

Red Hat Enterprise MRG 1.1

Grid User Guide

Use and configuration information for MRG Grid



Lana Brindley

Red Hat Enterprise MRG 1.1 Grid User Guide

Use and configuration information for MRG Grid

Edition 1

Author

Lana Brindley

lbrindle@redhat.com

Copyright © 2008 Red Hat, Inc

Copyright © 2008 Red Hat, Inc. This material may only be distributed subject to the terms and conditions set forth in the Open Publication License, V1.0 or later with the restrictions noted below (the latest version of the OPL is presently available at <http://www.opencontent.org/openpub/>).

Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Distribution of the work or derivative of the work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

1801 Varsity Drive

Raleigh, NC 27606-2072USAPhone: +1 919 754 3700

Phone: 888 733 4281

Fax: +1 919 754 3701

PO Box 13588Research Triangle Park, NC 27709USA

This book explains use and operation of the MRG Grid component of the Red Hat Enterprise MRG distributed computing platform. For installation instructions, see the MRG Grid Installation Guide.

Preface	v
1. Document Conventions	v
1.1. Typographic Conventions	vi
1.2. Pull-quote Conventions	vii
1.3. Notes and Warnings	viii
2. We Need Feedback!	viii
1. Overview	1
2. Configuration	3
2.1. System Wide Configuration File Variables	6
2.2. Logging configuration variables	11
3. Remote configuration tool	17
4. Jobs	21
4.1. Choosing a universe	21
4.2. Writing a submit description file	23
4.3. Time scheduling for job execution	24
4.4. Job Hooks	26
5. User Priorities and Negotiation	31
6. ClassAds	37
6.1. Writing ClassAd expressions	41
7. Policy Configuration	53
7.1. Machine states and transitioning	53
7.2. The <code>condor_startd</code> daemon	57
7.3. Conditions for state and activity transitions	59
7.4. Defining a policy	63
8. The Virtual Machine Universe	71
8.1. Configuring MRG Grid for the virtual machine universe	71
9. High Availability	75
9.1. High availability of the job queue	75
9.2. High availability of the central manager	76
10. Cloud Computing	83
10.1. MRG/EC2 Basic	83
10.2. MRG/EC2 Enhanced	87
11. Concurrency Limits	95
12. Dynamic provisioning	99
13. Low-latency scheduling	101
14. Application Program Interfaces (APIs)	105
14.1. Using the MRG Grid API	105
14.2. Methods	108
15. Frequently Asked Questions	121
15.1. Installing MRG Grid	121
15.2. Running MRG Grid jobs	121
15.3. Running MRG Grid on Windows platforms	124
15.4. Grid computing	125
16. More Information	127

Preface

Red Hat Enterprise MRG

This book contains information on the use and operation of the MRG Grid component of Red Hat Enterprise MRG. Red Hat Enterprise MRG is a high performance distributed computing platform consisting of three components:

1. *Messaging* — Cross platform, high performance, reliable messaging using the Advanced Message Queuing Protocol (AMQP) standard.
2. *Realtime* — Consistent low-latency and predictable response times for applications that require microsecond latency.
3. *Grid* — Distributed High Throughput (HTC) and High Performance Computing (HPC).

All three components of Red Hat Enterprise MRG are designed to be used as part of the platform, but can also be used separately.

MRG Grid

Grid computing allows organizations to fully utilize their computing resources to complete high-performance tasks. By monitoring all resources - rack-mounted clusters and general workstations - for availability, any spare computing power can be redirected towards other, more intensive tasks until it is explicitly required again. This allows a standard networked system to operate in a way that is similar to a supercomputer.

MRG Grid provides High Throughput and High Performance computing and enables enterprises to achieve higher peak computing capacity as well as improved infrastructure utilization by leveraging their existing technology to build high performance grids. MRG Grid provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their jobs to MRG Grid, where they are placed into a queue. MRG Grid then chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

MRG Grid is based on the [Condor Project](http://www.cs.wisc.edu/condor/)¹ developed within the [University of Wisconsin-Madison](http://www.wisc.edu/)². Condor also offers a comprehensive library of freely available documentation in its [Manual](http://www.cs.wisc.edu/condor/manual/)³.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](https://fedorahosted.org/liberation-fonts/)⁴ set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

¹ <http://www.cs.wisc.edu/condor/>

² <http://www.wisc.edu/>

³ <http://www.cs.wisc.edu/condor/manual/>

⁴ <https://fedorahosted.org/liberation-fonts/>

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

Press **Enter** to execute the command.

Press **Ctrl-Alt-F1** to switch to the first virtual terminal. Press **Ctrl-Alt-F7** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **Mono-spaced Bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

Mono-spaced Bold Italic Or Proportional Bold Italic

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type `ssh username@domain.name` at a shell prompt. If the remote machine is `example.com` and your username on that machine is john, type `ssh john@example.com`.

The `mount -o remount file-system` command remounts the named file system. For example, to remount the `/home` file system, the command is `mount -o remount /home`.

To see the version of a currently installed package, use the `rpm -q package` command. It will return a result as follows: `package-version-release`.

Note the words in bold italics above — `username`, `domain.name`, `file-system`, `package`, `version` and `release`. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules (MPMs)*. Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in Mono-spaced Roman and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in Mono-spaced Roman but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo            echo    = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.



Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: <http://bugzilla.redhat.com/bugzilla/> against the product **Red Hat Enterprise MRG**.

When submitting a bug report, be sure to mention the manual's identifier: *Grid_User_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Overview

MRG Grid is a software system that creates a High-Throughput Computing (HTC) environment. It uses the computational ability of many computers connected over a network to complete large or resource-intensive operations. MRG Grid harnesses existing resources by detecting when a workstation becomes available for use, and subsequently relinquishing that resource when it becomes unavailable.

When a job is submitted to MRG Grid, it finds an idle machine on the network and begins running the job on that machine. There is no requirement for machines to share file systems, so machines across an entire enterprise can run a job, including machines in different administrative domains.

MRG Grid does not require an account (login) on machines where it runs a job. This is because of its remote system call technology. Any tasks such as reading or writing from disk are transmitted over the network and performed on the machine where the job was submitted. This ensures that only the resources of the machine are used, without requiring MRG Grid to log in to each machine individually.

MRG Grid implements ClassAds, a clean design that simplifies the user's submission of jobs. All machines in the MRG Grid 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. When a job is submitted, the user specifies a resource request ad, specifying both the required and a desired set of properties. MRG Grid 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, MRG Grid 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 desire of machines in the pool to accept certain types of ads over others.

