_image** . Identifies the machine image of the instance.

¶

Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_json_file** . Specifies the path and file name of a file containing a set of JSON object members that should be added to the instance description submitted to the GCE service.

¶

Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_machine_type** . Specifies the hardware profile that should be used for a GCE instance.

¶

Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_metadata** . Defines a set of name/value pairs that can be accessed by the virtual machine.

¶

Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_metadata_file** . Specifies a path and file name of a file containing a set of name/value pairs that can be accessed by the virtual machine.

¶

Used for grid type gce jobs; a boolean taken from the definition of the submit description file command **gce_preemptible** . Specifies whether the virtual machine instance created in GCE should be preemptible.

¶

A string intended to be a unique job identifier within a pool. It currently contains the *condor_schedd* daemon `Name` attribute, a job identifier composed of attributes `ClusterId` and `ProcId` separated by a period, and the job's submission time in seconds since 1970-01-01 00:00:00 UTC, separated by # characters. The value submit.example.com#152.3#1358363336 is an example. While HTCondor guaratees this string will be globally unique, the contents are subject to change, and users should not parse out components of this string.

¶

A string containing the job's status as reported by the remote job management system.

¶

A string defined by the right hand side of the the submit description file command **grid_resource** . It specifies the target grid type, plus additional parameters specific to the grid type.

¶

Time at which the remote job management system became unavailable. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶

Currently only for scheduler and local universe jobs, a string containing a name of a signal to be sent to the job if the job is put on hold.

¶

A string containing a human-readable message about why a job is on hold. This is the message that will be displayed in response to the command `condor_q -hold`. It can be used to determine if a job should be released or not.

¶

An integer value that represents the reason that a job was put on hold. The below table defines all possible values used by attributes `HoldReasonCode`, `NumHoldsByReason`, and `HoldReasonSubCode`.

Integer HoldReasonCode [NumHoldsByReason Label]	Reason for Hold	HoldReasonSubCode
1 [UserRequest]	The user put the job on hold with *condor_hold*.	
3 [JobPolicy]	The `PERIODIC_HOLD` expression evaluated to `True`. Or, `ON_EXIT_HOLD` was true	User Specified
4 [CorruptedCredential]	The credentials for the job are invalid.	

continues on next page

Table 1 – continued from previous page

Integer HoldReasonCode [NumHoldsByReason Label]	Reason for Hold	HoldReasonSubCode
5 [JobPolicyUndefined]	A job policy expression evaluated to `Undefined`.	
6 [FailedToCreateProcess]	The *condor_starter* failed to start the executable.	The Unix errno number.
7 [UnableToOpenOutput]	The standard output file for the job could not be opened.	The Unix errno number.
8 [UnableToOpenInput]	The standard input file for the job could not be opened.	The Unix errno number.
9 [UnableToOpenOutputStream]	The standard output stream for the job could not be opened.	The Unix errno number.
10 [UnableToOpenInputStream]	The standard input stream for the job could not be opened.	The Unix errno number.
11 [InvalidTransferAck]	An internal HTCondor protocol error was encountered when transferring files.	
12 [TransferOutputError]	An error occurred while transferring job output files or self-checkpoint files.	The Unix errno number, or a plug-in error number; see below.
13 [TransferInputError]	An error occurred while transferring job input files.	The Unix errno number, or a plug-in error number; see below.
14 [IwdError]	The initial working directory of the job cannot be accessed.	The Unix errno number.
15 [SubmittedOnHold]	The user requested the job be submitted on hold.	

continues on next page

Table 1 – continued from previous page

Integer HoldReasonCode [NumHoldsByReason Label]	Reason for Hold	HoldReasonSubCode
16 [SpoolingInput]	Input files are being spooled.	
17 [JobShadowMismatch]	A standard universe job is not compatible with the *condor_shadow* version available on the submitting machine.	
18 [InvalidTransferGoAhead]	An internal HTCondor protocol error was encountered when transferring files.	
19 [HookPrepareJobFailure]	`<Keyword>_HOOK_PREPARE_JOB` was defined but could not be executed or returned failure.	
20 [MissedDeferredExecutionTime]	The job missed its deferred execution time and therefore failed to run.	
21 [StartdHeldJob]	The job was put on hold because `WANT_HOLD` in the machine policy was true.	
22 [UnableToInitUserLog]	Unable to initialize job event log.	
23 [FailedToAccessUserAccount]	Failed to access user account.	
24 [NoCompatibleShadow]	No compatible shadow.	
25 [InvalidCronSettings]	Invalid cron settings.	
26 [SystemPolicy]	`SYSTEM_PERIODIC_HOLD` evaluated to true.	

continues on next page

Table 1 – continued from previous page

Integer HoldReasonCode [NumHoldsByReason Label]	Reason for Hold	HoldReasonSubCode
27 [SystemPolicyUndefined]	The system periodic job policy evaluated to undefined.	
32 [MaxTransferInputSizeExceeded]	The maximum total input file transfer size was exceeded. (See `MAX_TRANSFER_INPUT_MB`	
33 [MaxTransferOutputSizeExceeded]	The maximum total output file transfer size was exceeded. (See `MAX_TRANSFER_OUTPUT_MB`	
34 [JobOutOfResources]	Memory usage exceeds a memory limit.	
35 [InvalidDockerImage]	Specified Docker image was invalid.	
36 [FailedToCheckpoint]	Job failed when sent the checkpoint signal it requested.	
37 [EC2UserError]	User error in the EC2 universe:	
	Public key file not defined.	1
	Private key file not defined.	2
	Grid resource string missing EC2 service URL.	4
	Failed to authenticate.	9
	Can't use existing SSH keypair with the given server's type.	10
	You, or somebody like you, cancelled this request.	20
38 [EC2InternalError]	Internal error in the EC2 universe:	
	Grid resource type not EC2.	3
	Grid resource type not set.	5
	Grid job ID is not for EC2.	7
	Unexpected remote job status.	21

continues on next page

Table 1 – continued from previous page

Integer HoldReasonCode [NumHoldsByReason Label]	Reason for Hold	HoldReasonSubCode
39 [EC2AdminError]	Adminstrator error in the EC2 universe:	
	EC2_GAHP not defined.	6
40 [EC2ConnectionProblem]	Connection problem in the EC2 universe	
	...while creating an SSH keypair.	11
	...while starting an on-demand instance.	12
	...while requesting a spot instance.	17
41 [EC2ServerError]	Server error in the EC2 universe:	
	Abnormal instance termination reason.	13
	Unrecognized instance termination reason.	14
	Resource was down for too long.	22
42 [EC2InstancePotentiallyLost]	Instance potentially lost due to an error in the EC2 universe:	
	Connection error while terminating an instance.	15
	Failed to terminate instance too many times.	16
	Connection error while terminating a spot request.	17
	Failed to terminated a spot request too many times.	18
	Spot instance request purged before instance ID acquired.	19
43 [PreScriptFailed]	Pre script failed.	
44 [PostScriptFailed]	Post script failed.	

continues on next page

Table 1 – continued from previous page

Integer HoldReasonCode [NumHoldsByReason Label]	Reason for Hold	HoldReasonSubCode
45 [SingularityTestFailed]	Test of singularity runtime failed before launching a job	
46 [JobDurationExceeded]	The job's allowed duration was exceeded.	
47 [JobExecuteExceeded]	The job's allowed execution time was exceeded.	
48 [HookShadowPrepareJobFailure]	<Keyword>_HOOK_SHADOW_PREPARE_JOB | | failed when it was executed; | status code indicated job should be | held. |	

Note for hold codes 12 [TransferOutputError] and 13 [TransferInputError]: file transfer may invoke file-transfer plug-ins. If it does, the hold subcodes may additionally be 62 (ETIME), if the file-transfer plug-in timed out; or the exit code of the plug-in shifted left by eight bits, otherwise.

¶

An integer value that represents further information to go along with the `HoldReasonCode`, for some values of `HoldReasonCode`. See `HoldReasonCode` for a table of possible values.

¶

A string that uniquely identifies a set of job hooks, and added to the ClassAd once a job is fetched.

¶

Maximum observed memory image size (i.e. virtual memory) of the job in KiB. The initial value is equal to the size of the executable for non-vm universe jobs, and 0 for vm universe jobs. A vanilla universe job's `ImageSize` is recomputed internally every 15 seconds. How quickly this updated information becomes visible to *condor_q* is controlled by `SHADOW_QUEUE_UPDATE_INTERVAL` and `STARTER_UPDATE_INTERVAL`.

Under Linux, `ProportionalSetSize` is a better indicator of memory usage for jobs with significant sharing of memory between processes, because `ImageSize` is simply the sum of virtual memory sizes across all of the processes in the job, which may count the same memory pages more than once.

¶

I/O wait time of the job recorded by the cgroup controller in seconds.

¶

A boolean expression that controls whether or not HTCondor attempts to flush a access point's NFS cache, in order to refresh an HTCondor job's initial working directory. The value will be `True`, unless a job explicitly adds this attribute, setting it to `False`.

¶

A comma-separated list of attribute names. The named attributes and their values are written in the job event log whenever any event is being written to the log. This is the same as the configuration setting `EVENT_LOG_INFORMATION_ATTRS` (see *Daemon Logging Configuration File Entries*) but it applies to the job event log instead of the system event log.

¶

If a job is given a batch name with the -batch-name option to *condor_submit*, this string valued attribute will contain the batch name.

¶

Time at which the job most recently finished transferring its input sandbox. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970)

¶

Time at which the job most recently finished transferring its output sandbox. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970)

¶

Time at which the job most recently began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶

Time at which the job most recently finished transferring its input sandbox and began executing. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970)

¶

Time at which the job most recently began transferring its input sandbox. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970)

¶

Time at which the job most recently finished executing and began transferring its output sandbox. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970)

¶

A string that may be defined for a job by setting **description** in the submit description file. When set, tools which display the executable such as *condor_q* will instead use this string. For interactive jobs that do not have a submit description file, this string will default to `"Interactive job"`.

¶

Time at which the *condor_shadow* and *condor_starter* become disconnected. Set to `Undefined` when a successful reconnect occurs. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶

The number of seconds set for a job lease, the amount of time that a job may continue running on a remote resource, despite its submitting machine's lack of response. See *Job Leases* for details on job leases.

¶

An integer expression that specifies the time in seconds requested by the job for being allowed to gracefully shut down.

¶

An integer indicating what events should be emailed to the user. The integer values correspond to the user choices for the submit command **notification** .

Value	Notification Value
0	Never
1	Always
2	Complete
3	Error

¶

Integer priority for this job, set by *condor_submit* or *condor_prio*. The default value is 0. The higher the number, the greater (better) the priority.

¶

This attribute is retained for backwards compatibility. It may go away in the future. It is equivalent to `NumShadowStarts` for all universes except **scheduler**. For the **scheduler** universe, this attribute is equivalent to `NumJobStarts`.

¶

Time at which the job first began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970). Due to a long standing bug in the 8.6 series and earlier, the job classad that is internal to the *condor_startd* and *condor_starter* sets this to the time that the job most recently began executing. This bug is scheduled to be fixed in the 8.7 series.

¶

Integer which indicates the current status of the job.

Value	Idle
1	Idle
2	Running
3	Removing
4	Completed
5	Held
6	Transferring Output
7	Suspended

¶

Integer which indicates how a job was submitted to HTCondor. Users can set a custom value for job via Python Bindings API.

Value	Method of Submission
Undefined	Unknown
0	*condor_submit*
1	DAGMan-Direct
2	Python Bindings
3	*htcondor job submit*
4	*htcondor dag submit*
5	*htcondor jobset submit*
100+	Portal/User-set

`JobUniverse`

Integer which indicates the job universe.

Value	Universe
5	vanilla, docker
7	scheduler
8	MPI
9	grid
10	java
11	parallel
12	local
13	vm

¶

An integer value that represents the number of seconds that the *condor_schedd* will continue to keep a claim, in the Claimed Idle state, after the job with this attribute defined completes, and there are no other jobs ready to run from this user. This attribute may improve the performance of linear DAGs, in the case when a dependent job can not be scheduled until its parent has completed. Extending the claim on the machine may permit the dependent job to be scheduled with less delay than with waiting for the *condor_negotiator* to match with a new machine.

¶

The Unix signal number that the job wishes to be sent before being forcibly killed. It is relevant only for jobs running on Unix machines.

¶

This attribute is replaced by the functionality in `JobMaxVacateTime` as of HTCondor version 7.7.3. The number of seconds that the job requests the *condor_starter* wait after sending the signal defined as `KillSig` and before forcibly removing the job. The actual amount of time will be the minimum of this value and the execute machine's configuration variable `KILLING_TIMEOUT`

¶

An integer containing the epoch time when the job was last successfully matched with a resource (gatekeeper) Ad.

¶

If, at any point in the past, this job failed to match with a resource ad, this attribute will contain a string with a human-readable message about why the match failed.

¶

An integer containing the epoch time when HTCondor-G last tried to find a match for the job, but failed to do so.

¶

The name of the *condor_collector* of the pool in which a job ran via flocking in the most recent run attempt. This attribute is not defined if the job did not run via flocking.

¶

Time at which the job last performed a successful suspension. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶

Time at which the job was last evicted from a remote workstation. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶

A boolean expression that defaults to `False`, causing the job to be removed from the queue upon completion. An exception is if the job is submitted using `condor_submit -spool`. For this case, the default expression causes the job to be kept in the queue for 10 days after completion.

¶

Machine attribute of name <X> that is placed into this job ClassAd, as specified by the configuration variable `SYSTEM_JOB_MACHINE_ATTRS`. With the potential for multiple run attempts, <N> represents an integer value providing historical values of this machine attribute for multiple runs. The most recent run will have a value of <N> equal to `0`. The next most recent run will have a value of <N> equal to `1`.

¶

The maximum number of hosts that this job would like to claim. As long as `CurrentHosts` is the same as `MaxHosts`, no more hosts are negotiated for.

¶

Maximum time in seconds to let this job run uninterrupted before kicking it off when it is being preempted. This can only decrease the amount of time from what the corresponding startd expression allows.

¶

This integer expression specifies the maximum allowed total size in Mbytes of the input files that are transferred for a job. This expression does not apply to grid universe or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not set, the system setting `MAX_TRANSFER_INPUT_MB` is used. If the observed size of all input files at submit time is larger than the limit, the job will be immediately placed on hold with a `HoldReasonCode` value of 32. If the job passes this initial test, but the size of the input files increases or the limit decreases so that the limit is violated, the job will be placed on hold at the time when the file transfer is attempted.

¶

This integer expression specifies the maximum allowed total size in Mbytes of the output files that are transferred for a job. This expression does not apply to grid universe or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not set, the system setting `MAX_TRANSFER_OUTPUT_MB` is used. If the total size of the job's output files to be transferred is larger than the limit, the job will be placed on hold with a `HoldReasonCode` value of 33. The output will be transferred up to the point when the limit is hit, so some files may be fully transferred, some partially, and some not at all.

¶

An integer expression in units of Mbytes that represents the peak memory usage for the job. Its purpose is to be compared with the value defined by a job with the **request_memory** submit command, for purposes of policy evaluation.

¶

The minimum number of hosts that must be in the claimed state for this job, before the job may enter the running state.

¶

An integer number of seconds delay time after this job starts until the next job is started. The value is limited by the configuration variable `MAX_NEXT_JOB_START_DELAY`

¶

Boolean value which when `True` indicates that this job is a nice job, raising its user priority value, thus causing it to run on a machine only when no other HTCondor jobs want the machine.

¶

A boolean value only relevant to grid universe jobs, which when `True` tells HTCondor to simply abort (remove) any problematic job, instead of putting the job on hold. It is the equivalent of doing *condor_rm* followed by *condor_rm* **-forcex** any time the job would have otherwise gone on hold. If not explicitly set to `True`, the default value will be `False`.

¶

A string that identifies the NT domain under which a job's owner authenticates on a platform running Windows.

¶

An integer value that will increment every time a job is placed on hold. It may be undefined until the job has been held at least once.

¶

The value of this attribute is a (nested) classad containing a count of how many times a job has been placed on hold grouped by the reason the job went on hold. It may be undefined until the job has been held at least once. Each attribute name in this classad is a NumHoldByReason label; see the table above under the documentation for job attribute `HoldReasonCode` for a table of possible values. Each attribute value is an integer stating how many times the job went on hold for that specific reason. An example:

```
NumHoldsByReason = [ UserRequest = 2; JobPolicy = 110; UnableToOpenInput = 1 ]
```

¶

An integer, initialized to zero, that is incremented by the *condor_shadow* each time the job's executable exits of its own accord, with or without errors, and successfully completes file transfer (if requested). Jobs which have done so normally enter the completed state; this attribute is therefore normally only of use when, for example, `on_exit_remove` or `on_exit_hold` is set.

¶

An integer that is incremented by the *condor_schedd* each time the job is matched with a resource ad by the negotiator.

¶

An integer count of the number of times a job successfully reconnected after being disconnected. This occurs when the *condor_shadow* and *condor_starter* lose contact, for example because of transient network failures or a *condor_shadow* or *condor_schedd* restart. This attribute is only defined for jobs that can reconnected: those in the **vanilla** and **java** universes.

¶

An integer count of the number of times the job started executing.

¶

A count of the number of child processes that this job has.

¶

A count of the number of restarts from a checkpoint attempted by this job during its lifetime. Currently updated only for VM universe jobs.

¶

An integer count of the number of times the *condor_shadow* daemon had a fatal error for a given job.

¶

An integer count of the number of times a *condor_shadow* daemon was started for a given job. This attribute is not defined for **scheduler** universe jobs, since they do not have a *condor_shadow* daemon associated with them. For **local** universe jobs, this attribute is defined, even though the process that manages the job is technically a *condor_starter* rather than a *condor_shadow*. This keeps the management of the local universe and other universes as similar as possible. **Note that this attribute is incremented every time the job is matched, even if the match is rejected by the execute machine; in other words, the value of this attribute may be greater than the number of times the job actually ran.**

¶

An integer that is incremented each time HTCondor-G places a job on hold due to some sort of error condition. This counter is useful, since HTCondor-G will always place a job on hold when it gives up on some error condition. Note that if the user places the job on hold using the *condor_hold* command, this attribute is not incremented.

¶

A string that defines a list of jobs. When the job with this attribute defined is removed, all other jobs defined by the list are also removed. The string is an expression that defines a constraint equivalent to the one implied by the command

```
$ condor_rm -constraint <constraint>
```

This attribute is used for jobs managed with *condor_dagman* to ensure that node jobs of the DAG are removed when the *condor_dagman* job itself is removed. Note that the list of jobs defined by this attribute must not form a cyclic removal of jobs, or the *condor_schedd* will go into an infinite loop when any of the jobs is removed.

¶

A URL, as defined by submit command **output_destination**.

¶

String describing the user who submitted this job.

¶

A string that is only relevant to parallel universe jobs. Without this attribute defined, the default policy applied to parallel universe jobs is to consider the whole job completed when the first node exits, killing processes running on all remaining nodes. If defined to the following strings, HTCondor's behavior changes:

`"WAIT_FOR_ALL"`
: HTCondor will wait until every node in the parallel job has completed to consider the job finished.

¶

Defines the command-line arguments for the post command using the old argument syntax, as specified in *condor_submit*. If both `PostArgs` and `PostArguments` exists, the former is ignored.

¶

Defines the command-line arguments for the post command using the new argument syntax, as specified in *condor_submit*, excepting that double quotes must be escaped with a backslash instead of another double quote. If both `PostArgs` and `PostArguments` exists, the former is ignored.

¶

A job in the vanilla, Docker, Java, or virtual machine universes may specify a command to run after the **Executable** has exited, but before file transfer is started. Unlike a DAGMan POST script command, this command is run on the execute machine; however, it is not run in the same environment as the **Executable** . Instead, its environment is set by `PostEnv` or `PostEnvironment`. Like the DAGMan POST script command, this command is not run in the same universe as the **Executable** ; in particular, this command is not run in a Docker container, nor in a virtual machine, nor in Java. This command is also not run with any of vanilla universe's features active, including (but not limited to): cgroups, PID namespaces, bind mounts, CPU affinity, Singularity, or job wrappers. This command is not automatically transferred with the job, so if you're using file transfer, you must add it to the **transfer_input_files** list.

If the specified command is in the job's execute directory, or any sub-directory, you should not set **vm_no_output_vm** , as that will delete all the files in the job's execute directory before this command has a chance to run. If you don't want any output back from your VM universe job, but you do want to run a post command, do not set **vm_no_output_vm** and instead delete the job's execute directory in your post command.

¶

If `SuccessPostExitCode` or `SuccessPostExitSignal` were set, and the post command has run, this attribute will true if the the post command exited on a signal and false if it did not. It is otherwise unset.

¶

If `SuccessPostExitCode` or `SuccessPostExitSignal` were set, the post command has run, and the post command did not exit on a signal, then this attribute will be set to the exit code. It is otherwise unset.

¶

If `SuccessPostExitCode` or `SuccessPostExitSignal` were set, the post command has run, and the post command exited on a signal, then this attribute will be set to that signal. It is otherwise unset.

¶

Defines the environment for the Postscript using the Old environment syntax. If both `PostEnv` and `PostEnvironment` exist, the former is ignored.

¶

Defines the environment for the Postscript using the New environment syntax. If both `PostEnv` and `PostEnvironment` exist, the former is ignored.

¶

Defines the command-line arguments for the pre command using the old argument syntax, as specified in *condor_submit*. If both `PreArgs` and `PreArguments` exists, the former is ignored.

¶

Defines the command-line arguments for the pre command using the new argument syntax, as specified in *condor_submit*, excepting that double quotes must be escape with a backslash instead of another double quote. If both `PreArgs` and `PreArguments` exists, the former is ignored.

¶

A job in the vanilla, Docker, Java, or virtual machine universes may specify a command to run after file transfer (if any) completes but before the **Executable** is started. Unlike a DAGMan PRE script command, this command is run on the execute machine; however, it is not run in the same environment as the **Executable** . Instead, its environment is set by `PreEnv` or `PreEnvironment`. Like the DAGMan POST script command, this command is not run in the same universe as the **Executable** ; in particular, this command is not run in a Docker container, nor in a virtual machine, nor in Java. This command is also not run with any of vanilla universe's features active, including (but not limited to): cgroups, PID namespaces, bind mounts, CPU affinity, Singularity, or job wrappers. This command is not automatically transferred with the job, so if you're using file transfer, you must add it to the **transfer_input_files** list.

¶

If `SuccessPreExitCode` or `SuccessPreExitSignal` were set, and the pre command has run, this attribute will true if the the pre command exited on a signal and false if it did not. It is otherwise unset.

¶

If `SuccessPreExitCode` or `SuccessPreExitSignal` were set, the pre command has run, and the pre command did not exit on a signal, then this attribute will be set to the exit code. It is otherwise unset.

¶

If `SuccessPreExitCode` or `SuccessPreExitSignal` were set, the pre command has run, and the pre command exited on a signal, then this attribute will be set to that signal. It is otherwise unset.

¶

Defines the environment for the prescript using the Old environment syntax. If both `PreEnv` and `PreEnvironment` exist, the former is ignored.

¶

Defines the environment for the prescript using the New environment syntax. If both `PreEnv` and `PreEnvironment` exist, the former is ignored.

¶

An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher priority. When not explicitly set for a job, 0 is used for comparison purposes. This attribute, when set, is considered first: before `PreJobPrio2`, before `JobPrio`, before `PostJobPrio1`, before `PostJobPrio2`, and before `QDate`.

¶

An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher priority. When not explicitly set for a job, 0 is used for comparison purposes. This attribute, when set, is considered after `PreJobPrio1`, but before `JobPrio`, before `PostJobPrio1`, before `PostJobPrio2`, and before `QDate`.

¶

An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher priority. When not explicitly set for a job, 0 is used for comparison purposes. This attribute, when set, is considered after `PreJobPrio1`, after `PreJobPrio1`, and after `JobPrio`, but before `PostJobPrio2`, and before `QDate`.

¶

An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher pri-

ority. When not explicitly set for a job, 0 is used for comparison purposes. This attribute, when set, is considered after PreJobPrio1, after PreJobPrio1, after JobPrio, and after PostJobPrio1, but before QDate.

¶

When True, the *condor_starter* will not prepend Iwd to Cmd, when Cmd is a relative path name and TransferExecutable is False. The default value is False. This attribute is primarily of interest for users of USER_JOB_WRAPPER for the purpose of allowing an executable's location to be resolved by the user's path in the job wrapper.

¶

When True, entries in the file transfer lists that are relative paths will be transferred to the same relative path on the destination machine (instead of the basename).

¶

Integer process identifier for this job. Within a cluster of many jobs, each job has the same ClusterId, but will have a unique ProcId. Within a cluster, assignment of a ProcId value will start with the value 0. The job (process) identifier described here is unrelated to operating system PIDs.

¶

On Linux execute machines with kernel version more recent than 2.6.27, this is the maximum observed proportional set size (PSS) in KiB, summed across all processes in the job. If the execute machine does not support monitoring of PSS or PSS has not yet been measured, this attribute will be undefined. PSS differs from ImageSize in how memory shared between processes is accounted. The PSS for one process is the sum of that process' memory pages divided by the number of processes sharing each of the pages. ImageSize is the same, except there is no division by the number of processes sharing the pages.

¶

Time at which the job was submitted to the job queue. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶.

The integer number of KiB read from disk for this job over the previous time interval defined by configuration variable STATISTICS_WINDOW_SECONDS.

¶.

The integer number of disk blocks read for this job over the previous time interval defined by configuration variable STATISTICS_WINDOW_SECONDS.

¶.

The integer number of KiB written to disk for this job over the previous time interval defined by configuration variable STATISTICS_WINDOW_SECONDS.

¶.

The integer number of blocks written to disk for this job over the previous time interval defined by configuration variable STATISTICS_WINDOW_SECONDS.

¶

A string containing a human-readable message about why the job was released from hold.

¶

The path to the directory in which a job is to be executed on a remote machine.

¶

The name of the *condor_collector* of the pool in which a job is running via flocking. This attribute is not defined if the job is not running via flocking.

¶

The total number of seconds of system CPU time (the time spent at system calls) the job used on remote machines. This does not count time spent on run attempts that were evicted.

¶

The total number of seconds of system CPU time the job used on remote machines, summed over all execution attempts.

¶

The total number of seconds of user CPU time the job used on remote machines. This does not count time spent on run attempts that were evicted. A job in the virtual machine universe will only report this attribute if run on a KVM hypervisor.

¶

The total number of seconds of user CPU time the job used on remote machines, summed over all execution attempts.

¶

Cumulative number of seconds the job has been allocated a machine. This also includes time spent in suspension (if any), so the total real time spent running is

```
RemoteWallClockTime - CumulativeSuspensionTime
```

Note that this number does not get reset to zero when a job is forced to migrate from one machine to another. `CommittedTime`, on the other hand, is just like `RemoteWallClockTime` except it does get reset to 0 whenever the job is evicted.

¶

Number of seconds the job was allocated a machine for its most recent completed execution. This attribute is set after the job exits or is evicted. It will be undefined until the first execution attempt completes or is terminated. When a job has been allocated a machine and is still running, the value will be undefined or will be the value from the previous execution attempt rather than the current one.

¶

Currently only for scheduler universe jobs, a string containing a name of a signal to be sent to the job if the job is removed.

¶

The number of CPUs requested for this job. If dynamic *condor_startd* provisioning is enabled, it is the minimum number of CPUs that are needed in the created dynamic slot.

¶

The amount of disk space in KiB requested for this job. If dynamic *condor_startd* provisioning is enabled, it is the minimum amount of disk space needed in the created dynamic slot.

¶

The number of GPUs requested for this job. If dynamic *condor_startd* provisioning is enabled, it is the minimum number of GPUs that are needed in the created dynamic slot.

¶

Constraint on the properites of GPUs requested for this job. If dynamic *condor_startd* provisioning is enabled, This constraint will be tested against the property attributes of the *AvailableGPUs* attribute of the partitionable slot when choosing which GPUs for the dynamic slot.

¶

A full path to the directory that the job requests the *condor_starter* use as an argument to chroot().

¶

The amount of memory space in MiB requested for this job. If dynamic *condor_startd* provisioning is enabled, it is the minimum amount of memory needed in the created dynamic slot. If not set by the job, its definition is specified by configuration variable `JOB_DEFAULT_REQUESTMEMORY`

`Requirements`
: A classad expression evaluated by the *condor_negotiator*, *condor_schedd*, and *condor_startd* in the context of slot ad. If true, this job is eligible to run on that slot. If the job requirements does not mention the (startd) attribute `OPSYS`, the schedd will append a clause to Requirements forcing the job to match the same `OPSYS` as the access point. The schedd appends a simliar clause to match the `ARCH`. The schedd parameter `APPEND_REQUIREMENTS`, will, if set, append that value to every job's requirements expression.

¶
: Maximum observed physical memory in use by the job in KiB while running.

¶
: The path and filename containing a SciToken to use for a Condor-C job.

¶
: Number of files and directories in the jobs' Scratch directory. The value is updated periodically while the job is running.

¶
: This is the current time, in Unix epoch seconds. It is added by the *condor_schedd* to the job ads that it sends in reply to a query (e.g. sent to *condor_q*). Since it it not present in the job ad in the *condor_schedd*, it should not be used in any expressions that will be evaluated by the *condor_schedd*.

¶
: Utilized for Linux jobs only, the number of bytes allocated for stack space for this job. This number of bytes replaces the default allocation of 512 Mbytes.

¶
: An attribute representing a Unix epoch time that is defined for a job that is spooled to a remote site using `condor_submit -spool` or HTCondor-C and causes HTCondor to hold the output in the spool while the job waits in the queue in the `Completed` state. This attribute is defined when retrieval of the output finishes.

¶
: An attribute representing a Unix epoch time that is defined for a job that is spooled to a remote site using `condor_submit -spool` or HTCondor-C and causes HTCondor to hold the output in the spool while the job waits in the queue in the `Completed` state. This attribute is defined when retrieval of the output begins.

¶
: The default value is `False`. If `True`, and `TransferErr` is `True`, then standard error is streamed back to the access point, instead of doing the transfer (as a whole) after the job completes. If `False`, then standard error is transferred back to the submit machine (as a whole) after the job completes. If `TransferErr` is `False`, then this job attribute is ignored.

¶
: The default value is `False`. If `True`, and `TransferOut` is `True`, then job output is streamed back to the access point, instead of doing the transfer (as a whole) after the job completes. If `False`, then job output is transferred back to the access point (as a whole) after the job completes. If `TransferOut` is `False`, then this job attribute is ignored.

¶
: A boolean attribute defined by the *condor_negotiator* when it makes a match. It will be `True` if the resource was claimed via negotiation when the configuration variable `GROUP_AUTOREGROUP` was `True`. It will be `False` otherwise.

¶
: When HTCondor-C submits a job to a remote *condor_schedd*, it sets this attribute in the remote job ad to match the `GlobalJobId` attribute of the original, local job.

¶

The accounting group name defined by the *condor_negotiator* when it makes a match.

¶

The accounting group name under which the resource negotiated when it was claimed, as set by the *condor_negotiator*.

¶

Specifies if the `executable` exits with a signal after a successful self-checkpoint.

¶

Specifies the exit code, if any, with which the `executable` exits after a successful self-checkpoint.

¶

Specifies the signal, if any, by which the `executable` exits after a successful self-checkpoint.

¶

Specifies if a succesful pre command must exit with a signal.

¶

Specifies the code with which the pre command must exit to be considered successful. Pre commands which are not successful cause the job to go on hold with `ExitCode` set to `PreCmdExitCode`. The exit status of a pre command without one of `SuccessPreExitCode` or `SuccessPreExitSignal` defined is ignored.