Groups of researchers, engineers, and scientists have used MRG Grid to establish pools ranging in size from a handful to tens of thousands of workstations. We hope that MRG Grid will help revolutionize your computing environment as well.

Configuration

This section describes how to configure all parts of the MRG Grid system. General information about the configuration files and their syntax is followed by a description of settings that affect all MRG Grid daemons and tools.

The configuration files are used to customize how MRG Grid operates. This section discusses how the configuration files for MRG Grid work and how to customize the files should you need to.

Initial configuration

It is advised that you review the configuration file stored at `/etc/condor/condor_config` before starting MRG Grid. The default configuration sets up a **Personal Condor**. **Personal Condor** is a specific style of installation suited for individual users who do not have their own pool of machines. To allow other machines to join your pool you will need to customize the `HOSTALLOW_WRITE` option. Open the `/etc/condor/condor_config` file in your preferred text editor and locate the section titled **Host/IP Access Levels**. The value for this option should be set to allow machines to join your pool and submit jobs. Any machine that you give write access to using the `HOSTALLOW_WRITE` option should also be given read access using the `HOSTALLOW_READ` option:

```
HOSTALLOW_WRITE = *.your.domain.com
```



Warning

The simplest option is to change the `HOSTALLOW_WRITE` option to `HOSTALLOW_WRITE = *`. However, this will allow anyone to submit jobs or add machines to your pool. This is a serious security risk and therefore not recommended.

There are different configuration files offering varying levels of control. The files are parsed in the following order:

1. *Global configuration file*

The global configuration file is shared by all machines in the pool. For ease of administration, this file can be located on a shared file system. If this is not possible, it will need to be the same across all nodes. Ideally, this file will only be customized through the local configuration files.

MRG Grid will look in different places for the global configuration file, in the following order:

- a. The filename specified in the `CONDOR_CONFIG` environment variable
- b. `/etc/condor/condor_config`
- c. `/usr/local/etc/condor_config`
- d. `~condor/condor_config`



Note

If a file is specified in the `CONDOR_CONFIG` environment variable and there's a problem reading that file, MRG Grid will print an error message and exit. It will not

continue to search the other options. Leaving the `CONDOR_CONFIG` environment variable blank will ensure that MRG Grid will search through the other options.

If a valid configuration file is not found in any of the searched locations, MRG Grid will print an error message and exit.

2. Local configuration file

A local configuration file exists for each machine. Settings in this configuration file will override settings in the global file for that machine.

The location of the local configuration file is stored in the global configuration file, using the `LOCAL_CONFIG_FILE` setting. This can be a list of files or a single file. If this is not set, no local configuration file is used.

Once MRG Grid has completed parsing the four configuration files, it will check for environment variables. These configuration variables are prefixed by the string `_CONDOR_` or `_condor_`. MRG Grid parses environment variables last, subsequently any settings made this way will override conflicting settings in the configuration files.

Adding entries to configuration files

1. All entries in a configuration file use the same syntax. The entries are in the form:

```
# This is a comment
SUBSYSTEM_NAME.CONFIG_VARIABLE = VALUE
```

Things to note about the syntax:

- Each valid entry requires an operator of `=`
 - A line that does not contain an operator and is prefixed by a `#` symbol will be treated as a comment and ignored
 - The `SUBSYSTEM_NAME` is optional
 - There must be white space on either side of the `=` sign
2. An entry can continue over multiple lines by placing a `\` character at the end of the line to be continued. For example:

```
ADMIN_MACHINES = condor.example.com, raven.example.com, \
stork.example.com, ostrich.example.com \
bigbird.example.com
```



Important

The line continuation character will also work within a comment, which will cause MRG Grid to ignore the second line. The following example would be ignored entirely:

```
# This comment has line continuation \
```

```
characters, so FOO will not be set \
FOO = BAR
```

Executing a Program to Produce Configuration Entries

1. MRG Grid can run a specialized program to obtain configuration entries. To run a program from the configuration file, insert a `|` character at the end of the line. This syntax will only work with `CONDOR_CONFIG`, or the configuration variable `LOCAL_CONFIG_FILE`. For example, to run a program located at `/bin/make_the_config` to populate the local configuration file, use the following entry:

```
LOCAL_CONFIG_FILE = /bin/make_the_config|
```

Pre-Defined Configuration Macros

MRG Grid provides pre-defined configuration macros to help simplify configuration. These settings are determined automatically and cannot be overwritten.

FULL_HOSTNAME

The fully qualified hostname of the local machine (domain name and hostname)

HOSTNAME

The hostname of the local machine

IP_ADDRESS

The local machine's IP address as an ASCII string

TILDE

The full path to the home directory of the user Condor, if the 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 MRG Grid system. The possible subsystem names are:

- STARTD
- SCHEDD
- MASTER
- COLLECTOR
- NEGOTIATOR
- KBDD
- SHADOW
- STARTER
- GRIDMANAGER
- HAD

- REPLICATION
- QUILL

Static Pre-Defined Configuration Macros

These settings are determined automatically and cannot be overwritten.

ARCH

Defines the string used to identify the architecture of the local machine to MRG Grid. This allows jobs to be submitted for a given platform and MRG Grid will force them to run on the correct machines

OPSYS

Defines the string used to identify the operating system of the local machine to MRG Grid. If it is not defined in the configuration file, MRG Grid will automatically insert the operating system of the current machine as determined by the `uname` command

UNAME_ARCH

The architecture as reported by the `uname` command's *machine* field

UNAME_OP SYS

The operating system as reported by the `uname` command's *sysname* field

PID

The process ID of the daemon or tool

PPID

The process ID of the daemon or tool's parent process

USERNAME

The name of the user running the daemon or tool. For daemons started as the root user, but running under another user, that username will be used instead of root

2.1. System Wide Configuration File Variables

These settings affect all parts of the MRG Grid system.

FILESYSTEM_DOMAIN

Defaults to the fully qualified hostname of the current machine.

UID_DOMAIN

Defaults to the fully qualified hostname of the current machine it is evaluated on.

COLLECTOR_HOST

The host name of the machine where the `condor_collector` is running for your pool. `COLLECTOR_HOST` must be defined for the pool to work properly.

This setting can also be used to specify the network port of the `condor_collector`. The port is separated from the host name by a colon. To set the network port to 1234, use the following syntax:

```
COLLECTOR_HOST = $(CONDOR_HOST):1234
```

If no port is specified, the default port of 9618 is used.