¶

Specifies the signal on which the pre command must exit be considered successful. Pre commands which are not successful cause the job to go on hold with `ExitSignal` set to `PreCmdExitSignal`. The exit status of a pre command without one of `SuccessPreExitCode` or `SuccessPreExitSignal` defined is ignored.

¶

Specifies if a succesful post command must exit with a signal.

¶

Specifies the code with which the post command must exit to be considered successful. Post commands which are not successful cause the job to go on hold with `ExitCode` set to `PostCmdExitCode`. The exit status of a post command without one of `SuccessPostExitCode` or `SuccessPostExitSignal` defined is ignored.

¶

Specifies the signal on which the post command must exit be considered successful. Post commands which are not successful cause the job to go on hold with `ExitSignal` set to `PostCmdExitSignal`. The exit status of a post command without one of `SuccessPostExitCode` or `SuccessPostExitSignal` defined is ignored.

¶

ToE stands for Ticket of Execution, and is itself a nested classad that describes how a job was terminated by the execute machine. See the *Managing a Job* section for full details.

¶

A count of the number of times this job has been suspended during its lifetime.

¶

A string attribute containing a comma separated list of directories and/or files that should be transferred from the execute machine to the access point's spool when the job successfully checkpoints.

¶

A boolean expresion that controls whether the HTCondor should transfer the container image from the submit node to the worker node.

¶

An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the error output from the job is transferred from the remote machine back to the access point. The name of the file after transfer is the

file referred to by job attribute `Err`. If `False`, no transfer takes place (remote to access point), and the name of the file is the file referred to by job attribute `Err`.

¶

An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job executable is transferred from the access point to the remote machine. The name of the file (on the access point) that is transferred is given by the job attribute `Cmd`. If `False`, no transfer takes place, and the name of the file used (on the remote machine) will be as given in the job attribute `Cmd`.

¶

An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job input is transferred from the access point to the remote machine. The name of the file that is transferred is given by the job attribute `In`. If `False`, then the job's input is taken from a file on the remote machine (pre-staged), and the name of the file is given by the job attribute `In`.

¶

A string attribute containing a comma separated list of directories, files and/or URLs that should be transferred from the access point to the remote machine when input file transfer is enabled.

¶

When the job finished the most recent recent transfer of its input sandbox, measured in seconds from the epoch. (00:00:00 UTC Jan 1, 1970).

¶

If the job's most recent transfer of its input sandbox was queued, this attribute says when, measured in seconds from the epoch (00:00:00 UTC Jan 1, 1970).

¶

: When the job actually started to transfer files, the most recent time it transferred its input sandbox, measured in seconds from the epoch. This will be later than `TransferInQueued` (if set). (00:00:00 UTC Jan 1, 1970).

¶

The total size in Mbytes of input files to be transferred for the job. Files transferred via file transfer plug-ins are not included. This attribute is automatically set by *condor_submit*; jobs submitted via other submission methods, such as SOAP, may not define this attribute.

¶

The value of this classad attribute is a nested classad, whose values contain several attributes about HTCondor-managed file transfer. These refer to the transfer of the sandbox from the AP submit point to the worker node, or the EP.

Each attribute name has a prefix, either "Cedar", for the HTCondor built-in file transfer method, or the prefix of the file transfer plugin method (such as HTTP). For each of these types of file transfer there is an attribute with that prefix whose body is "FilesCount", the number of files transfered by that method during the last transfer, and "FilesCountTotal", the sum of FilesCount over all execution attempts. In addition, for container universe jobs, there is a sub-attribute `` `ContainerDuration` ``, the number of seconds it took to transfer the container image (if transfered), and `` `ContainerDurationTotal` ``, the sum over all execution attempts.

¶

An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the output from the job is transferred from the remote machine back to the access point. The name of the file after transfer is the file referred to by job attribute `Out`. If `False`, no transfer takes place (remote to access point), and the name of the file is the file referred to by job attribute `Out`.

¶

A string attribute containing a comma separated list of files and/or URLs that should be transferred from the remote machine to the access point when output file transfer is enabled.

¶

When the job finished the most recent recent transfer of its output sandbox, measured in seconds from the epoch.

(00:00:00 UTC Jan 1, 1970).

¶

If the job's most recent transfer of its output sandbox was queued, this attribute says when, measured in seconds from the epoch (00:00:00 UTC Jan 1, 1970).

¶

The value of this classad attribute is a nested classad, whose values mirror those for `TransferInputStats`, but for the transfer from the EP worker node back to the AP submit point.

¶

When the job actually started to transfer files, the most recent time it transferred its output sandbox, measured in seconds from the epoch. This will be later than TransferOutQueued (if set). (00:00:00 UTC Jan 1, 1970).

¶

A boolean value that indicates whether the job is currently transferring input files. The value is Undefined if the job is not scheduled to run or has not yet attempted to start transferring input. When this value is True, to see whether the transfer is active or queued, check TransferQueued.

¶

A boolean value that indicates whether the job is currently transferring output files. The value is Undefined if the job is not scheduled to run or has not yet attempted to start transferring output. When this value is True, to see whether the transfer is active or queued, check TransferQueued.

¶

A string value containing a semicolon separated list of file transfer plugins to be supplied by the job. Each entry in this list will be of the form TAG1[,TAG2[,...]]=/path/to/plugin were *TAG* values are URL prefixes like *HTTP*, and /path/to/plugin is the path that the transfer plugin is to be transferred from. The files mentioned in this list will be transferred to the job sandbox before any file transfer plugins are invoked. A transfer plugin supplied in this will way will be used even if the execute node has a file transfer plugin installed that handles that URL prefix.

¶

A string value containing a comma separated list of file transfer plugin URL prefixes that are needed by the job but not supplied via the TransferPlugins attribute. This attribute is intended to provide a convenient way to match against jobs that need a certain transfer plugin.

¶

A boolean value that indicates whether the job is currently waiting to transfer files because of limits placed by MAX_CONCURRENT_DOWNLOADS or MAX_CONCURRENT_UPLOADS.

¶

The full path and file name on the access point of the log file of job events.

¶

A boolean that when true, tells HTCondor to run this job in container universe. Note that container universe jobs are a "topping" above vanilla universe, and the JobUniverse attribute of container jobs will be 5 (vanilla)

¶

A boolean that when true, tells HTCondor to run this job in docker universe. Note that docker universe jobs are a "topping" above vanilla universe, and the JobUniverse attribute of docker jobs will be 5 (vanilla)

¶

A boolean that, when True, specifies that when the executable exits as described by SuccessCheckpointExitCode, SuccessCheckpointExitBySignal, and SuccessCheckpointExitSignal, HTCondor should do (output) file transfer and immediately continue the job in the same sandbox by restarting executable with the same arguments as the first time.

¶

A boolean expression that, when True, specifies that a graceful shutdown of the job should be done when the

job is removed or put on hold.

¶

An integer, extracted from the platform type of the machine upon which this job is submitted, representing a major version number (currently 5 or 6) for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

¶

An integer, extracted from the platform type of the machine upon which this job is submitted, representing a build number for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

¶

An integer, extracted from the platform type of the machine upon which this job is submitted, representing a minor version number (currently 0, 1, or 2) for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

¶

The full path and file name of the file containing the X.509 user proxy.

¶

For a job with an X.509 proxy credential, this is the email address extracted from the proxy.

¶

For a job that defines the submit description file command **x509userproxy** , this is the time at which the indicated X.509 proxy credential will expire, measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶

For a vanilla or grid universe job that defines the submit description file command **x509userproxy** , this is the VOMS Fully Qualified Attribute Name (FQAN) of the primary role of the credential. A credential may have multiple roles defined, but by convention the one listed first is the primary role.

¶

For a vanilla or grid universe job that defines the submit description file command **x509userproxy** , this is a serialized list of the DN and all FQAN. A comma is used as a separator, and any existing commas in the DN or FQAN are replaced with the string `&comma;`. Likewise, any ampersands in the DN or FQAN are replaced with `&`.

¶

For a vanilla or grid universe job that defines the submit description file command **x509userproxy** , this attribute contains the Distinguished Name (DN) of the credential used to submit the job.

¶

For a vanilla or grid universe job that defines the submit description file command **x509userproxy** , this is the name of the VOMS virtual organization (VO) that the user's credential is part of.

The following job ClassAd attributes appear in the job ClassAd only for declared cron jobs. These represent various allotted job start times that will be used to calculate the jobs `DeferralTime`. These attributes can be represented as an integer, a list of integers, a range of integers, a step (intervals of a range), or an * for all allowed values. For more information visit *CronTab Scheduling*.

¶

The minutes in an hour when the cron job is allowed to start running. Represented by the numerical values 0 to 59.

¶

The hours in the day when the cron job is allowed to start running. Represented by the numerical values 0 to 23.

¶

The days of the month when the cron job is allowed to start running. Represented by the numerical values 1 to 31.

¶
The months of the year when the cron job is allowed to start running. Represented by numerical values 1 to 12.

¶
The days of the week when the cron job is allowed to start running. Represented by numerical values 0 to 7. Both 0 and 7 represent Sunday.

The following job ClassAd attributes are relevant only for **vm** universe jobs.

¶
The MAC address of the virtual machine's network interface, in the standard format of six groups of two hexadecimal digits separated by colons. This attribute is currently limited to apply only to Xen virtual machines.

The following job ClassAd attributes appear in the job ClassAd only for the *condor_dagman* job submitted under DAGMan. They represent status information for the DAG.

¶
The value 1 if the DAG is in recovery mode, and The value 0 otherwise.

¶
The number of DAG nodes that have finished successfully. This means that the entire node has finished, not only an actual HTCondor job or jobs.

¶
The number of DAG nodes that have failed. This value includes all retries, if there are any.

¶
The number of DAG nodes for which a POST script is running or has been deferred because of a POST script throttle setting.

¶
The number of DAG nodes for which a PRE script is running or has been deferred because of a PRE script throttle setting.

¶
The number of DAG nodes for which the actual HTCondor job or jobs are queued. The queued jobs may be in any state.

¶
The number of DAG nodes that are ready to run, but which have not yet started running.

¶
The total number of nodes in the DAG, including the FINAL node, if there is a FINAL node.

¶
The number of DAG nodes that are not ready to run. This is a node in which one or more of the parent nodes has not yet finished.

¶
The number of DAG nodes that will never run due to the failure of an ancestor node. Where an ancestor is a node that a another node depends on either directly or indirectly through a chain of PARENT/CHILD relationships.

¶
The overall status of the DAG, with the same values as the macro `$DAG_STATUS` used in DAGMan FINAL nodes.

0	OK
3	the DAG has been aborted by an ABORT-DAG-ON specification

¶
A timestamp for when the DAGMan process last sent an update of internal information to its job ad.

The following job ClassAd attributes appear in the job ClassAd only for the *condor_dagman* job submitted under DAGMan. They represent job process information about the DAG. These values will reset when a DAG is run via rescue and be retained when a DAG is run via recovery mode.

¶
The total number of job processes submitted by all the nodes in the DAG.

¶
The number of job processes currently idle within the DAG.

¶
The number of job processes currently held within the DAG.

¶
The number of job processes currently executing within the DAG.

¶
The total number of job processes within the DAG that have successfully completed.

The following job ClassAd attributes do not appear in the job ClassAd as kept by the *condor_schedd* daemon. They appear in the job ClassAd written to the job's execute directory while the job is running.

¶
The number of Cpus allocated to the job. With statically-allocated slots, it is the number of Cpus allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `RequestCpus`, but may be larger due to the minimum given to a dynamic slot.

¶
CpusUsage (Note the plural *Cpus*) is a floating point value that represents the number of cpu cores fully used over the lifetime of the job. A cpu-bound, single-threaded job will have a CpusUsage of 1.0. A job that is blocked on I/O for half of its life and is cpu bound for the other have will have a CpusUsage of 0.5. A job that uses two cores fully will have a CpusUsage of 2.0. Jobs with unexpectedly low CpusUsage may be showing lowered throughput due to blocking on network or disk.

¶
The amount of disk space in KiB allocated to the job. With statically-allocated slots, it is the amount of disk space allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `RequestDisk`, but may be larger due to the minimum given to a dynamic slot.

¶
The amount of memory in MiB allocated to the job. With statically-allocated slots, it is the amount of memory space allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `RequestMemory`, but may be larger due to the minimum given to a dynamic slot.

¶
The amount of the custom resource identified by <Name> allocated to the job. For jobs using GPUs, <Name> will be `GPUs`. With statically-allocated slots, it is the amount of the resource allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `Request`<Name>, but may be larger due to the minimum given to a dynamic slot.

16.4 Machine ClassAd Attributes

¶
Boolean which indicates if the slot accepted its current job while the machine was draining.

¶
String which describes HTCondor job activity on the machine. Can have one of the following values:

"Idle"
There is no job activity

"Busy"
A job is busy running

"Suspended"
A job is currently suspended

"Vacating"
A job is currently vacating

"Killing"
A job is currently being killed

"Benchmarking"
The startd is running benchmarks

"Retiring"
Waiting for a job to finish or for the maximum retirement time to expire

¶
String with the architecture of the machine. Currently supported architectures have the following string definitions:

"INTEL"
Intel x86 CPU (Pentium, Xeon, etc).

"X86_64"
AMD/Intel 64-bit X86

¶
On X86_64 Linux machines, this advertises the x86_64 microarchitecture, like *x86_64-v2*. See https://en.wikipedia.org/wiki/X86-64#Microarchitecture_levels for details.

¶
The *condor_startd* has the capability to shut down or hibernate a machine when certain configurable criteria are met. However, before the *condor_startd* can shut down a machine, the hardware itself must support hibernation, as must the operating system. When the *condor_startd* initializes, it checks for this support. If the machine has the ability to hibernate, then this boolean ClassAd attribute will be `True`. By default, it is `False`.

¶
The time at which the slot will leave the `Claimed` state. Currently, this only applies to partitionable slots. This is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
The day of the week, where 0 = Sunday, 1 = Monday, . . . , and 6 = Saturday.

¶
The number of minutes passed since midnight.

¶
The load average contributed by HTCondor, either from remote jobs or running benchmarks.

¶
A string containing the HTCondor version number for the *condor_startd* daemon, the release date, and the build identification number.

¶
The number of seconds since activity on the system console keyboard or console mouse has last been detected. The value can be modified with `SLOTS_CONNECTED_TO_CONSOLE` as defined in the *condor_startd Configuration File Macros* section.

¶
The number of CPUs (cores) in this slot. It is 1 for a single CPU slot, 2 for a dual CPU slot, etc. For a partitionable slot, it is the remaining number of CPUs in the partitionable slot.

¶
On Linux machines, the Cpu family, as defined in the /proc/cpuinfo file.

¶
On Linux machines, the Cpu model number, as defined in the /proc/cpuinfo file.

¶
On Linux machines, the size of the L3 cache, in kbytes, as defined in the /proc/cpuinfo file.

¶
A float which represents this machine owner's affinity for running the HTCondor job which it is currently hosting. If not currently hosting an HTCondor job, `CurrentRank` is 0.0. When a machine is claimed, the attribute's value is computed by evaluating the machine's `Rank` expression with respect to the current job's ClassAd.

¶
Set by the value of configuration variable `DETECTED_CORES`

¶
Set by the value of configuration variable `DETECTED_MEMORY`. Specified in MiB.

¶
The amount of disk space on this machine available for the job in KiB (for example, 23000 = 23 MiB). Specifically, this is the amount of disk space available in the directory specified in the HTCondor configuration files by the `EXECUTE` macro, minus any space reserved with the `RESERVED_DISK` macro. For static slots, this value will be the same as machine ClassAd attribute `TotalSlotDisk`. For partitionable slots, this value will be the quantity of disk space remaining in the partitionable slot.

¶
This attribute is `True` when the slot is draining and undefined if not.

¶
This attribute contains a string that is the request id of the draining request that put this slot in a draining state. It is undefined if the slot is not draining.

¶
The .NET framework versions currently installed on this computer. Default format is a comma delimited list. Current definitions:

"1.1"
: for .Net Framework 1.1

"2.0"
: for .Net Framework 2.0

"3.0"
: for .Net Framework 3.0

`"3.5"`
for .Net Framework 3.5

`"4.0Client"`
for .Net Framework 4.0 Client install

`"4.0Full"`
for .Net Framework 4.0 Full install

¶
For SMP machines that allow dynamic partitioning of a slot, this boolean value identifies that this dynamic slot may be partitioned.

¶
Time at which the machine entered the current Activity (see `Activity` entry above). On all platforms (including NT), this is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
The job run time in cpu-seconds that would be lost if graceful draining were initiated at the time this ClassAd was published. This calculation assumes that jobs will run for the full retirement time and then be evicted.

¶
The estimated time at which graceful draining of the machine could complete if it were initiated at the time this ClassAd was published and there are no active claims. This is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). This value is computed with the assumption that the machine policy will not suspend jobs during draining while the machine is waiting for the job to use up its retirement time. If suspension happens, the upper bound on how long draining could take is unlimited. To avoid suspension during draining, the `SUSPEND` and `CONTINUE` expressions could be configured to pay attention to the `Draining` attribute.

¶
The job run time in cpu-seconds that would be lost if quick or fast draining were initiated at the time this ClassAd was published. This calculation assumes that all evicted jobs will not save a checkpoint.

¶
Time at which quick or fast draining of the machine could complete if it were initiated at the time this ClassAd was published and there are no active claims. This is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
A domain name configured by the HTCondor administrator which describes a cluster of machines which all access the same, uniformly-mounted, networked file systems usually via NFS or AFS. This is useful for Vanilla universe jobs which require remote file access.

¶
A boolean value set to `True` if the machine is capable of executing container universe jobs.

¶
A boolean value set to `True` if the machine is capable of executing docker universe jobs.

¶
An integer value containing the number of megabytes of space used by the docker image cache for cached images used by a worker node. Excludes any images that may be in the cache that were not placed there by HTCondor.

¶
A boolean value set to `True` if the machine is capable of executing container universe jobs with a singularity "sandbox" image type

¶
A boolean value set to `True` if the machine is capable of executing container universe jobs with a singularity "SIF" image type

¶
A boolean value set to `True` if the machine is capable of encrypting execute directories.

¶
A boolean value that when `True` identifies that the machine can use the file transfer mechanism.

¶
A string of comma-separated file transfer protocols that the machine can support. The value can be modified with `FILETRANSFER_PLUGINS` as defined in *condor_starter Configuration File Entries*.

¶
A boolean value that when `True` identifies that the jobs on this machine can create user namespaces without root privileges.

¶
A boolean value set to `True` if the machine being advertised supports the SSE 4.1 instructions, and `Undefined` otherwise.

¶
A boolean value set to `True` if the machine being advertised supports the SSE 4.2 instructions, and `Undefined` otherwise.

¶
A boolean value set to `True` if the machine being advertised supports the SSSE 3 instructions, and `Undefined` otherwise.

¶
A boolean value set to `True` if the machine being advertised supports the avx instructions, and `Undefined` otherwise.

¶
A boolean value set to `True` if the machine being advertised supports the avx2 instructions, and `Undefined` otherwise.

¶
A boolean value set to `True` if the machine being advertised support the avx512f (foundational) instructions.

¶
A boolean value set to `True` if the machine being advertised support the avx512dq instructions.

¶
A boolean value set to `True` if the machine being advertised support the avx512dnni instructions.

¶
A boolean value set to `True` if the machine being advertised supports transferring (checkpoint) files (to the submit node) when the job successfully self-checkpoints.

¶
A boolean value set to `True` if the machine being advertised supports running jobs within Singularity containers.

¶
A boolean value set to `True` if the machine has a /usr/sbin/sshd installed. If `False`, *condor_ssh_to_job* is unlikely to function.

¶
If the configuration triggers the detection of virtual machine software, a boolean value reporting the success thereof; otherwise undefined. May also become `False` if HTCondor determines that it can't start a VM (even if the appropriate software is detected).

¶
A boolean value that when `True` identifies that the machine has the capability to be woken into a fully powered and running state by receiving a Wake On LAN (WOL) packet. This ability is a function of the operating system,

the network adapter in the machine (notably, wireless network adapters usually do not have this function), and BIOS settings. When the *condor_startd* initializes, it tries to detect if the operating system and network adapter both support waking from hibernation by receipt of a WOL packet. The default value is `False`.

¶

If the hardware and software have the capacity to be woken into a fully powered and running state by receiving a Wake On LAN (WOL) packet, this feature can still be disabled via the BIOS or software. If BIOS or the operating system have disabled this feature, the *condor_startd* sets this boolean attribute to `False`.

¶

The Average lifetime of all jobs, including transfer time. This is determined by measuring the lifetime of each *condor_starter* that has exited. This attribute will be undefined until the first time a *condor_starter* has exited.

¶

attribute. This is also the the total number times a *condor_starter* has exited.

¶

The Maximum lifetime of all jobs, including transfer time. This is determined by measuring the lifetime of each *condor_starter* s that has exited. This attribute will be undefined until the first time a *condor_starter* has exited.

¶

The Minimum lifetime of all jobs, including transfer time. This is determined by measuring the lifetime of each *condor_starter* that has exited. This attribute will be undefined until the first time a *condor_starter* has exited.

¶

The Average lifetime of all jobs that have exited in the last 20 minutes, including transfer time. This is determined by measuring the lifetime of each *condor_starter* that has exited in the last 20 minutes. This attribute will be undefined if no *condor_starter* has exited in the last 20 minutes.

¶

The total number of jobs used to calulate the `RecentJobBusyTimeAvg` attribute. This is also the the total number times a *condor_starter* has exited in the last 20 minutes.

¶

The Maximum lifetime of all jobs that have exited in the last 20 minutes, including transfer time. This is determined by measuring the lifetime of each *condor_starter* s that has exited in the last 20 minutes. This attribute will be undefined if no *condor_starter* has exited in the last 20 minutes.

¶

The Minimum lifetime of all jobs, including transfer time. This is determined by measuring the lifetime of each *condor_starter* that has exited. This attribute will be undefined if no *condor_starter* has exited in the last 20 minutes.

¶

The Average lifetime time of all jobs, not including time spent transferring files. This attribute will be undefined until the first time a job exits. Jobs that never start (because they fail to transfer input, for instance) will not be included in the average.

¶

attribute. This is also the the total number times a job has exited. Jobs that never start (because input transfer fails, for instance) are not included in the count.

¶

The lifetime of the longest lived job that has exited. This attribute will be undefined until the first time a job exits.

¶

The lifetime of the shortest lived job that has exited. This attribute will be undefined until the first time a job exits.

¶

The Average lifetime time of all jobs, not including time spent transferring files, that have exited in the last 20

minutes. This attribute will be undefined if no job has exited in the last 20 minutes.

¶

The total number of jobs used to calulate the `RecentJobDurationAvg` attribute. This is the total number of jobs that began execution and have exited in the last 20 minutes.

¶

The lifetime of the longest lived job that has exited in the last 20 minutes. This attribute will be undefined if no job has exited in the last 20 minutes.

¶

The lifetime of the shortest lived job that has exited in the last 20 minutes. This attribute will be undefined if no job has exited in the last 20 minutes.

¶

The total number of times a running job has been preempted on this machine.

¶

The total number of times a running job has been preempted on this machine due to the machine's rank of jobs since the *condor_startd* started running.

¶

The total number of jobs which have been started on this machine since the *condor_startd* started running.

¶

The total number of times a running job has been preempted on this machine based on a fair share allocation of the pool since the *condor_startd* started running.

¶

An attribute defined if a vm universe job is running on this slot. Defined by the number of virtualized CPUs in the virtual machine.

¶

The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account. Pseudo-terminals have virtual keyboard activity from telnet and rlogin sessions. Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`. The value can be modified with `SLOTS_CONNECTED_TO_KEYBOARD` as defined in the *condor_startd Configuration File Macros* section.

¶

Relative floating point performance as determined via a Linpack benchmark.

¶

Time when draining of this *condor_startd* was last initiated (e.g. due to *condor_defrag* or *condor_drain*).

¶

Time when draining of this *condor_startd* was last stopped (e.g. by being cancelled).

¶

Time when the HTCondor central manager last received a status update from this machine. Expressed as the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). Note: This attribute is only inserted by the central manager once it receives the ClassAd. It is not present in the *condor_startd* copy of the ClassAd. Therefore, you could not use this attribute in defining *condor_startd* expressions (and you would not want to).

¶

A floating point number representing the current load average.

¶

A string with the machine's fully qualified host name.

¶

An integer expression that specifies the time in seconds the machine will allow the job to gracefully shut down.

¶

The maximum number of seconds that the slot may remain in the *Claimed* state before returning to the *Unclaimed* state. Currently, this only applies to partitionable slots.

¶

When the *condor_startd* wants to kick the job off, a job which has run for less than this number of seconds will not be hard-killed. The *condor_startd* will wait for the job to finish or to exceed this amount of time, whichever comes sooner. If the job vacating policy grants the job X seconds of vacating time, a preempted job will be soft-killed X seconds before the end of its retirement time, so that hard-killing of the job will not happen until the end of the retirement time if the job does not finish shutting down before then. This is an expression evaluated in the context of the job ClassAd, so it may refer to job attributes as well as machine attributes.

¶

The amount of RAM in MiB in this slot. For static slots, this value will be the same as in `TotalSlotMemory`. For a partitionable slot, this value will be the quantity remaining in the partitionable slot.

¶

Relative integer performance as determined via a Dhrystone benchmark.

¶

The number of seconds that this daemon has been running.

¶

The fraction of recent CPU time utilized by this daemon.

¶

The amount of virtual memory consumed by this daemon in KiB.

¶

The current number of sockets registered by this daemon.

¶

The amount of resident memory used by this daemon in KiB.

¶

The number of open (cached) security sessions for this daemon.

¶

The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

¶

String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd. When using CCB, *condor_shared_port*, and/or an additional private network interface, that information will be included here as well.

¶

The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_startd* daemon last sent a ClassAd update to the *condor_collector*.

¶

The ClassAd type; always set to the literal string `"Machine"`.

¶

The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

¶

A string that lists specific instances of a user-defined machine resource, identified by `name`. Each instance is currently unavailable for purposes of match making.

¶

A ClassAd list that specifies which job universes are presently offline, both as strings and as the corresponding job universe number. Could be used the the startd to refuse to start jobs in offline universes:

```
START = OfflineUniverses is undefined || (! member( JobUniverse, OfflineUniverses ))
```

May currently only contain `"VM"` and `13`.

¶

String describing the operating system running on this machine. Currently supported operating systems have the following string definitions:

"LINUX"
for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, LINUX 2.6.x, or LINUX 3.10.0 kernel systems, as well as Scientific Linux, Ubuntu versions 14.04, and Debian 7.0 (wheezy) and 8.0 (jessie)

"OSX"
for Darwin

"FREEBSD7"
for FreeBSD 7

"FREEBSD8"
for FreeBSD 8

"WINDOWS"
for all versions of Windows

¶

A string indicating an operating system and a version number.

For Linux operating systems, it is the value of the `OpSysName` attribute concatenated with the string version of the `OpSysMajorVer` attribute:

"RedHat5"
for RedHat Linux version 5

"RedHat6"
for RedHat Linux version 6

"RedHat7"
for RedHat Linux version 7

"Fedora16"
for Fedora Linux version 16

"Debian6"
for Debian Linux version 6

"Debian7"
for Debian Linux version 7

"Debian8"
for Debian Linux version 8

"Debian9"
for Debian Linux version 9

"Ubuntu14"
for Ubuntu 14.04

"SL5"
for Scientific Linux version 5

"SL6"
for Scientific Linux version 6

"SLFermi5"
for Fermi's Scientific Linux version 5

"SLFermi6"
for Fermi's Scientific Linux version 6

"SLCern5"
for CERN's Scientific Linux version 5

"SLCern6"
for CERN's Scientific Linux version 6

For MacOS operating systems, it is the value of the `OpSysShortName` attribute concatenated with the string version of the `OpSysVer` attribute:

"MacOSX605"
for MacOS version 10.6.5 (Snow Leopard)

"MacOSX703"
for MacOS version 10.7.3 (Lion)

For BSD operating systems, it is the value of the `OpSysName` attribute concatenated with the string version of the `OpSysMajorVer` attribute:

"FREEBSD7"
for FreeBSD version 7

"FREEBSD8"
for FreeBSD version 8

For Windows operating systems, it is the value of the `OpSys` attribute concatenated with the string version of the `OpSysMajorVer` attribute:

"WINDOWS500"
for Windows 2000

"WINDOWS501"
for Windows XP

"WINDOWS502"
for Windows Server 2003

"WINDOWS600"
for Windows Vista

"WINDOWS601"
for Windows 7

If
A string that holds the long-standing values for the `OpSys` attribute. Currently supported operating systems have the following string definitions:

"LINUX"
for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, LINUX 2.6.x, or LINUX 3.10.0 kernel systems, as well as Scientific Linux, Ubuntu versions 14.04, and Debian 7 and 8

"OSX"
: for Darwin

"FREEBSD7"
: for FreeBSD version 7

"FREEBSD8"
: for FreeBSD version 8

"WINDOWS"
: for all versions of Windows

If

A string giving a full description of the operating system. For Linux platforms, this is generally the string taken from `/etc/hosts`, with extra characters stripped off Debian versions.

"Red Hat Enterprise Linux Server release 6.2 (Santiago)"
: for RedHat Linux version 6

"Red Hat Enterprise Linux Server release 7.0 (Maipo)"
: for RedHat Linux version 7.0

"Ubuntu 14.04.1 LTS"
: for Ubuntu 14.04 point release 1

"Debian GNU/Linux 8"
: for Debian 8.0 (jessie)

"Fedora release 16 (Verne)"
: for Fedora Linux version 16

"MacOSX 7.3"
: for MacOS version 10.7.3 (Lion)

"FreeBSD8.2-RELEASE-p3"
: for FreeBSD version 8

"Windows XP SP3"
: for Windows XP

"Windows 7 SP2"
: for Windows 7

If

An integer value representing the major version of the operating system.

5
: for RedHat Linux version 5 and derived platforms such as Scientific Linux

6
: for RedHat Linux version 6 and derived platforms such as Scientific Linux

7
: for RedHat Linux version 7

14
: for Ubuntu 14.04

7
: for Debian 7

8
: for Debian 8

16
for Fedora Linux version 16

6
for MacOS version 10.6.5 (Snow Leopard)

7
for MacOS version 10.7.3 (Lion)

7
for FreeBSD version 7

8
for FreeBSD version 8

501
for Windows XP

600
for Windows Vista

601
for Windows 7

OpSysLongName
A string containing a terse description of the operating system.

"RedHat"
for RedHat Linux version 6 and 7

"Fedora"
for Fedora Linux version 16

"Ubuntu"
for Ubuntu versions 14.04

"Debian"
for Debian versions 7 and 8

"SnowLeopard"
for MacOS version 10.6.5 (Snow Leopard)

"Lion"
for MacOS version 10.7.3 (Lion)

"FREEBSD"
for FreeBSD version 7 or 8

"WindowsXP"
for Windows XP

"WindowsVista"
for Windows Vista

"Windows7"
for Windows 7

"SL"
for Scientific Linux

"SLFermi"
for Fermi's Scientific Linux

"SLCern"
for CERN's Scientific Linux

If
A string containing a short name for the operating system.