CONDOR_VIEW_HOST

The host name of the machine where the **CondorView** server is running. This service is optional, and requires additional configuration to enable it. If **CONDOR_VIEW_HOST** is not defined, no **CondorView** server is used.

RELEASE_DIR

The full path to the MRG Grid release directory, which holds the **bin**, **etc**, **lib** and **sbin** directories. There is no default value for **RELEASE_DIR**.

BIN

The directory where user-level programs are installed.

LIB

The directory where libraries used to link jobs for MRG Grid's standard universe are stored. The **condor_compile** program uses this macro to find the libraries, so it must be defined for **condor_compile** to function.

LIBEXEC

The directory where support commands for Condor are placed. Do not add this directory to a user or system-wide path.

INCLUDE

The directory where header files are placed.

SBIN

The directory where system binaries and administrative tools are installed. The directory defined at **SBIN** should also be in the path of users acting as Condor administrators.

LOCAL_DIR

The location of the local Condor directory on each machine in your pool. One common option is to use the condor user's home directory which may be specified with **\$(TILDE)**, in this format:

```
LOCAL_DIR = $(TILDE)
```

On machines with a shared file system, where the directory is shared among all machines in your pool, use the **\$(HOSTNAME)** macro and have a directory with many sub-directories, one for each machine in your pool. For example:

```
LOCAL_DIR = $(tilde)/hosts/$(hostname)
```

or:

```
LOCAL_DIR = $(release_dir)/hosts/$(hostname)
```

LOG

The directory where each daemon writes its log files. The names of the log files themselves are defined with other macros, which require the **\$(LOG)** macro.

SPOOL

The directory where files used by `condor_schedd` are stored, including the job queue file and the initial executables of any jobs that have been submitted. If a given machine executes jobs but does not submit them, it does not require a `SPOOL` directory.

EXECUTE

The scratch directory for the local machine. The scratch directory is used as the destination for 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 is specified. If a given machine submits jobs but does not execute them, it does not require an `EXECUTE` directory. To customize the execute directory independently for each batch slot, use `SLOTx_EXECUTE`.

LOCAL_CONFIG_FILE

The location of the local configuration file for each machine in the pool. The value of `LOCAL_CONFIG_FILE` is treated as a list of files. The items in the list are delimited by either commas or spaces. The list is processed in the order given (with settings in later files overwriting values from previous files). This allows the use of one global configuration file for multiple platforms in the pool. If `LOCAL_CONFIG_FILE` is not defined, and `REQUIRE_LOCAL_CONFIG_FILE` has not been explicitly set to false, an error will be caused.

REQUIRE_LOCAL_CONFIG_FILE

A boolean value that defaults to true. This will cause MRG Grid to exit with an error if any file listed in `LOCAL_CONFIG_FILE` cannot be located. If the value is set to false, MRG Grid will ignore any local configuration files that cannot be located and continue. If `LOCAL_CONFIG_FILE` is not defined, and `REQUIRE_LOCAL_CONFIG_FILE` has not been explicitly set to false, an error will be caused.

CONDOR_IDS

The User ID (UID) and Group ID (GID) for Condor daemons to use when run by the root user. This value can also be set using the `CONDOR_IDS` environment variable. The syntax is:

```
CONDOR_IDS = UID.GID
```

To set a UID of 1234 and a GID of 5678, use the following setting:

```
CONDOR_IDS = 1234.5678
```

If `CONDOR_IDS` is not set and the daemons are run by the root user, MRG Grid will search for a condor user on the system, and use that UID and GID.

CONDOR_ADMIN

An email address for MRG Grid to send messages about any errors that occur in the pool, such as a daemon failing.

CONDOR_SUPPORT_EMAIL

The email address to be included in the footer of all email sent out by MRG Grid. The footer reads:

```
Email address of the local MRG Grid administrator: admin@example.com
```

If this setting is not defined, MRG Grid will use the address specified in `CONDOR_ADMIN`.

MAIL

The full path to a text based email client, such as `/bin/mail`. The email client must be able to accept mail messages and headers as standard input (`STDIN`) and use the `-s` command to specify a subject for the message. On all platforms, the default shipped with MRG Grid should work. This setting will only need to be changed if the installation is in a non-standard location. The `condor_schedd` will not function unless **MAIL** is defined.

RESERVED_SWAP

The amount (in megabytes) of memory swap space reserved for use by the machine. MRG Grid will stop initializing processes if the amount of available swap space falls below this level. The default value is 5MB.

RESERVED_DISK

The amount (in megabytes) of disk space reserved for use by the machine. When reporting, MRG Grid will subtract this amount from the total amount of available disk space. The default value is 0MB (zero megabytes).

LOCK

MRG Grid creates lock files in order to synchronize access to various log files. If the local Condor directory is not on a local partition, be sure to set the **LOCK** entry to avoid problems with file locking.

The user and group that MRG Grid runs as need to have write access to the directory that contains the lock files. If no value for **LOCK** is provided, the value of **LOG** is used.

HISTORY

The location of the history file, which stores information about all jobs that have completed on a given machine. This setting is used by `condor_schedd` to append information, and `condor_history` the user-level program used to view the file. The default value is `$(SPOOL)/history`. If not defined, no history file will be kept.

ENABLE_HISTORY_ROTATION

A boolean value that defaults to true. When false, the history file will not be rotated, and the history will continue to grow in size until it reaches the limits defined by the operating system. The rotated files are stored in the same directory as the history file. Use **MAX_HISTORY_LOG** to define the size of the file and **MAX_HISTORY_ROTATIONS** to define the number of files to use when rotation is enabled.

MAX_HISTORY_LOG

Defines the maximum size (in bytes) for the history file, before it is rotated. Default value is 20,971,520 bytes (20MB). This parameter is only used if history file rotation is enabled.

MAX_HISTORY_ROTATIONS

Defines how many files to use for rotation. Defaults to 2. In this case, there may be up to three history files at any one time - two backups and the history file that is currently being written. The oldest file will be removed first on rotation.

MAX_JOB_QUEUE_LOG_ROTATIONS

The job queue database file is periodically rotated in order to save disk space. This option controls how many rotated files are saved. Defaults to 1. In this case, there may be up to two history files at any one time - the backup which has been rotated out of use, and the history file that is currently being written. The oldest file will be removed first on rotation.

NO_DNS

A boolean value that defaults to false. When true, MRG Grid constructs hostnames automatically using the machine's IP address and **DEFAULT_DOMAIN_NAME**.

DEFAULT_DOMAIN_NAME

The domain name for the machine. This value is appended to the hostname in order to create a fully qualified hostname. This value should be set in the global configuration file, as MRG Grid can depend on knowing this value in order to locate the local configuration files. The default value is an example, and must be changed to a valid domain name. This variable only operates when **NO_DNS** is set to true.

EMAIL_DOMAIN

Defines the domain to use for email. If a job is submitted and the user has not specified *notify_user* in the submit description file, MRG Grid will send any email about that job to *username@UID_DOMAIN*. If all the machines share a common UID domain, but email to this address will not work, you will need to define the correct domain to use. In many cases, you can set **EMAIL_DOMAIN** to **FULL_HOSTNAME**.

CREATE_CORE_FILES

A boolean value that is undefined by default, in order to allow the default operating system value to take precedence. If set to true, the Condor daemons will create core files in the **LOG** directory in the case of a segmentation fault (segfault). When set to false no core files will be created. When left undefined, it will retain the setting that was in effect when the Condor daemons were started. Core files are used primarily for debugging purposes.

ABORT_ON_EXCEPTION

A boolean value that defaults to false. When set to true MRG Grid will abort on a fatal internal exception. If **CREATE_CORE_FILES** is also true, MRG Grid will create a core file when an exception occurs.

Q_QUERY_TIMEOUT

The amount of time (in seconds) that **condor_q** will wait when trying to connect to **condor_schedd**, before causing a timeout error. Defaults to 20 seconds.

DEAD_COLLECTOR_MAX_AVOIDANCE_TIME

For pools where High Availability is in use. Defines the maximum time (in seconds) to wait in between checks for a failed primary **condor_collector** daemon. If connections to the dead daemon take very little time to fail, new query attempts become more frequent. Defaults to 3600 (1 hour).

NETWORK_MAX_PENDING_CONNECTS

The maximum number of simultaneous network connection attempts. **condor_schedd** can 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. Defaults to 80% of the process file descriptor limit, except on Windows operating systems, where the default is 1600.

WANT_UDP_COMMAND_SOCKET

A boolean value that defaults to true. When true, Condor daemons will create a UDP command socket in addition to the required TCP command socket. When false, only the TCP command socket will be created. If you modify this setting, you will need to restart all Condor daemons.

MASTER_INSTANCE_LOCK

The name of the lock file to prevent multiple `condor_master` daemons from starting. This is useful when using shared file systems like NFS, where the lock files exist on a local disk. Defaults to `$(LOCK)/InstanceLock`. The `$(LOCK)` macro can be used to specify the location of all lock files, not just the `condor_master` instance lock. If `$(LOCK)` is undefined, the master log itself will be locked.

SHADOW_LOCK

The lock file to be used for access to the `ShadowLog` file. It must be a separate file from the `ShadowLog`, since the `ShadowLog` might be rotated and access will need to be synchronized across rotations. This macro is defined relative to the `$(LOCK)` macro.

2.2. Logging configuration variables

These variables control logging. Many of these variables apply to each of the possible subsystems. In each case, replace the word *SUBSYSTEM* with the name of the appropriate subsystem. The possible subsystems are:

- **STARTD**
- **SCHEDD**
- **MASTER**
- **COLLECTOR**
- **NEGOTIATOR**
- **KBDD**
- **SHADOW**
- **STARTER**
- **SUBMIT**
- **GRIDMANAGER**
- **TOOL**
- **HAD**
- **REPLICATION**
- **QUILL**

SUBSYSTEM_LOG

The 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.

MAX_SUBSYSTEM_LOG

The maximum size a log file is allowed to grow to, in bytes. Each 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, thus the maximum space devoted to logging for any one program will be twice

the maximum length of its log file. A value of *0* specifies that the file may grow without bounds. Defaults to 1MB.

TRUNC_SUBSYSTEM_LOG_ON_OPEN

When *TRUE*, the log will be restarted with an empty file every time the program is run. When *FALSE* new entries will be appended. Defaults to *FALSE*.

SUBSYSTEM_LOCK

Specifies the lock file used to synchronize additions to the log file. It must be a separate file from the `$(SUBSYSTEM_LOG)` file, since that file can be rotated and synchronization should occur across log file rotations. A lock file is only required for log files which are accessed by more than one process. Currently, this includes only the SHADOW subsystem. This macro is defined relative to the `$(LOCK)` macro.

FILE_LOCK_VIA_MUTEX

This setting is for Windows platforms only. When *TRUE* log are able to be locked using a mutex instead of by file locking. This can correct problems on Windows platforms where processes starve waiting for a lock on a log file. Defaults to *TRUE* on Windows platforms. Always set to *FALSE* on Unix platforms.

ENABLE_USERLOG_LOCKING

When *TRUE* the job log specified in the submit description file is locked before being written to. Defaults to *TRUE*.

TOUCH_LOG_INTERVAL

The time interval between daemons creating (using the `touch` command) log files, in seconds. 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

Formatting of the current time at the start of each line in the log files. When *TRUE*, Unix Epoch Time is used. When *FALSE*, the time is printed in the local timezone using the syntax:

```
[Month]/[Day] [Hour]:[Minute]:[Second]
```

. Defaults to *FALSE*.

SUBSYSTEM_DEBUG

The Condor daemons are all capable of producing different levels of output. All daemons default to `D_ALWAYS`. This logs all messages. Settings are a comma or space-separated list of these values:

- **D_ALL**

Enables all of the debug levels at once. There is no need to list any other debug levels in addition to `D_ALL`. This setting generates an extremely large amount of output.

- **D_FULLDEBUG**

Verbose output. Only very frequent log messages for very specific debugging purposes are excluded.

- **D_DAEMONCORE**

Logs messages that specific to DaemonCore, such as timers the daemons have set and the commands that are registered.

- **D_PRIV**

Logs messages about privilege state switching.

- **D_COMMAND**

With this flag set, any daemon that uses DaemonCore will print out a log message whenever a command is received. 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.

- **D_LOAD**

The `condor_startd` records the load average on the machine where it is running. Both the general system load average, and the load average being generated by MRG Grid activity 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` subsystem.

- **D_KEYBOARD**

Logs messages related to the values for remote and local keyboard idle times. This flag only affects the `condor_startd` subsystem.

- **D_JOB**

Logs the contents of any job ClassAd that the `condor_schedd` sends to claim the `condor_startd`. This flag only affects the `condor_startd` subsystem.

- **D_MACHINE**

Logs the contents of any machine ClassAd that the `condor_schedd` sends to claim the `condor_startd`. This flag only affects the `condor_startd` subsystem.

- **D_SYSCALLS**

Logs remote syscall requests and return values.