"RedHat"
for RedHat Linux version 5, 6 or 7

"Fedora"
for Fedora Linux version 16

"Debian"
for Debian Linux version 6 or 7 or 8

"Ubuntu"
for Ubuntu versions 14.04

"MacOSX"
for MacOS version 10.6.5 (Snow Leopard) or for MacOS version 10.7.3 (Lion)

"FreeBSD"
for FreeBSD version 7 or 8

"XP"
for Windows XP

"Vista"
for Windows Vista

"7"
for Windows 7

"SL"
for Scientific Linux

"SLFermi"
for Fermi's Scientific Linux

"SLCern"
for CERN's Scientific Linux

If
An integer value representing the operating system version number.

700
for RedHat Linux version 7.0

602
for RedHat Linux version 6.2

1600
for Fedora Linux version 16.0

1404
for Ubuntu 14.04

700
for Debian 7.0

800
for Debian 8.0

704
for FreeBSD version 7.4

802
for FreeBSD version 8.2

605
for MacOS version 10.6.5 (Snow Leopard)

703
for MacOS version 10.7.3 (Lion)

500
for Windows 2000

501
for Windows XP

502
for Windows Server 2003

600
for Windows Vista or Windows Server 2008

601
for Windows 7 or Windows Server 2008

¶
For SMP machines, a boolean value identifying that this slot may be partitioned.

¶
The total number of jobs which have been preempted from this machine in the last twenty minutes.

¶
The total number of times a running job has been preempted on this machine due to the machine's rank of jobs in the last twenty minutes.

¶
The total number of jobs which have been started on this machine in the last twenty minutes.

¶
The total number of times a running job has been preempted on this machine based on a fair share allocation of the pool in the last twenty minutes.

¶
A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before HTCondor will allow the job to use this machine.

¶ **when the**
running job can be evicted. `MaxJobRetirementTime` is the expression of how much retirement time the machine offers to new jobs, whereas `RetirementTimeRemaining` is the negotiated amount of time remaining for the current running job. This may be less than the amount offered by the machine's `MaxJobRetirementTime` expression, because the job may ask for less.

¶
A string containing the version of Singularity available, if the machine being advertised supports running jobs within a Singularity container (see `HasSingularity`).

¶
For SMP machines, the integer that identifies the slot. The value will be X for the slot with

```
name="slotX@full.hostname"
```

For non-SMP machines with one slot, the value will be 1.

¶

For SMP machines with partitionable slots, the partitionable slot will have this attribute set to "Partitionable", and all dynamic slots will have this attribute set to "Dynamic".

¶

This specifies the weight of the slot when calculating usage, computing fair shares, and enforcing group quotas. For example, claiming a slot with SlotWeight = 2 is equivalent to claiming two SlotWeight = 1 slots. See the description of SlotWeight in *condor_startd Configuration File Macros*.

¶

String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd. When using CCB, *condor_shared_port*, and/or an additional private network interface, that information will be included here as well.

¶

String which publishes the machine's HTCondor state. Can be:

"Owner"
: The machine owner is using the machine, and it is unavailable to HTCondor.

"Unclaimed"
: The machine is available to run HTCondor jobs, but a good match is either not available or not yet found.

"Matched"
: The HTCondor central manager has found a good match for this resource, but an HTCondor scheduler has not yet claimed it.

"Claimed"
: The machine is claimed by a remote *condor_schedd* and is probably running a job.

"Preempting"
: An HTCondor job is being preempted in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

"Drained"
: This slot is not accepting jobs, because the machine is being drained.

¶

Describes what type of ClassAd to match with. Always set to the string literal "Job", because machine ClassAds always want to be matched with jobs, and vice-versa.

¶

The load average contributed by HTCondor summed across all slots on the machine, either from remote jobs or running benchmarks.

¶

The number of CPUs (cores) that are on the machine. This is in contrast with Cpus, which is the number of CPUs in the slot.

¶

The quantity of disk space in KiB available across the machine (not the slot). For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute TotalSlotDisk.

¶

A floating point number representing the current load average summed across all slots on the machine.

¶
The total job runtime in cpu-seconds that has been lost due to job evictions caused by draining since this *condor_startd* began executing. In this calculation, it is assumed that jobs are evicted without checkpointing.

¶
The total machine-wide time in cpu-seconds that has not been used (i.e. not matched to a job submitter) due to draining since this *condor_startd* began executing.

¶
The quantity of RAM in MiB available across the machine (not the slot). For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute `TotalSlotMemory`.

¶
The number of CPUs (cores) in this slot. For static slots, this value will be the same as in `Cpus`.

¶
The quantity of disk space in KiB given to this slot. For static slots, this value will be the same as machine ClassAd attribute `Disk`. For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute `TotalDisk`.

¶
The quantity of RAM in MiB given to this slot. For static slots, this value will be the same as machine ClassAd attribute `Memory`. For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute `TotalMemory`.

¶
A sum of the static slots, partitionable slots, and dynamic slots on the machine at the current time.

¶
The number of seconds that this machine (slot) has accumulated within the backfill busy state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶
The number of seconds that this machine (slot) has accumulated within the backfill idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶
The number of seconds that this machine (slot) has accumulated within the backfill killing state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶
The number of seconds that this machine (slot) has accumulated within the claimed busy state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶
The number of seconds that this machine (slot) has accumulated within the claimed idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶
The number of seconds that this machine (slot) has accumulated within the claimed retiring state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶
The number of seconds that this machine (slot) has accumulated within the claimed suspended state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶
The number of seconds that this machine (slot) has accumulated within the matched idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶

The number of seconds that this machine (slot) has accumulated within the owner idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶

The number of seconds that this machine (slot) has accumulated within the preempting killing state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶

The number of seconds that this machine (slot) has accumulated within the preempting vacating state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶

The number of seconds that this machine (slot) has accumulated within the unclaimed benchmarking state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶

The number of seconds that this machine (slot) has accumulated within the unclaimed idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

¶

file entries, and therefore all have the same logins.

¶

The amount of currently available virtual memory (swap space) expressed in KiB. On Linux platforms, it is the sum of paging space and physical memory, which more accurately represents the virtual memory size of the machine.

¶

The maximum number of vm universe jobs that can be started on this machine. This maximum is set by the configuration variable `VM_MAX_NUMBER`.

¶

An attribute defined if a vm universe job is running on this slot. Defined by the amount of memory in use by the virtual machine, given in Mbytes.

¶

Gives the amount of memory available for starting additional VM jobs on this machine, given in Mbytes. The maximum value is set by the configuration variable `VM_MEMORY`.

¶

A boolean value indicating whether networking is allowed for virtual machines on this machine.

¶

The type of virtual machine software that can run on this machine. The value is set by the configuration variable `VM_TYPE`

¶

The reason the VM universe went offline (usually because a VM universe job failed to launch).

¶

The time that the VM universe went offline.

¶

An integer, extracted from the platform type, representing a build number for a Windows operating system. This attribute only exists on Windows machines.

¶
An integer, extracted from the platform type, representing a major version number (currently 5 or 6) for a Windows operating system. This attribute only exists on Windows machines.

¶
An integer, extracted from the platform type, representing a minor version number (currently 0, 1, or 2) for a Windows operating system. This attribute only exists on Windows machines.

In addition, there are a few attributes that are automatically inserted into the machine ClassAd whenever a resource is in the Claimed state:

¶
The host name of the machine that has claimed this resource

¶
A boolean attribute which is `True` if this resource was claimed via negotiation when the configuration variable `GROUP_AUTOREGROUP` is `True`. It is `False` otherwise.

¶
The accounting group name corresponding to the submitter that claimed this resource.

¶
The accounting group name under which this resource negotiated when it was claimed. This attribute will frequently be the same as attribute `RemoteGroup`, but it may differ in cases such as when configuration variable `GROUP_AUTOREGROUP` is `True`, in which case it will have the name of the root group, identified as `<none>`.

¶
The name of the user who originally claimed this resource.

¶
The name of the user who is currently using this resource. In general, this will always be the same as the `RemoteOwner`, but in some cases, a resource can be claimed by one entity that hands off the resource to another entity which uses it. In that case, `RemoteUser` would hold the name of the entity currently using the resource, while `RemoteOwner` would hold the name of the entity that claimed the resource.

¶
The name of the *condor_schedd* which claimed this resource.

¶
The name of the user who is preempting the job that is currently running on this resource.

¶
The name of the user who is preempting the job that is currently running on this resource. The relationship between `PreemptingUser` and `PreemptingOwner` is the same as the relationship between `RemoteUser` and `RemoteOwner`.

¶
A float which represents this machine owner's affinity for running the HTCondor job which is waiting for the current job to finish or be preempted. If not currently hosting an HTCondor job, `PreemptingRank` is undefined. When a machine is claimed and there is already a job running, the attribute's value is computed by evaluating the machine's `Rank` expression with respect to the preempting job's ClassAd.

¶
A running total of the amount of time (in seconds) that all jobs (under the same claim) ran (have spent in the Claimed/Busy state).

¶
A running total of the amount of time (in seconds) that all jobs (under the same claim) have been suspended (in the Claimed/Suspended state).

¶
A running total of the amount of time (in seconds) that a single job ran (has spent in the Claimed/Busy state).

¶
A running total of the amount of time (in seconds) that a single job has been suspended (in the Claimed/Suspended state).

There are a few attributes that are only inserted into the machine ClassAd if a job is currently executing. If the resource is claimed but no job are running, none of these attributes will be defined.

¶
The job's identifier (for example, 152.3), as seen from *condor_q* on the submitting machine.

¶
The time stamp in integer seconds of when the job began executing, since the Unix epoch (00:00:00 UTC, Jan 1, 1970). For idle machines, the value is `UNDEFINED`.

¶
If the job has performed a periodic checkpoint, this attribute will be defined and will hold the time stamp of when the last periodic checkpoint was begun. If the job has yet to perform a periodic checkpoint, or cannot checkpoint at all, the `LastPeriodicCheckpoint` attribute will not be defined.

There are a few attributes that are applicable to machines that are offline, that is, hibernating.

¶
The Unix epoch time when this offline ClassAd would have been matched to a job, if the machine were online. In addition, the slot1 ClassAd of a multi-slot machine will have `slot<X>_MachineLastMatchTime` defined, where `<X>` is replaced by the slot id of each of the slots with `MachineLastMatchTime` defined.

¶
A boolean value, that when `True`, indicates this machine is in an offline state in the *condor_collector*. Such ClassAds are stored persistently, such that they will continue to exist after the *condor_collector* restarts.

¶
A boolean expression that specifies when a hibernating machine should be woken up, for example, by *condor_rooster*.

For machines with user-defined or custom resource specifications, including GPUs, the following attributes will be in the ClassAd for each slot. In the name of the attribute, `<name>` is substituted with the configured name given to the resource.

¶
A space separated list that identifies which of these resources are currently assigned to slots.

¶
A space separated list that indicates which of these resources is unavailable for match making.

¶
An integer quantity of the total number of these resources.

For machines with custom resource specifications that include GPUs, the following attributes may be in the ClassAd for each slot, depending on the value of configuration variable `MACHINE_RESOURCE_INVENTORY_GPUs` and what GPUs are detected. In the name of the attribute, `<name>` is substituted with the *prefix string* assigned for the GPU.

¶
For NVIDIA devices, a dynamic attribute representing the temperature in Celsius of the board containing the GPU.

¶
The CUDA-defined capability for the GPU.

- For CUDA or Open CL devices, the integer clocking speed of the GPU in MHz.
- For CUDA or Open CL devices, the integer number of compute units per GPU.
- For CUDA devices, the integer number of cores per compute unit.
- For CUDA or Open CL devices, a string representing the manufacturer's proprietary device name.
- For NVIDIA devices, a dynamic attribute representing the temperature in Celsius of the GPU die.
- For CUDA devices, a string representing the manufacturer's driver version.
- For CUDA or Open CL devices, a boolean value representing whether error correction is enabled.
- For NVIDIA devices, a count of the number of double bit errors detected for this GPU.
- For NVIDIA devices, a count of the number of single bit errors detected for this GPU.
- For NVIDIA devices, a value between 0 and 100 (inclusive), used to represent the level of fan operation as percentage of full fan speed.
- For CUDA or Open CL devices, the quantity of memory in Mbytes in this GPU.
- For Open CL devices, a string representing the manufacturer's version number.
- For CUDA devices, a string representing the manufacturer's version number.

The following attributes are advertised for a machine in which partitionable slot preemption is enabled.

- A ClassAd list containing the values of the `AccountingGroup` attribute for each dynamic slot of the partitionable slot.
- A ClassAd list containing the values of the `Activity` attribute for each dynamic slot of the partitionable slot.
- A ClassAd list containing the values of the `Cpus` attribute for each dynamic slot of the partitionable slot.
- A ClassAd list containing the values of the `CurrentRank` attribute for each dynamic slot of the partitionable slot.
- A ClassAd list containing the values of the `EnteredCurrentState` attribute for each dynamic slot of the partitionable slot.
- A ClassAd list containing the values of the `Memory` attribute for each dynamic slot of the partitionable slot.

¶
A ClassAd list containing the values of the `Name` attribute for each dynamic slot of the partitionable slot.

¶
A ClassAd list containing the values of the `RemoteOwner` attribute for each dynamic slot of the partitionable slot.

¶
A ClassAd list containing the values of the `RemoteUser` attribute for each dynamic slot of the partitionable slot.

¶
A ClassAd list containing the values of the `RetirementTimeRemaining` attribute for each dynamic slot of the partitionable slot.

¶
A ClassAd list containing the values of the `State` attribute for each dynamic slot of the partitionable slot.

¶
A boolean value set to `True` in both the partitionable and dynamic slots, when configuration variable `ADVERTISE_PSLOT_ROLLUP_INFORMATION` is `True`, such that the *condor_negotiator* knows when partitionable slot preemption is possible and can directly preempt a dynamic slot when appropriate.

The single attribute, `CurrentTime`, is defined by the ClassAd environment.

¶
Evaluates to the the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

Common Cloud Attributes

The following attributes are advertised when `use feature:CommonCloudAttributesGoogle` or `use feature: CommonCloudAttributesAWS` is enabled. All values are strings.

¶
Identifies the VM image. (“image” or “AMI ID”)

¶
Identifies the type of resource allocated. (“machine type” or “instance type”)

¶
Identifies the geographic area in which the instance is running.

¶
Identifies a specific (“availability”) zone within the region.

¶
Presently, either `"Google"` or `"AWS"`.

¶
Presently, either `"GCE"` or `"EC2"`.

¶
The instance’s identifier with its provider (on its platform).

¶
`"True"` if the instance, and `"False"` otherwise.

16.5 DaemonMaster ClassAd Attributes

¶
A string containing the HTCondor version number, the release date, and the build identification number.

¶
The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
The time that this daemon was configured, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
A hexidecimal formatted string that holds all set effective linux capabilities bit mask. This hex string can be decoded using `capsh`. Only exists if running on a Linux OS.

¶
A string with the machine's fully qualified host name.

¶
String with the IP and port address of the *condor_master* daemon which is publishing this DaemonMaster ClassAd.

¶
The number of seconds that this daemon has been running.

¶
The fraction of recent CPU time utilized by this daemon.

¶
The amount of virtual memory consumed by this daemon in Kbytes.

¶
The current number of sockets registered by this daemon.

¶
The amount of resident memory used by this daemon in Kbytes.

¶
The number of open (cached) security sessions for this daemon.

¶
The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

¶
String with the IP and port address of the *condor_master* daemon which is publishing this ClassAd.

¶
The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_master* daemon last sent a ClassAd update to the *condor_collector*.

¶
The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

¶
Description is not yet written.

¶
The UID under which the *condor_master* is started.

¶
An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

16.6 Scheduler ClassAd Attributes

¶
A Statistics attribute defining the number of active autoclusters.

¶
The name of the main *condor_collector* which this *condor_schedd* daemon reports to, as copied from `COLLECTOR_HOST`. If a *condor_schedd* flocks to other *condor_collector* daemons, this attribute still represents the "home" *condor_collector*, so this value can be used to discover if a *condor_schedd* is currently flocking.

¶
A string containing the HTCondor version number, the release date, and the build identification number.