- **D_MATCH**

Logs messages for every match performed by the `condor_negotiator`.

- **D_NETWORK**

All daemons will log a message on every TCP accept, connect, and close, and on every UDP send and receive.

- **D_HOSTNAME**

Logs verbose messages explaining how host names, domain names and IP addresses have been resolved.

- **D_CKPT**

The Condor process checkpoint support code, which is linked into a standard universe user job, will output some low-level details about the checkpoint procedure. This logging appears only in the `$(SHADOW_LOG)`.

- **D_SECURITY**

Logs messages regarding secure network communications. Includes messages about negotiation of a socket authentication mechanism, management of a session key cache, and messages about the authentication process.

- **D_PROCFAMILY**

Logs messages regarding management of families of processes. A process family is defined as a process and all descendents of that process.

- **D_ACCOUNTANT**

Logs messages regarding the computation of user priorities.

- **D_PROTOCOL**

Log messages regarding the protocol for the matchmaking and resource claiming framework.

- **D_PID**

This flag is used to change the formatting of all log messages that are printed. If **D_PID** is set, the process identifier (PID) of the process writing each line to the log file will be recorded.

- **D_FDS**

This flag is used to change the formatting of all log messages that are printed. If **D_FDS** is set, the file descriptor that the log file was allocated will be recorded.

ALL_DEBUG

Used to make all subsystems share a debug flag. For example, to turn on all debugging in all subsystems, set **ALL_DEBUG = D_ALL**.

TOOL_DEBUG

Uses the same values (debugging levels) as **SUBSYSTEM_DEBUG** to describe the amount of debugging information sent to *STDERR* for Condor tools.

SUBMIT_DEBUG

Uses the same values (debugging levels) as **SUBSYSTEM_DEBUG** to describe the amount of debugging information sent to *STDERR* for **condor_submit**.

SUBSYSTEM_[LEVEL]_LOG

This is the name of a log file for messages at a specific debug level for a specific subsystem. If the debug level is included in `$(SUBSYSTEM_DEBUG)`, then all messages of this debug level will be written both to the `$(SUBSYSTEM_LOG)` file and the `$(SUBSYSTEM_[LEVEL]_LOG)` file.

MAX_SUBSYSTEM_[LEVEL]_LOG

Similar to **MAX_SUBSYSTEM_LOG**.

TRUNC_SUBSYSTEM_[LEVEL]_LOG_ON_OPEN

Similar to **TRUNC_SUBSYSTEM_LOG_ON_OPEN**.

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 it is not defined.

MAX_EVENT_LOG

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* allows the file to grow continuously. Defaults to 1MB.

EVENT_LOG_USE_XML

When *TRUE*, events are logged in XML format. Defaults to *FALSE*.

EVENT_LOG_JOB_AD_INFORMATION_ATTRS

A comma-separated list of job ClassAd attributes. When evaluated, these values form a new event of *JobAdInformationEvent*. This new event is placed in the event log in addition to each logged event.

Remote configuration tool

The remote configuration feature simplifies configuration and management of a pool. It allows a single machine to configure all nodes in a condor pool and easily modify or change a node's configuration. Any change that impacts other nodes in the system will also be handled appropriately. The configuration tools allow management of multiple independent pools, and can build a condor node from the ground up needing only an operating system and the appropriate remote configuration packages.

Configuring the server for remote configuration

1. The remote configuration feature requires a server machine with the **condor-remote-configuration-server** package. Only one machine in a cluster should act as the server, and should be the same machine that the MRG Management Console is installed on. Install the package on the server using **yum**:

```
# yum install condor-remote-configuration-server
```

2. A **CNAME** record will need to be added to the DNS configuration with the name *puppet*. It needs to point to the machine that has the **condor-remote-configuration-server** installed.
3. Copy the **puppet.conf.master** file to the **/etc/puppet** directory using the following command:

```
$ cp -f /etc/opt/grid/examples/puppet.conf.master /etc/puppet/  
puppet.conf
```

4. Create the **site.pp** file:

```
$ echo 'import "condor"' >> /etc/puppet/manifests/site.pp
```

5. Start the service from the shell prompt:

```
# service puppetmaster start  
Starting puppetmaster service:          [ OK ]
```

Configuring a client for remote configuration

1. The remote configuration feature for client machines requires the **condor-remote-configuration** package. Install the package using **yum**:

```
# yum install condor-remote-configuration
```

2. Copy the `puppet.conf.client` file to the `/etc/puppet` directory using the following command:

```
$ cp -f /etc/opt/grid/examples/puppet.conf.client /etc/puppet/  
puppet.conf
```

Then copy the `namespaceauth.conf` file to `/etc/puppet`:

```
$ cp -f /etc/opt/grid/examples/namespaceauth.conf /etc/puppet
```

3. Open the `namespaceauth.conf` configuration file in your preferred text editor and locate the line that states `allow <puppetmaster.fqdn>` entry. Replace the `<puppetmaster.fqdn>` text with the fully qualified domain name of the server machine. This is the machine that has the `condor-remote-configuration-server` package installed:

```
allow server.example.com
```

4. Start the service from the shell prompt:

```
# service puppet start  
Starting puppet service:          [ OK ]
```

Using the remote configuration tool

To use the remote configuration tools to configure a node, you will need to know the node's fully qualified domain name. This name is used by the node to identify itself to the configuration system.

1. To configure a node, use the following syntax:

```
$ condor_configure_node -n [name of node] [action] [features]
```

Use the `--help` option to see a full list of possible commands:

```
$ condor_configure_node --help
```

2. The possible actions are:
 - `add`: Used to add features to the node being configured. If any additional information is required, the tool will prompt for it.
 - `delete`: Used to remove features on the node.
3. There are three items that the tool will ask for, regardless of the options in use:

- a. *Schedulers*: If the node being configured is not a scheduler, then the configuration tool will prompt for the list of schedulers the node should be allowed to submit to. First it will prompt for the default scheduler, then for a comma separated list of additional schedulers. This entry should contain the fully qualified domain names of the schedulers in the pool. If the pool has a high availability scheduler, use **ha-schedd@**.
 - b. *Collector Name*: This is a human readable text field that will identify the pool. This value will be set as the **COLLECTOR_NAME** for the node.
 - c. *QMF Broker Information*: The QMF Broker is the AMQP broker that is used to communicate with the MRG Management Console. The configuration tool will prompt for the IP or hostname where the broker is running, as well as the port the broker is listening on. If no port is provided, the default port will be used.
4. The configuration tool will always prompt you to save the configuration. When it is saved, the tool will automatically check the configuration of all known nodes, to ensure they are up to date.
 5. The remote configuration tool controls the content of the `~/condor_config.local` local configuration file. It is possible to provide a custom configuration for a node by creating a file named `~/condor_config.overrides` and adding configuration entries to that file.



Important

Be very cautious when adding entries to a `~/condor_config.overrides` file. Settings in this file will override any settings provided by the remote configuration feature. This can result in lost or incorrect functionality of the features controlled by the remote configuration feature.

This example gives some common uses of the remote configuration tool.

To enable a machine named **condor_ca.domain.com** to be a High Available Central Manager:

```
$ condor_configure_node -n condor_ca.domain.com -a ha_central_manager
```

To enable a machine name **twofer.domain.com** to be both a scheduler and an execute node:

```
$ condor_configure_node -n twofer.domain.com -a scheduler,startd
```

To remove the execute functionality from **twofer.domain.com**:

```
$ condor_configure_node -n twofer.domain.com -d startd
```

Example 3.1. Examples of use of the remote configuration tool

Jobs

Submitting a job consists of six main steps:

Prepare the job

Jobs must be able to run without interaction by the user, as MRG Grid runs unattended and in the background. This means that all interactive input and output must be automated. MRG Grid can redirect standard input (STDIN) and console output (STDOUT and STDERR) to and from files. Create any files you need to perform these functions, and test them to make sure they will run correctly with the job.

Choose a universe

MRG Grid uses a runtime environment, called a *universe*, to determine how a job is handled as it is being processed. You will need to select which universe under which to run the job. For more information on choosing a universe, see [Section 4.1, “Choosing a universe”](#)

Write a submit description file

The submit description file controls the details of the job submission. The file contains information about the job, such as:

- Which executable to run
- The files to use for keyboard and screen data
- The platform required to run the program
- The universe to use. If you are unsure which universe to use, select the vanilla universe.
- Where to send notification emails
- How many times to run a program

You will need to write a submit description file to go with the job. For more information on writing a submit description file, see [Section 4.2, “Writing a submit description file”](#)

Submit the job

Submit the program using the `condor_submit` command.

Monitor the progress of the job

Once a job has been submitted, MRG Grid will go ahead and run the job. You can monitor the job's progress using the `condor_q` and `condor_status` commands.

Finishing the job

When your job finishes, MRG Grid will notify you of the exit status of your job and various statistics about the performance, including time used and input/output performed. If you are using a log file for the job, the exit status will also be recorded. You can also remove a job from the queue prematurely with the `condor_rm` command.

4.1. Choosing a universe

MRG Grid uses an execution environment, called a *universe*. Jobs will run in the vanilla universe by default, unless a different universe is specified in the submit description file.

Currently, the following universes are supported:

- Vanilla
- Java
- VM (for Xen)
- Grid
- Scheduler
- Local
- Parallel

Vanilla universe

The vanilla universe is the default universe, and has very few restrictions.

If a vanilla universe job is partially completed when the remote machine has to be returned, or fails for some other reason, MRG Grid will perform one of two actions. It will either suspend the job, in case it can complete it on the same machine at a later time, or it will cancel the job and restart it again on another machine in the pool.

Grid universe

The Grid Universe provides jobs access to external schedulers. For example, jobs submitted to EC2 are routed through the Grid Universe.

Java universe

The java universe allows users to run jobs written for the Java Virtual Machine (JVM). 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. MRG Grid will automatically locate details such as finding the JVM binary and setting the classpath.

Scheduler universe

The scheduler universe is primarily for use with the `condor_dagman` daemon. It allows users to submit lightweight jobs to be run immediately, alongside the `condor_schedd` daemon on the host machine. Scheduler universe jobs are not matched with a remote machine, and will never be pre-empted.

The scheduler universe, however, offers few features and limited policy support. The local universe is a better choice for most jobs which must run on the submitting machine, as it offers a richer set of job management features, and is more consistent with the other universes.

Local universe

The local universe allows a 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 is executed immediately, on the machine where the job is submitted. Jobs submitted in the local universe will never be pre-empted.

Parallel universe

The parallel universe is used to run jobs that require simultaneous startup on multiple execution nodes, such as Message Passing Interface (MPI) jobs.

VM universe

The VM universe allows for the running of Xen virtual machine instances. A VM universe job's lifecycle is tied to the virtual machine that is being run.

4.2. Writing a submit description file

A job is submitted for execution using `condor_submit`, which requires a file called a *submit description file*. The submit description file contains the name of the executable, the initial working directory and command-line arguments.

Example submit description files

The following examples are common submit description files.

This example submits a job called *physica*.

Since no platform is specified in this description file, MRG Grid will default to run the job on a machine which has the same architecture and operating system as the machine from which it was submitted. The submit description file does not specify input, output, and error commands, this will cause MRG Grid to use `/dev/null` for all `STDIN`, `STDOUT` and `STDERR`. A log file, called *physica.log* will be created. When the job finishes, its exit conditions will be noted in the log file. It is recommended that you always have a log file.

```
Executable      = physica
Log             = physica.log
Queue
```

Example 4.1. Basic submit description file

This example queues two copies of the program *mathematica*.

The first copy will run in directory *run_1*, and the second will run in directory *run_2*. For both queued copies, **STDIN** will be **test.data**, **STDOUT** will be **loop.out**, and **STDERR** will be **loop.error**. There will be two sets of files written, as the files for each job are written to the individual directories. The job will be run in the vanilla universe.

```
Executable      = mathematica
Universe        = vanilla
input           = test.data
output          = loop.out
error           = loop.error
Log             = mathematica.log

Initialdir      = run_1
Queue

Initialdir      = run_2
Queue
```

Example 4.2. Using multiple directories in a submit description file

This example queues 150 runs of program *chemistria*.

This job must be run only on Linux workstations that have greater than 32 megabytes of physical memory. If machines with greater than 64 megabytes of physical memory are available, the job should be run on those machines as a preference. This submit description file also advises that it will use up to 28 megabytes of memory when running. Each of the 150 runs of the program is given its own process number, starting with process number 0. In this case, **STDIN**, **STDOUT**, and **STDERR** will refer to **in.0**, **out.0** and **err.0** for the first run of the program, and **in.1**, **out.1** and **err.1** for the second run of the program. A log file will be written to *chemistria.log*.

```
Executable      = chemistria
Requirements    = Memory >= 32 && OpSys == "LINUX" && Arch == "X86_64"
Rank            = Memory >= 64
Image_Size      = 28 Meg

Error           = err.%(Process)
Input           = in.%(Process)
Output          = out.%(Process)
Log             = chemistria.log

Queue 150
```

Example 4.3. Specifying execution requirements in a submit description file

4.3. Time scheduling for job execution

MRG Grid allows jobs to begin execution at a later time. This feature can be accessed by adding a deferral time to the submit description file. Jobs running on a Unix platform can also be set to run periodically.