¶
A Statistics attribute defining the ratio of the time spent handling messages and events to the elapsed time for the time period defined by `StatsLifetime` of this *condor_schedd*. A value near 0.0 indicates an idle daemon, while a value near 1.0 indicates a daemon running at or above capacity.

¶
The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
The time that this daemon was configured, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
The number of detected machine CPUs/cores.

¶
The amount of detected machine RAM in MBytes.

¶
A comma separated list of *condor_collector* addresses to which *condor_schedd* jobs are actively flocking.

¶
This attribute contains the Unix epoch time when the job_queue.log file which stores the scheduler's database was first created.

¶
A Statistics attribute defining the sum of the all of the time jobs which did not complete successfully have spent running over the lifetime of this *condor_schedd*.

¶
A Statistics attribute defining the sum of the all of the time jobs which did not complete successfully due to *condor_shadow* exceptions have spent running over the lifetime of this *condor_schedd*.

¶.
A Statistics attribute defining the sum of the all of the time jobs have spent running in the time interval defined by attribute StatsLifetime.

¶.
A Statistics attribute defining the sum of all the time jobs have spent waiting to start in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by time spent running, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute JobsRuntimesHistogramBuckets.

¶
A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by image size, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute JobsSizesHistogramBuckets.

¶
A Statistics attribute defining the number of times jobs that have exited with a *condor_shadow* exit code of JOB_CKPTED in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining the number of jobs successfully completed in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining a histogram count of jobs that completed successfully as classified by time spent running, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute JobsRuntimesHistogramBuckets.

¶
A Statistics attribute defining a histogram count of jobs that completed successfully as classified by image size, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute JobsSizesHistogramBuckets.

¶
A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_COREDUMPED in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of DPRINTF_ERROR in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXEC_FAILED in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining the number of times that jobs that exited (successfully or not) in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining the number of times jobs have exited with a *condor_shadow* exit code of JOB_EXITED_AND_CLAIM_CLOSING in the time interval defined by attribute StatsLifetime.

¶
A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of

JOB_EXITED or with an exit code of JOB_EXITED_AND_CLAIM_CLOSING in the time interval defined by attribute StatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXCEPTION or with an unknown status in the time interval defined by attribute StatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_KILLED in the time interval defined by attribute StatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_MISSED_DEFERRAL_TIME in the time interval defined by attribute StatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_NOT_STARTED in the time interval defined by attribute StatsLifetime.

¶

A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* is currently attempting to reconnect to, in order to recover a job that was running when the *condor_schedd* was restarted.

¶

A Statistics attribute defining a histogram count of *condor_startd* daemons that the *condor_schedd* could not reconnect to in order to recover a job that was running when the *condor_schedd* was restarted, as classified by the time the job spent running. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute JobsRuntimesHistogramBuckets.

¶

A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* tried and failed to reconnect to in order to recover a job that was running when the *condor_schedd* was restarted.

¶

A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* attempted to reconnect to, in order to recover a job that was running when the *condor_schedd* was restarted, but the attempt was interrupted, for example, because the job was removed.

¶

A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* could not attempt to reconnect to, in order to recover a job that was running when the *condor_schedd* was restarted, because the job lease had already expired.

¶

A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* has successfully reconnected to, in order to recover a job that was running when the *condor_schedd* was restarted.

¶

A Statistics attribute representing the number of jobs currently running.

¶

A Statistics attribute defining a histogram count of jobs currently running, as classified by elapsed runtime. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute JobsRuntimesHistogramBuckets.

¶

A Statistics attribute defining a histogram count of jobs currently running, as classified by image size. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute JobsSizesHistogramBuckets.

¶

A Statistics attribute defining the predefined bucket boundaries for histogram statistics that classify run times. Defined as

```
JobsRuntimesHistogramBuckets = "30Sec, 1Min, 3Min, 10Min, 30Min, 1Hr, 3Hr,
        6Hr, 12Hr, 1Day, 2Day, 4Day, 8Day, 16Day"
```

¶

A Statistics attribute defining the number of times that jobs have exited because there was not enough memory to start the *condor_shadow* in the time interval defined by attribute `StatsLifetime`.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_HOLD` in the time interval defined by attribute `StatsLifetime`.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_REMOVE` in the time interval defined by attribute `StatsLifetime`.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_REQUEUE` in the time interval defined by attribute `StatsLifetime`.

¶

A Statistics attribute defining the predefined bucket boundaries for histogram statistics that classify image sizes. Defined as

```
JobsSizesHistogramBuckets = "64Kb, 256Kb, 1Mb, 4Mb, 16Mb, 64Mb, 256Mb,
        1Gb, 4Gb, 16Gb, 64Gb, 256Gb"
```

Note that these values imply powers of two in numbers of bytes.

¶.

A Statistics attribute defining the number of jobs started in the time interval defined by attribute `StatsLifetime`.

¶.

A Statistics attribute defining the number of jobs submitted in the time interval defined by attribute `StatsLifetime`.

¶.

A Statistics attribute defining the number of jobs submitted as late materialization jobs that have not yet materialized.

¶

A string with the machine's fully qualified host name.

¶

The same integer value as set by the evaluation of the configuration variable `MAX_JOBS_RUNNING` . See the definition in the *condor_schedd Configuration File Entries* section.

¶

The number of seconds that this daemon has been running.

¶

The fraction of recent CPU time utilized by this daemon.

¶

The amount of virtual memory consumed by this daemon in Kbytes.

¶

The current number of sockets registered by this daemon.

¶
The amount of resident memory used by this daemon in Kbytes.

¶
The number of open (cached) security sessions for this daemon.

¶
The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

¶
String with the IP and port address of the *condor_schedd* daemon which is publishing this ClassAd.

¶
The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

¶
The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

¶
The number times a job requiring a *condor_shadow* daemon could have been started, but was not started because of the values of configuration variables `JOB_START_COUNT` and `JOB_START_DELAY`

¶
The number of machines (*condor_startd* daemons) matched to this *condor_schedd* daemon, which this *condor_schedd* knows about, but has not yet managed to claim.

¶
The integer number of distinct users with jobs in this *condor_schedd* 's queue.

¶
This is the public network address of this daemon.

¶
A Statistics attribute defining the ratio of the time spent handling messages and events to the elapsed time in the previous time interval defined by attribute `RecentStatsLifetime`.

¶
A Statistics attribute defining the sum of the all of the time that jobs which did not complete successfully have spent running in the previous time interval defined by attribute `RecentStatsLifetime`.

¶
A Statistics attribute defining the sum of the all of the time jobs which have exited in the previous time interval defined by attribute `RecentStatsLifetime` spent running.

¶
A Statistics attribute defining the sum of all the time jobs which have exited in the previous time interval defined by attribute `RecentStatsLifetime` had spent waiting to start.

¶
A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by time spent running, in the previous time interval defined by attribute `RecentStatsLifetime`. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute `JobsRuntimesHistogramBuckets`.

¶
A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by im-

age size, in the previous time interval defined by attribute RecentStatsLifetime. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute JobsSizesHistogramBuckets.

¶

A Statistics attribute defining the number of times jobs that have exited with a *condor_shadow* exit code of JOB_CKPTED in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of jobs successfully completed in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining a histogram count of jobs that completed successfully, as classified by time spent running, in the previous time interval defined by attribute RecentStatsLifetime. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute JobsRuntimesHistogramBuckets.

¶

A Statistics attribute defining a histogram count of jobs that completed successfully, as classified by image size, in the previous time interval defined by attribute RecentStatsLifetime. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute JobsSizesHistogramBuckets.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_COREDUMPED in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of DPRINTF_ERROR in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXEC_FAILED in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited normally in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXITED_AND_CLAIM_CLOSING in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXITED or with an exit code of JOB_EXITED_AND_CLAIM_CLOSING in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXCEPTION or with an unknown status in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_KILLED in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of

JOB_MISSED_DEFERRAL_TIME in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_NOT_STARTED in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited because there was not enough memory to start the *condor_shadow* in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_SHOULD_HOLD in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_SHOULD_REMOVE in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_SHOULD_REQUEUE in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of jobs started in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of jobs submitted in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the number of times that *condor_shadow* daemons lost connection to their *condor_starter* daemons and successfully reconnected in the previous time interval defined by attribute RecentStatsLifetime. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

A Statistics attribute defining the number of times *condor_shadow* processes have been recycled for use with a new job in the previous time interval defined by attribute RecentStatsLifetime. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

A Statistics attribute defining the number of *condor_shadow* daemons started in the previous time interval defined by attribute RecentStatsLifetime.

¶

A Statistics attribute defining the time in seconds over which statistics values have been collected for attributes with names that begin with Recent. This value starts at 0, and it may grow to a value as large as the value defined for attribute RecentWindowMax.

¶

A Statistics attribute defining the time that attributes with names that begin with Recent were last updated, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970). This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

A Statistics attribute defining the maximum time in seconds over which attributes with names that begin with

Recent are collected. The value is set by the configuration variable STATISTICS_WINDOW_SECONDS , which defaults to 1200 seconds (20 minutes). This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

String with the IP and port address of the *condor_schedd* daemon which is publishing this Scheduler ClassAd.

¶

A Statistics attribute defining the number of times *condor_shadow* s lost connection to their *condor_starter* s and successfully reconnected in the previous StatsLifetime seconds. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

A Statistics attribute defining the number of times *condor_shadow* processes have been recycled for use with a new job in the previous StatsLifetime seconds. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

A Statistics attribute defining the number of *condor_shadow* daemons currently running that are owned by this *condor_schedd*.

¶

A Statistics attribute defining the maximum number of *condor_shadow* daemons running at one time that were owned by this *condor_schedd* over the lifetime of this *condor_schedd*.

¶

A Statistics attribute defining the number of *condor_shadow* daemons started in the previous time interval defined by attribute StatsLifetime.

¶

The same boolean value as set in the configuration variable START_LOCAL_UNIVERSE . See the definition in the *condor_schedd Configuration File Entries* section.

¶

The same boolean value as set in the configuration variable START_SCHEDULER_UNIVERSE. See the definition in the *condor_schedd Configuration File Entries* section.

¶

A Statistics attribute defining the time that statistics about jobs were last updated, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970). This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

A Statistics attribute defining the time in seconds over which statistics have been collected for attributes with names that do not begin with Recent. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

¶

The total number of jobs from this *condor_schedd* daemon that are currently flocked to other pools.

¶

The total number of jobs from this *condor_schedd* daemon that are currently on hold.

¶

The total number of jobs from this *condor_schedd* daemon that are currently idle, not including local or scheduler universe jobs.

¶

The total number of all jobs (in all states) from this *condor_schedd* daemon.

¶
The total number of **local universe** jobs from this *condor_schedd* daemon that are currently idle.

¶
The total number of **local universe** jobs from this *condor_schedd* daemon that are currently running.

¶
The current number of all running jobs from this *condor_schedd* daemon that have remove requests.

¶
The total number of jobs from this *condor_schedd* daemon that are currently running, not including local or scheduler universe jobs.

¶
The total number of **scheduler universe** jobs from this *condor_schedd* daemon that are currently idle.

¶
The total number of **scheduler universe** jobs from this *condor_schedd* daemon that are currently running.

¶
A ClassAd expression that provides the name of the transfer queue that the *condor_schedd* will be using for job file transfer.

¶
The interval, in seconds, between publication of this *condor_schedd* ClassAd and the previous publication.

¶
An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

¶
Description is not yet written.

¶ **causes the *condor_negotiator***
daemon to send to this *condor_schedd* daemon a full machine ClassAd corresponding to a matched job.

When using file transfer concurrency limits, the following additional I/O usage statistics are published. These includes the sum and rate of bytes transferred as well as time spent reading and writing to files and to the network. These statistics are reported for the sum of all users and may also be reported individually for recently active users by increasing the verbosity level `STATISTICS_TO_PUBLISH = TRANSFER:2`. Each of the per-user statistics is prefixed by a user name in the form `Owner_<username>_FileTransferUploadBytes`. In this case, the attribute represents activity by the specified user. The published user name is actually the file transfer queue name, as defined by configuration variable `TRANSFER_QUEUE_USER_EXPR`. This expression defaults to `Owner_` followed by the name of the job owner. The attributes that are rates have a suffix that specifies the time span of the exponential moving average. By default the time spans that are published are 1m, 5m, 1h, and 1d. This can be changed by configuring configuration variable `TRANSFER_IO_REPORT_TIMESPANS`. These attributes are only reported once a full time span has accumulated.

¶
The exponential moving average of the disk load that exceeds the upper limit set for the disk load throttle. Periods of time in which there is no excess and no waiting transfers do not contribute to the average. This attribute is published only if configuration variable `FILE_TRANSFER_DISK_LOAD_THROTTLE` is defined.

¶
The desired upper limit for the disk load from file transfers, as configured by `FILE_TRANSFER_DISK_LOAD_THROTTLE`. This attribute is published only if configuration variable `FILE_TRANSFER_DISK_LOAD_THROTTLE` is defined.

¶
The current concurrency limit set by the disk load throttle. The limit is applied to the sum of uploads and downloads. This attribute is published only if configuration variable `FILE_TRANSFER_DISK_LOAD_THROTTLE` is defined.

¶

The lower limit for the disk load from file transfers, as configured by FILE_TRANSFER_DISK_LOAD_THROTTLE. This attribute is published only if configuration variable FILE_TRANSFER_DISK_LOAD_THROTTLE is defined.

¶

The exponential moving average of the disk load that falls below the upper limit set for the disk load throttle. Periods of time in which there is no excess and no waiting transfers do not contribute to the average. This attribute is published only if configuration variable FILE_TRANSFER_DISK_LOAD_THROTTLE is defined.

¶

Total number of bytes downloaded as output from jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferDownloadBytes. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Exponential moving average over the specified time span of the rate at which bytes have been downloaded as output from jobs. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS , which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferDownloadBytesPerSecond_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time reading from files to be transferred as input to jobs. One file transfer process spending nearly all of its time reading files will generate a load close to 1.0. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS , which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferFileReadLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Total number of submit-side transfer process seconds spent reading from files to be transferred as input to jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferFileReadSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time writing to files transferred as output from jobs. One file transfer process spending nearly all of its time writing to files will generate a load close to 1.0. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS , which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferFileWriteLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Total number of submit-side transfer process seconds spent writing to files transferred as output from jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name

Owner_<username>_FileTransferFileWriteSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time reading from the network when transferring output from jobs. One file transfer process spending nearly all of its time reading from the network will generate a load close to 1.0. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow reads from the disk on the execute side. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS , which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetReadLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Total number of submit-side transfer process seconds spent reading from the network when transferring output from jobs since this *condor_schedd* was started. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow reads from the disk on the execute side. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetReadSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time writing to the network when transferring input to jobs. One file transfer process spending nearly all of its time writing to the network will generate a load close to 1.0. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow writes to the disk on the execute side. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS , which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetWriteLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Total number of submit-side transfer process seconds spent writing to the network when transferring input to jobs since this *condor_schedd* was started. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow writes to the disk on the execute side. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetWriteSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Total number of bytes uploaded as input to jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferUploadBytes. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Exponential moving average over the specified time span of the rate at which bytes have been uploaded as input to jobs. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS , which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If

STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferUploadBytesPerSecond_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR

¶

Number of megabytes of output files waiting to be downloaded.

¶

Number of megabytes of input files waiting to be uploaded.

¶

Number of jobs waiting to transfer output files.

¶

Number of jobs waiting to transfer input files.

16.7 Negotiator ClassAd Attributes

¶

A string containing the HTCondor version number, the release date, and the build identification number.