Deferring jobs

Job deferral allow the submitter to specify an exact date and time at which a job is to begin. MRG Grid attempts to match the job to an execution machine as normal, however, the job will wait until the specified time to begin execution. Submitters can also provide details for how to handle a job that misses it's specified execution time.

The *deferral time* is defined in the submit description file as a Unix Epoch timestamp. Unix Epoch timestamps are the number of seconds elapsed since midnight on January 1, 1970, Coordinated Universal Time.

After a job has been matched and the files transferred to a machine for execution, MRG Grid checks to see if the job has a deferral time. If it does, and the time for execution is still in the future, the job will wait. While it waits, *JobStatus* will indicate that the job is running.

If a job reports that the time for execution is in the past - that is, the job has failed to execute when it should have - then the job is evicted from the execution machine and put on hold in the queue. This could occur if the files were transferred too slowly, or because of a network outage. This can be avoided by specifying a *deferral window* within which the job can still begin. When a job arrives too late, the difference between the current time and the deferral time is calculated. If the difference is within the deferral window, the job will begin executing immediately.

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.

If a job is waiting to begin execution and a `condor_hold` command is issued, the job is removed from the execution machine and put on hold. If a job is waiting to begin execution and a `condor_suspend` command is issued, the job continues to wait, and when the deferral time arrives, the job will be immediately suspended.

Limitations to the job deferral feature

There are some limitations to the job deferral feature:

- Job deferral will not work with scheduler universe jobs. If a deferral time is specified in a job submitted to the scheduler universe, a fatal error will occur.
- Job deferral times are based on the execution machine's system clock, not the submission machine's system clock.
- A job's *JobStatus* attribute will always show the job as *running* when job deferral is used. As of the 1.1 release of MRG Grid, there is no way to distinguish between a job that is executing and a job that has been deferred and is waiting to begin execution. This will be addressed in future versions.

Example submit description files

The following examples show how to set job deferral times and deferral windows.

¹ <http://www.epochconverter.com/>

This example starts a job on January 1, 2008 at 09:00:00 GMT.

To calculate the date and time as Unix epoch time on a Unix-based machine, use the `date` program from the shell prompt with the following syntax:

```
$ date --date "MM/DD/YYYY HH:MM:SS" +%s
```

You could also use an online time converter, such as the [Epoch Converter](#)¹.

January 1, 2008 at 09:00:00 GMT converts to 1199178000 in Unix epoch time. The line you will need to add to the submit description file is:

```
deferral_time = 1199178000
```

Example 4.4. Setting the deferral time using Unix epoch time

This example starts a job one minute from the submit time.

This parameter uses a value in seconds to determine the start time:

```
deferral_time = (CurrentTime + 60)
```

Example 4.5. Setting the deferral time in seconds from submission time

This example sets a deferral window of 120 seconds, within which a job can begin execution

This parameter uses a value in seconds to determine the length of the deferral window:

```
deferral_time = (CurrentTime + 60)
```

Example 4.6. Setting a deferral window in the submit description file

This example schedules a job to begin on January 1st, 2010 at 09:00:00 GMT, and sets a deferral prep time of 1 minute.

The `deferral_prep_time` attribute delays the job from being matched until the specified number of seconds before the job is to begin execution. This prevents the job from being assigned to a machine long before it is due to start and unnecessarily tying up resources.

```
deferral_time      = 1262336400
deferral_prep_time = 60
```

Example 4.7. Setting a deferral prep time in the submit description file

4.4. Job Hooks

A *hook* is an external program or script invoked during the life cycle of a job. External programs or scripts can contain external code and logic, which MRG Grid can then hook and use to execute the job. This can result in an easier and more direct method of interfacing with an external scheduling system, although some of the flexibility offered by the Condor daemons might be lost.

Hooks can also be useful where a job needs to be performed behind a firewall, but requires data from outside. The hook only needs an outbound network connection to complete the task, thereby being able to operate from behind the firewall, without the intervention of a connection broker.

Hooks can also be used to manage the execution of a job. They can be used to fetch execution environment variables, update information about the job as it runs, notify when it exits, or take special action if the job is evicted.

Periodically, MRG Grid will send out a hook to see if there is any work to be fetched. When a new job is hooked, it is evaluated to decide if it should be executed, and whether or not it should pre-empt any currently running jobs. If the resources are not available to run the hooked job, it will be refused, and will need to be hooked again.

When a job is accepted the `condor_startd` daemon will spawn a `condor_starter` daemon to manage the execution of the job. The job will then be treated as any other, and can potentially be pre-empted by a higher ranking job.

Hooks used for fetching jobs are handled either by the `condor_startd` or the `condor_starter` daemon. The different types of hooks are:

HOOK_FETCH_WORK

This hook returns any work to be considered by the `condor_startd` daemon. The *FetchWorkDelay* configuration variable determines how long the daemon will wait between attempts to fetch work.

HOOK_REPLY_FETCH

When a new job is hooked with `HOOK_FETCH_WORK`, the `condor_startd` decides whether to accept or reject the fetched job and uses `HOOK_REPLY_FETCH` to send notification of this decision.

Importantly, this hook is advisory in nature. `condor_startd` will not wait for the results of `HOOK_REPLY_FETCH` before performing other actions. The output and exit status of this hook is ignored.

HOOK_EVICT_CLAIM

`HOOK_EVICT_CLAIM` is invoked by `condor_startd` in order to evict a fetched job. This hook is also advisory in nature.

HOOK_PREPARE_JOB

When a job is going to be run, `condor_starter` invokes `HOOK_PREPARE_JOB`. This will execute command to set up the job environment and perform actions such as transferring input files.

`condor_starter` will wait for `HOOK_PREPARE_JOB` to be returned before it attempts to execute the job. An exit status of 0 indicates that the job has been prepared successfully. If the hook returns with an exit status that is not 0 - indicating that an error has occurred - the job will be aborted.