¶

The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶

The time that this daemon was configured, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶

The integer number of submitters the *condor_negotiator* attempted to negotiate with in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of slot ClassAds after filtering by NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT. This is the number of slots actually considered for matching. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of seconds that it took to complete the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The time, represented as the number of seconds since the Unix epoch, at which the negotiation cycle ended. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of successful matches that were made in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of matched jobs divided by the duration of this cycle giving jobs per second. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of matched jobs divided by the period of this cycle giving jobs per second. The period is the time elapsed between the end of the previous cycle and the end of this cycle, and so this rate includes the interval between cycles. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of idle jobs considered for matchmaking. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of jobs requests returned from the schedulers for consideration. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of individual schedulers negotiated with during matchmaking. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of seconds elapsed between the end of the previous negotiation cycle and the end of this cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The duration, in seconds, of Phase 1 of the negotiation cycle: the process of getting submitter and machine ClassAds from the *condor_collector*. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The duration, in seconds, of Phase 2 of the negotiation cycle: the process of filtering slots and processing accounting group configuration. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The duration, in seconds, of Phase 3 of the negotiation cycle: sorting submitters by priority. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The duration, in seconds, of Phase 4 of the negotiation cycle: the process of matching slots to jobs in conjunction with the schedulers. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of rejections that occurred in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of iterations performed during the negotiation cycle. Each iteration includes the reallocation of remaining slots to accounting groups, as defined by the implementation of hierarchical group quotas, together with the negotiation for those slots. The maximum number of iterations is limited by the configuration variable `GROUP_QUOTA_MAX_ALLOCATION_ROUNDS`. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

A string containing a space and comma-separated list of the names of all submitters who failed to negotiate in the negotiation cycle. One possible cause of failure is a communication timeout. This list does not include submitters who ran out of time due to `NEGOTIATOR_MAX_TIME_PER_SUBMITTER`. Those are listed separately in

LastNegotiationCycleSubmittersOutOfTime<X>. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

A string containing a space and comma separated list of the names of all submitters who ran out of time due to NEGOTIATOR_MAX_TIME_PER_SUBMITTER in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

A string containing a space and comma separated list of names of submitters who encountered their fair-share slot limit during the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the negotiation cycle started. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The total number of slot ClassAds received by the *condor_negotiator*. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

The number of slot ClassAds left after trimming currently claimed slots (when enabled). The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

¶

A string with the machine's fully qualified host name.

¶

String with the IP and port address of the *condor_negotiator* daemon which is publishing this ClassAd.

¶

The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

¶

The name of this resource; typically the same value as the Machine attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form slot#@full.hostname, for example, slot1@vulture.cs.wisc.edu, which signifies slot number 1 from vulture.cs.wisc.edu.

¶

String with the IP and port address of the *condor_negotiator* daemon which is publishing this Negotiator ClassAd.

¶

Description is not yet written.

¶

An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

16.8 Submitter ClassAd Attributes

¶
A string containing the HTCondor version number, the release date, and the build identification number.

¶
The number of jobs from this submitter that are running in another pool.

¶
The number of jobs from this submitter that are in the hold state.

¶
The number of jobs from this submitter that are now idle. Scheduler and Local universe jobs are not included in this count.

¶
The number of Local universe jobs from this submitter that are now idle.

¶
The number of Local universe jobs from this submitter that are running.

¶
The IP address associated with the *condor_schedd* daemon used by the submitter.

¶
The fully qualified name of the user or accounting group. It will be of the form `name@submit.domain`.

¶
The number of jobs from this submitter that are running now. Scheduler and Local universe jobs are not included in this count.

¶
The IP address associated with the *condor_schedd* daemon used by the submitter. This attribute is obsolete Use MyAddress instead.

¶
The fully qualified host name of the machine that the submitter submitted from. It will be of the form `submit.domain`.

¶
The number of Scheduler universe jobs from this submitter that are now idle.

¶
The number of Scheduler universe jobs from this submitter that are running.

¶
The fully qualified host name of the central manager of the pool used by the submitter, if the job flocked to the local pool. Or, it will be the empty string if submitter submitted from within the local pool.

¶
A total number of requested cores across all Idle jobs from the submitter, weighted by the slot weight. As an example, if `SLOT_WEIGHT = CPUS`, and a job requests two CPUs, the weight of that job is two.

¶
A total number of requested cores across all Running jobs from the submitter.

16.9 Defrag ClassAd Attributes

¶
Fraction of time CPUs in the pool have spent on jobs that were killed during draining of the machine. This is calculated in each polling interval by looking at `TotalMachineDrainingBadput`. Therefore, it treats evictions of jobs that do and do not produce checkpoints the same. When the *condor_startd* restarts, its counters start over from 0, so the average is only over the time since the daemons have been alive.

¶
Fraction of time CPUs in the pool have spent unclaimed by a user during draining of the machine. This is calculated in each polling interval by looking at `TotalMachineDrainingUnclaimedTime`. When the *condor_startd* restarts, its counters start over from 0, so the average is only over the time since the daemons have been alive.

¶
The time that this daemon was started, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
The time that this daemon was configured, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶
A count of the number of fully drained machines which have arrived during the run time of this *condor_defrag* daemon.

¶
Total count of failed attempts to initiate draining during the lifetime of this *condor_defrag* daemon.

¶
Total count of successful attempts to initiate draining during the lifetime of this *condor_defrag* daemon.

¶
A string with the machine's fully qualified host name.

¶
Number of machines that were observed to be draining in the last polling interval.

¶
Largest number of machines that were ever observed to be draining.

¶
The mean time in seconds between arrivals of fully drained machines.

¶
The number of seconds that this daemon has been running.

¶
The fraction of recent CPU time utilized by this daemon.

¶
The amount of virtual memory consumed by this daemon in KiB.

¶
The current number of sockets registered by this daemon.

¶
The amount of resident memory used by this daemon in KiB.

¶
The number of open (cached) security sessions for this daemon.

¶
The time, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

¶
String with the IP and port address of the *condor_defrag* daemon which is publishing this ClassAd.

¶
The time, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_defrag* daemon last sent a ClassAd update to the *condor_collector*.

¶
The name of this daemon; typically the same value as the `Machine` attribute, but could be customized by the site administrator via the configuration variable `DEFRAG_NAME` .

¶
A ClassAd list of ClassAds describing the last ten cancel commands sent by this daemon. Attributes include `when`, as the number of seconds since the Unix epoch; and `who`, the `Name` of the slot being drained.

¶
Count of failed attempts to initiate draining during the past `RecentStatsLifetime` seconds.

¶
Count of successful attempts to initiate draining during the past `RecentStatsLifetime` seconds.

¶
A ClassAd list of ClassAds describing the last ten drain commands sent by this daemon. Attributes include `when`, as the number of seconds since the Unix epoch; `who`, the `Name` of the slot being drained; and `what`, one of the three strings `graceful`, `quick`, or `fast`.

¶
A Statistics attribute defining the time in seconds over which statistics values have been collected for attributes with names that begin with `Recent`.

¶
An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

¶
Number of machines that were observed to be defragmented in the last polling interval.

¶
Largest number of machines that were ever observed to be simultaneously defragmented.

16.10 Grid ClassAd Attributes

¶
A Statistics attribute defining the time it takes for commands issued to the GAHP server to complete.

¶
A Statistics attribute defining the number of commands issued to the GAHP server that haven't completed yet.

¶
A Statistics attribute defining the total number of commands that have been issued to the GAHP server.

¶
A Statistics attribute defining the number of commands the *condor_gridmanager* is refraining from issuing to the GAHP server due to configuration parameter `GRIDMANAGER_MAX_PENDING_REQUESTS`.

¶

A Statistics attribute defining the number of commands issued to the GAHP server that didn't complete within the timeout period set by configuration parameter `GRIDMANAGER_GAHP_RESPONSE_TIMEOUT`.

¶

The process id of the GAHP server used to interact with the grid service.

¶

Time at which the grid service became unavailable. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

¶

A string giving details as to why the grid service is currently considered unavailable.

¶

An integer classifying the type of error that caused the grid service to be considered unavailable.

Value	Failure Type
1	GAHP PING command failed
2	Failed to start GAHP server

¶

The number of idle jobs currently submitted to the grid service by this *condor_gridmanager*.

¶

The maximum number of jobs this *condor_gridmanager* will submit to the grid service at a time. This is controlled by configuration parameter `GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE`.

¶

The number of jobs this *condor_gridmanager* is managing that are intended for the grid service.

¶

The number of jobs this *condor_gridmanager* currently has submitted to the grid resource.

¶

The number of jobs this *condor_gridmanager* has refrained from submitting to the grid resource due to `JobLimit`.

16.11 Collector ClassAd Attributes

¶

Current number of forked child processes handling queries.

¶

Peak number of forked child processes handling queries since collector startup or statistics reset.

¶

Total number of queries aborted since collector startup (or statistics reset) because `COLLECTOR_QUERY_WORKERS_PENDING` exceeded, or exceeded, or client closed TCP socket while request was pending. This statistic is also available as `RecentDroppedQueries` which represents a count of recently dropped queries that occured within a recent time window (default of 20 minutes).

¶

String with the IP and port address of the *condor_collector* daemon which is publishing this ClassAd.

¶

A string containing the HTCondor version number, the release date, and the build identification number.

¶

The current number of active forks of the Collector. The Windows version of the Collector does not fork and will not have this statistic.

¶

An integer value representing the sum of all jobs running under all universes.

¶

An integer value representing the current number of jobs running under the universe which forms the attribute name. For example

```
CurrentJobsRunningVanilla = 567
```

identifies that the *condor_collector* counts 567 vanilla universe jobs currently running. <universe> is one of Unknown, Vanilla, Scheduler, Java, Parallel, VM, or Local. There are other universes, but they are not listed here, as they represent ones that are no longer used in Condor.

¶

The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶

The time that this daemon was configured, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

¶

Number of locate queries the Collector has handled without forking since it started.

¶

Total time spent handling locate queries without forking since the Collector started. This attribute also has minimum, maximum, average and standard deviation statistics with Min, Max, Avg and Std suffixes respectively.

¶

Number of locate queries the Collector has handled by forking since it started. The Windows operating system does not fork and will not have this statistic.

¶

Total time spent forking to handle locate queries since the Collector started. This attribute also has minimum, maximum, average and standard deviation statistics with Min, Max, Avg and Std suffixes respectively. The Windows operating system does not fork and will not have this statistic.

¶

Number of locate queries the Collector recieved since the Collector started that could not be handled immediately because there were already too many forked child processes. The Windows operating system does not fork and will not have this statistic.

¶

Total time spent queueing pending locate queries that could not be immediately handled by forking since the Collector started. This attribute also has minimum, maximum, average and standard deviation statistics with Min, Max, Avg and Std suffixes respectively. The Windows operating system does not fork and will not have this statistic.

¶

Number of queries that are not locate queries the Collector has handled without forking since it started.

¶

Total time spent handling queries that are not locate queries without forking since the Collector started. This

attribute also has minimum, maximum, average and standard deviation statistics with Min, Max, Avg and Std suffixes respectively.

¶

Number of queries that are not locate queries the Collector has handled by forking since it started. The Windows operating system does not fork and will not have this statistic.

¶

Total time spent forking to handle queries that are not locate queries since the Collector started. This attribute also has minimum, maximum, average and standard deviation statistics with Min, Max, Avg and Std suffixes respectively. The Windows operating system does not fork and will not have this statistic.

¶

Number of queries that are not locate queries the Collector recieved since the Collector started that could not be handled immediately because there were already too many forked child processes. The Windows operating system does not fork and will not have this statistic.

¶

Total time spent queueing pending non-locate queries that could not be immediately handled by forking since the Collector started. This attribute also has minimum, maximum, average and standard deviation statistics with Min, Max, Avg and Std suffixes respectively. The Windows operating system does not fork and will not have this statistic.

¶

Description is not yet written.

¶

Description is not yet written.

¶

Description is not yet written.

¶

Description is not yet written.

¶

Description is not yet written.

¶

A string with the machine's fully qualified host name.

¶

An integer value representing the sum of all `MaxJobsRunning<universe>` values.

¶

An integer value representing largest number of currently running jobs ever seen under the universe which forms the attribute name, over the life of this *condor_collector* process. For example

```
MaxJobsRunningVanilla = 401
```

identifies that the *condor_collector* saw 401 vanilla universe jobs currently running at one point in time, and that was the largest number it had encountered. `<universe>` is one of `Unknown`, `Vanilla`, `Scheduler`, `Java`, `Parallel`, `VM`, or `Local`. There are other universes, but they are not listed here, as they represent ones that are no longer used in Condor.

¶

String with the IP and port address of the *condor_collector* daemon which is publishing this ClassAd.

¶

The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

¶

The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

¶

The maximum number of active forks of the Collector at any time since the Collector started. The Windows version of the Collector does not fork and will not have this statistic.

¶

Number of queries pending that are waiting to fork.

¶

Peak number of queries pending that are waiting to fork since collector startup or statistics reset.

¶

Definition not yet written.

¶

The integer number of unique *condor_startd* daemon ClassAds counted at the most recent time the *condor_collector* updated its own ClassAd.

¶

The largest integer number of unique *condor_startd* daemon ClassAds seen at any one time, since the *condor_collector* began executing.

¶

The integer number of unique submitters counted at the most recent time the *condor_collector* updated its own ClassAd.

¶

The largest integer number of unique submitters seen at any one time, since the *condor_collector* began executing.

¶

Description is not yet written.

¶

An integer that begins at 0, and increments by one each time the same ClassAd is again advertised.

¶

A Statistics attribute representing a count of unique ClassAds seen, over the lifetime of this *condor_collector*. Counts per ClassAd are advertised in attributes named by ClassAd type as `UpdatesInitial_<ClassAd-Name>`. `<ClassAd-Name>` is each of `CkptSrvr`, `Collector`, `Defrag`, `Master`, `Schedd`, `Start`, `StartdPvt`, and `Submittor`.

¶

A Statistics attribute representing the count of updates lost, over the lifetime of this *condor_collector*. Counts per ClassAd are advertised in attributes named by ClassAd type as `UpdatesLost_<ClassAd-Name>`. `<ClassAd-Name>` is each of `CkptSrvr`, `Collector`, `Defrag`, `Master`, `Schedd`, `Start`, `StartdPvt`, and `Submittor`.

¶

A Statistics attribute defining the largest number of updates lost at any point in time, over the lifetime of this *condor_collector*. ClassAd sequence numbers are used to detect lost ClassAds.

¶

A Statistics attribute defining the floating point ratio of the total number of updates to the number of updates lost over the lifetime of this *condor_collector*. ClassAd sequence numbers are used to detect lost ClassAds. A value of 1 indicates that all ClassAds have been lost.

¶

A Statistics attribute representing the count of the number of ClassAd updates received over the lifetime of this *condor_collector*. Counts per ClassAd are advertised in attributes named by ClassAd type as `UpdatesTotal_<ClassAd-Name>`. `<ClassAd-Name>` is each of `CkptSrvr`, `Collector`, `Defrag`, `Master`, `Schedd`, `Start`, `StartdPvt`, and `Submittor`.

16.12 ClassAd Attributes Added by the *condor_collector*

¶

The authenticated name assigned by the *condor_collector* to the daemon that published the ClassAd.

¶

The authentication method used by the *condor_collector* to determine the `AuthenticatedIdentity`.

¶

The time inserted into a daemon's ClassAd representing the time that this *condor_collector* last received a message from the daemon. Time is represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970). This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

¶

A bitmap representing the status of the most recent updates received from the daemon. This attribute is only added if is non-zero. See the *condor_collector Configuration File Entries* section for more information on this setting. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

¶

An integer count of the number of updates from the daemon that the *condor_collector* can definitively determine were lost since the *condor_collector* started running. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

¶

An integer count of the number of updates received from the daemon, for which the *condor_collector* can tell how many were or were not lost, since the *condor_collector* started running. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

¶

An integer count started when the *condor_collector* started running, representing the sum of the number of updates actually received from the daemon plus the number of updates that the *condor_collector* determined were lost. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

16.13 DaemonCore Statistics Attributes

Every HTCondor daemon keeps a set of operational statistics, some of which are common to all daemons, others are specific to the running of a particular daemon. In some cases, the statistics can reveal buggy or slow performance of the HTCondor system. The following statistics are available for all daemons, and can be accessed directly with the condor_status command with a direct query, such as

```
$ condor_status -direct somehostname.example.com -schedd -statistics DC:2 -l
```

¶

This attribute is the number of bytes in the incoming UDP receive queue for this daemon, if it has a UDP command

port. This attribute is polled once a minute by default, so may be out of date. The attribute DCUdpQueueDepthPeak records the peak depth since the daemon has started.

¶

This attribute is the count of debugging messages printed to the daemon's debug log, such as the ScheddLog. There is a moderate cost to writing these logging messages, if the debug level is very high for an active daemon, the logging will slow performance. The corresponding attribute RecentDebugOuts is the count of the messages in the last 20 minutes.

¶

This attribute is the number of messages received on a Unix pipe by this daemon since start time. The corresponding attribute RecentPipeMessages is the count of message in the last 20 minutes.

¶

This attribute respresents the total number of wall clock seconds this daemon has spent processing pipe message since start. The corresponding attribute RecentPipeRuntime is the total time in the last 20 minutes.

¶

This attribute represents the total number of wall clock seconds this daemon has spent completely idle, waiting to process incoming requests or internal timers. The attribute DaemonCoreDutyCycle, which may be easier to write policy around, is based off of this.

¶

This attribute respresents the total number of wall clock time seconds this daemon has spent processing signals since start. The corresponding attribute RecentSignalRuntime is the total time in the last 20 minutes.

¶

This attribute is the number of signals, either Unix signals, or HTCondor simulated signals received by this daemon since start time. The corresponding attribute RecentSignals is the number of signals in the last 20 minutes.

¶

This attribute respresents the total number of wall clock time seconds this daemon has spent processing socket messages since start. The corresponding attribute RecentTimerRuntime is the total time in the last 20 minutes.

¶

This attribute is the number of messages received on socket by this daemon since start time. The corresponding attribute RecentSockMessages is the count of message in the last 20 minutes.

¶

This attribute respresents the total number of wall clock time seconds this daemon has spent processing timers since start. The corresponding attribute RecentTimerRuntime is the total time in the last 20 minutes.

¶

This attribute is the number of internal timers which have fired in this daemon during the most recent pass of the event loop. The corresponding attribute TimersFiredPeak is the maximum number of timers fired in one pass of the event loop since daemon start time.

CHAPTER

SEVENTEEN

CODES AND OTHER NEEDED VALUES

17.1 *condor_shadow* Exit Codes

When a *condor_shadow* daemon exits, the *condor_shadow* exit code is recorded in the *condor_schedd* log, and it identifies why the job exited. Prose in the log appears of the form

```
Shadow pid XXXXX for job XX.X exited with status YYY
```

where `YYY` is the exit code, or

```
Shadow pid XXXXX for job XX.X reports job exit reason 100.
```

where the exit code is the value 100. The following table lists these codes:

Value	Error Name	Description
4	JOB_EXCEPTION	the job exited with an exception
44	DPRINTF_ERROR	there was a fatal error with dprintf()
100	JOB_EXITED	the job exited (not killed)
101	JOB_CKPTED	no longer used
102	JOB_KILLED	the job was killed
103	JOB_COREDUMPED	the job was killed and a core file was produced
105	JOB_NO_MEM	not enough memory to start the condor_shadow
106	JOB_SHADOW_USAGE	incorrect arguments to condor_shadow
107	JOB_NOT_CKPTED	no longer used
107	JOB_SHOULD_REQUEUE	same number as JOB_NOT_CKPTED, to achieve the same behavior. This exit code implies that we want the job to be put back in the job queue and run again.
108	JOB_NOT_STARTED	can not connect to the *condor_startd* or request refused
109	JOB_BAD_STATUS	job status != RUNNING on start up
110	JOB_EXEC_FAILED	exec failed for some reason other than ENOMEM
111	JOB_NO_CKPT_FILE	no longer used
112	JOB_SHOULD_HOLD	the job should be put on hold
113	JOB_SHOULD_REMOVE	the job should be removed
114	JOB_MISSED_DEFERRAL_TIME	the job goes on hold, because it did not run within the specified window of time
115	JOB_EXITED_AND_CLAIM_CLOSING	the job exited (not killed) but the *condor_startd* is not accepting any more jobs on this claim
116	JOB_RECONNECT_FAILED	the *condor_shadow* was started in reconnect mode, and yet failed to reconnect to the starter

17.2 Job Event Log Codes

Table *B.2* lists codes that appear as the first

These are all of the events that can show up in a job log file:

Event Number: 000
Event Name: Job submitted
Event Description: This event occurs when a user submits a job. It is the first event you will see for a job, and it should only occur once.

Event Number: 001
Event Name: Job executing
Event Description: This shows up when a job is running. It might occur more than once.

Event Number: 002
Event Name: Error in executable
Event Description: The job could not be run because the executable was bad.

Event Number: 003
Event Name: Job was checkpointed
Event Description: No longer used.

Event Number: 004
Event Name: Job evicted from machine
Event Description: A job was removed from a machine before it finished, usually for a policy reason. Perhaps an interactive user has claimed the computer, or perhaps another job is higher priority.

Event Number: 005
Event Name: Job terminated
Event Description: The job has completed.

Event Number: 006
Event Name: Image size of job updated
Event Description: An informational event, to update the amount of memory that the job is using while running. It does not reflect the state of the job.

Event Number: 007
Event Name: Shadow exception
Event Description: The *condor_shadow*, a program on the submit computer that watches over the job and performs some services for the job, failed for some catastrophic reason. The job will leave the machine and go back into the queue.

Event Number: 008
Event Name: Generic log event
Event Description: Not used.

Event Number: 009
Event Name: Job aborted
Event Description: The user canceled the job.

Event Number: 010
Event Name: Job was suspended
Event Description: The job is still on the computer, but it is no longer executing. This is usually for a policy reason, such as an interactive user using the computer.

Event Number: 011
Event Name: Job was unsuspended
Event Description: The job has resumed execution, after being suspended earlier.

Event Number: 012
Event Name: Job was held
Event Description: The job has transitioned to the hold state. This might happen if the user applies the *condor_hold* command to the job.

Event Number: 013
Event Name: Job was released
Event Description: The job was in the hold state and is to be re-run.

Event Number: 014
Event Name: Parallel node executed
Event Description: A parallel universe program is running on a node.

Event Number: 015
Event Name: Parallel node terminated
Event Description: A parallel universe program has completed on a node.

Event Number: 016
Event Name: POST script terminated
Event Description: A node in a DAGMan work flow has a script that should be run after a job. The script is run on the submit host. This event signals that the post script has completed.

Event Number: 021
Event Name: Remote error
Event Description: The *condor_starter* (which monitors the job on the execution machine) has failed.

Event Number: 022
Event Name: Remote system call socket lost
Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) have lost contact.

Event Number: 023
Event Name: Remote system call socket reestablished
Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) have been able to resume contact before the job lease expired.

Event Number: 024
Event Name: Remote system call reconnect failure
Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) were unable to resume contact before the job lease expired.

Event Number: 025
Event Name: Grid Resource Back Up
Event Description: A grid resource that was previously unavailable is now available.

Event Number: 026
Event Name: Detected Down Grid Resource
Event Description: The grid resource that a job is to run on is unavailable.

Event Number: 027
Event Name: Job submitted to grid resource
Event Description: A job has been submitted, and is under the auspices of the grid resource.

Event Number: 028
Event Name: Job ad information event triggered.
Event Description: Extra job ClassAd attributes are noted. This event is written as a supplement to other events when the configuration parameter `EVENT_LOG_JOB_AD_INFORMATION_ATTRS` is set.

Event Number: 029
Event Name: The job's remote status is unknown
Event Description: No updates of the job's remote status have been received for 15 minutes.

Event Number: 030
Event Name: The job's remote status is known again
Event Description: An update has been received for a job whose remote status was previous logged as unknown.

Event Number: 031
Event Name: Job stage in

Event Description: A grid universe job is doing the stage in of input files.

Event Number: 032
Event Name: Job stage out
Event Description: A grid universe job is doing the stage out of output files.

Event Number: 033
Event Name: Job ClassAd attribute update
Event Description: A Job ClassAd attribute is changed due to action by the *condor_schedd* daemon. This includes changes by *condor_prio*.

Event Number: 034
Event Name: Pre Skip event
Event Description: For DAGMan, this event is logged if a PRE SCRIPT exits with the defined PRE_SKIP value in the DAG input file. This makes it possible for DAGMan to do recovery in a workflow that has such an event, as it would otherwise not have any event for the DAGMan node to which the script belongs, and in recovery, DAGMan's internal tables would become corrupted.

Event Number: 035
Event Name: Cluster Submit
Event Description: This event occurs when a user submits a cluster with multiple procs.

Event Number: 036
Event Name: Cluster Remove
Event Description: This event occurs after all the jobs in a multi-proc cluster have completed, or when the cluster is removed (by *condor_rm*).

Event Number: 037
Event Name: Factory Paused
Event Description: This event occurs when job materialization for a cluster has been paused.

Event Number: 038
Event Name: Factory Resumed
Event Description: This event occurs when job materialization for a cluster has been resumed

Event Number: 039
Event Name: None
Event Description: This event should never occur in a log but may be returned by log reading code in certain situations (e.g., timing out while waiting for a new event to appear in the log).

Event Number: 040
Event Name: File Transfer

Event Description: This event occurs when a file transfer event occurs: transfer queued, transfer started, or transfer finished, for both the input and output sandboxes.

Table B.2: Event Codes in a Job Event Log

001	EXECUTE	Execute
002	EXECUTABLE_ERROR	Executable error
003	CHECKPOINTED	no longer used
004	JOB_EVICTED	Job evicted
005	JOB_TERMINATED	Job terminated
006	IMAGE_SIZE	Image size
007	SHADOW_EXCEPTION	Shadow exception
009	JOB_ABORTED	Job aborted
010	JOB_SUSPENDED	Job suspended
011	JOB_UNSUSPENDED	Job unsuspended
012	JOB_HELD	Job held
013	JOB_RELEASED	Job released
014	NODE_EXECUTE	Node execute
015	NODE_TERMINATED	Node terminated
016	POST_SCRIPT_TERMINATED	Post script terminated
021	REMOTE_ERROR	Remote error
022	JOB_DISCONNECTED	Job disconnected
023	JOB_RECONNECTED	Job reconnected
024	JOB_RECONNECT_FAILED	Job reconnect failed
025	GRID_RESOURCE_UP	Grid resource up
026	GRID_RESOURCE_DOWN	Grid resource down
027	GRID_SUBMIT	Grid submit
028	JOB_AD_INFORMATION	Job ClassAd attribute values added to event log
029	JOB_STATUS_UNKNOWN	Job status unknown
030	JOB_STATUS_KNOWN	Job status known
031	JOB_STAGE_IN	Grid job stage in
032	JOB_STAGE_OUT	Grid job stage out
033	ATTRIBUTE_UPDATE	Job ClassAd attribute update
034	PRESKIP	DAGMan PRE_SKIP defined
035	CLUSTER_SUBMIT	Cluster submitted
036	CLUSTER_REMOVE	Cluster removed
037	FACTORY_PAUSED	Factory paused
038	FACTORY_RESUMED	Factory resumed
039	NONE	No event could be returned
040	FILE_TRANSFER	File transfer

17.3 Job Universe Numbers

Table B.3: Job Universe Numbers (job attribute JobUniverse)

Number	Job Universe
1	Standard (no longer used)
2	Pipe (no longer used)
3	Linda (no longer used)
4	PVM (no longer used)
5	Vanilla
6	PVMD (no longer used)
7	Scheduler
8	MPI
9	Grid
10	Java
11	Parallel
12	Local
13	VM

17.4 DaemonCore Command Numbers

Table B.4: DaemonCore Commands

60000	DC_RAISESIGNAL
60001	DC_PROCESSEXIT
60002	DC_CONFIG_PERSIST
60003	DC_CONFIG_RUNTIME
60004	DC_RECONFIG
60005	DC_OFF_GRACEFUL
60006	DC_OFF_FAST
60007	DC_CONFIG_VAL
60008	DC_CHILDALIVE
60009	DC_SERVICEWAITPIDS
60010	DC_AUTHENTICATE
60011	DC_NOP
60012	DC_RECONFIG_FULL
60013	DC_FETCH_LOG
60014	DC_INVALIDATE_KEY
60015	DC_OFF_PEACEFUL
60016	DC_SET_PEACEFUL_SHUTDOWN
60017	DC_TIME_OFFSET
60018	DC_PURGE_LOG

17.5 DaemonCore Daemon Exit Codes

Table B.5: DaemonCore Daemon Exit Codes

Exit Code	Description
0	Normal exit of daemon
4	Daemon fatal internal error
44	Failure to write to daemon log
99	`DAEMON_SHUTDOWN` evaluated to `True`

CHAPTER

EIGHTEEN

GLOSSARY

AP (Access Point)
An Access Point (AP) is the machine where users place jobs to be queued to be run. It usually runs the *condor_schedd* and other daemons.

Classad
A classad is a set of key value pairs. Every object in an HTCSS is described by a classad. Classad values can also be an expression, which can be evaluated in the context of another classad, in order to provide matching or ranking policy.

CM (Central Manager)
The Central Manager (CM) is the machine with the central in-memory database (*condor_collector*) of all the services, an accountant and *condor_negotiator*.

Daemon
A long-running process often operating in the background. An older term for "service". The *condor_master*, *condor_collector*, *condor_schedd*, *condor_starter* and *condor_shadow* are some of the daemon in HTCSS.

EP (Execution Point)
The Execution Point (EP), sometimes called the worker node is where jobs run. It is managed by the *condor_startd* daemon, which is responsible for dividing all of the resources the machine into slot.

Job
Job has a very specific meaning in the HTCSS. It is the atomic unit of work in HTCSS. A job is defined by a job classad, which is usually created by *condor_submit* and a submit file. A job can have defined input files, which HTCSS will transfer to the EP. One or more operating system processes can run inside a job. Every job is a member of a cluster of jobs, which have cluster id. Each job also has a "proc id". The job id uniquely identifies every job on an AP, the id is the cluster id followed by a dot followed by the proc id.

Sandbox
Every job has a sandbox associated with it, which is a set of files. The input sandbox is the set of files the job needs as input, and should be transferred to the EP when the job starts. As the job runs, any scratch files created by the job are added to the sandbox. If the job is evicted after running, and WhenToTransferFiles is set to OnExitOrEvict, this sandbox is saved to the AP in the spool directory, and the sandbox is restored to the EP when the job restarts. Any non-input files in the sandbox that exist when the job exits of its own accord are the "output sandbox", which is transferred back to the AP on successful job completion.

Slot
The slot is the location on the EP where the job runs. The *condor_starter* creates a slot, with sufficient Cpu, memory, disk, and other resources for the job to run. The resource usage of a job running in a slot is monitored and reported up to the *condor_startd*, and back to the AP, and, depending on configuration, may be enforced, such that a job that uses too many resources will be evicted from the machine.

Universe
A type of job, describing some of the services it may need on an EP. The default universe, with the minimal additional services needed, is called "vanilla". Other universes include Container, Grid, and VM.

Workflow

A set, possibly ordered, of activities necessary to complete a larger task. In high-throughput computing, each activity is a job. For example, consider the task of searching a large genome for a specific pattern. This might be implemented with 1,000 independent jobs, each searching a subset of the full genome. A different workflow might assemble a genome from sequences. Workflows may be composed; a third workflow compose the first two, so that it assembles the genome from sequences, then searches it for a pattern. The requirements for jobs (or workflows) to run before or after others may be represented by a directed acyclic graph [https://en.wikipedia.org/wiki/Directed_acyclic_graph] (DAG) See the *condor_dagman* (*DAGMan Introduction*) to automatically execute a workflow represented as a dag.

CHAPTER

NINETEEN

INDEX

CHAPTER

TWENTY

LICENSING AND COPYRIGHT

PYTHON MODULE INDEX

c

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X