HOOK_UPDATE_JOB_INFO

This hook is invoked periodically during the life of a job to update job status information. By default, this hook is invoked for the first time 8 seconds after the job is begun. This can be changed by adjusting the `STARTER_INITIAL_UPDATE_INTERVAL` configuration variable. The frequency of the check can be adjusted with the `STARTER_UPDATE_INTERVAL` configuration variable, which defaults to 300 seconds (5 minutes).

HOOK_JOB_EXIT

This hook is invoked whenever a job exits - either through completion or eviction.

The `condor_starter` will wait for this hook to return before taking any further action.

Defining the `FetchWorkDelay` Expression

The `condor_startd` daemon will attempt to fetch new work in two circumstances:

1. When `condor_startd` evaluates its own state; and
2. When the `condor_starter` exits after completing fetched work.

It is possible that, even if a slot is already running another job, it could be pre-empted by a new job, which could result in a problem known as *thrashing*. In this situation, every job gets pre-empted and no job has a chance to finish. By adjusting the frequency that `condor_startd` checks for new work, this can be prevented. This can be achieved by defining the `FetchWorkDelay` configuration variable.

The `FetchWorkDelay` variable is expressed as the number of seconds to wait in between the last fetch attempt completing and attempting to fetch another job.

This example instructs `condor_startd` to wait for 300 seconds (5 minutes) between attempts to fetch jobs, unless the slot is marked as *Claimed/Idle*. In this case, `condor_startd` should attempt to fetch a job immediately:

```
FetchWorkDelay = ifThenElse(State == "Claimed" && Activity == "Idle", 0,
    300)
```

If the `FetchWorkDelay` variable is not defined, `condor_startd` will default to a 300 second (5 minute) delay between all attempts to fetch work, regardless of the state of the slot.

Example 4.8. Setting the `FetchWorkDelay` configuration variable

Using keywords to define hooks in configuration files

Hooks are defined in the configuration files by prefixing the name of the hook with a keyword. This allows a machine to have multiple sets of hooks, with each set identified by a keyword.

Each slot on a machine can define a separate keyword for the set of hooks that should be used. If a slot-specific keyword is not used, `condor_startd` will use the global keyword defined in the `STARTD_JOB_HOOK_KEYWORD` configuration variable.



Note

Slots are the logical equivalent of the physical cores on a machine. For example, a quad-core workstation would have four slots - with each slot being a dedicated allocation of memory (note however that hyperthreading will generally double the amount of slots available - a quad-core machine with hyperthreading would have eight slots).

Once a job has been hooked using `HOOK_FETCH_WORK`, the `condor_startd` daemon will use the keyword for that job to select the hooks required to execute it.

This is an example configuration file that defines hooks on a machine with four slots.

Three of the slots (slots 1-3) use the global keyword for running work from a database-driven system. These slots need to fetch work and provide a reply to the database system for each attempted claim.

The fourth slot (slot 4) uses a custom keyword to handle work fetched from a web service. It needs only to fetch work.

```
STARTD_JOB_HOOK_KEYWORD = DATABASE
```

```
SLOT4_JOB_HOOK_KEYWORD = WEB
```

```
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
```

```
WEB_HOOK_DIR = /usr/local/condor/fetch/web
```

```
WEB_HOOK_FETCH_WORK = $(WEB_HOOK_DIR)/fetch_work.php
```

Note that the keywords *DATABASE* and *WEB* are very generic terms. It is advised that you choose more specific keywords for your own installation.

Example 4.9. Using keywords when defining hooks

User Priorities and Negotiation

MRG Grid uses priorities and negotiation to allocate jobs between the machines in the pool. Every user is identified by `username@uid_domain` and is assigned a priority value. These values are assigned to the user, not the machine. This enables users to submit jobs from different machines in the same domain, or even from multiple machines in multiple domains.

Priorities are numerical values assigned to each user. The highest possible priority is 1, and the priority decreases as the number rises. There are two priority values assigned to users:

- Real User Priority (RUP), which measures the amount of resources consumed by the user.
- Effective User Priority (EUP), which determines the number of resources available to the user.

Real User Priority (RUP)

RUP measures the amount of resources consumed by the user over time. Every user begins with a RUP of 0.5 and will stabilize over time if the user consumes resources at a stable rate. For example, if a user continuously uses exactly ten resources for a long period of time, the RUP of that user will stabilize to 10.

The RUP will get better as the user decreases the amount of resources being consumed. The rate at which the RUP decays can be set in the configuration files using the `PRIORITY_HALFLIFE` setting, which measures in seconds. For example, if the `PRIORITY_HALFLIFE` is set to 86400 (1 day), and a user who's RUP is 10 removes all their jobs and consumes no further resources, the RUP would become 5 in one day, 2.5 in two days, and so on.

Effective User Priority (EUP)

EUP is used to determine how many resources a user can access. The EUP is related to the RUP by a priority factor which can be defined on a per-user basis. By default, the priority factor for all users is 1.0, and so the EUP will remain the same as the the RUP. This can be used to preferentially serve some users over others.

The number of resources that a user can access is inversely related to the EUP of each user. For example, Alice has an EUP of 5, Bob has an EUP of 10 and Charlie has an EUP of 20. In this case, Alice will be able to access twice as many resources as Bob, who can access twice as many as Charlie. However, if a user does not consume the full amount of resources they have been allocated, the remainder will be redistributed among the remaining users.

There are two settings that can affect EUP when submitting jobs:

Nice users

A *nice user* gets their RUP raised by a priority factor, which is specified in the configuration file. This results in a large EUP and subsequently a low priority for access to resources, causing the job to run as the equivalent of a background job.

Remote Users

In some situations, users from other domains may be able to submit jobs to the local pool. It may be preferable to treat local users preferentially over remote users. In this case, a *remote user* would get their RUP raised by a priority factor, which is specified in the configuration file. This results in a large EUP and subsequently a low priority for access to resources.

Pre-emption

Priorities are used to ensure that users get an appropriate allocation of resources. MRG Grid can also pre-empt jobs and reallocate them if conditions change, so that higher priority jobs are continually pushed further up the queue.

However, too many pre-emptions can lead to a condition known as *thrashing*, where a new job with a higher priority is identified every cycle. In this situation, every job gets pre-empted and no job has a chance to finish. To avoid thrashing, conditions for pre-emption can be set using the `PREEMPTION_REQUIREMENTS` setting in the configuration file. Set this variable to deny pre-emption when the current job has been running for a relatively short period of time. This limits the number of pre-emptions per resource, per time period. There is more information about the `PREEMPTION_REQUIREMENTS` setting in [Chapter 2, Configuration](#).

Negotiation

MRG Grid uses negotiation to match jobs with the resources capable of running them. The `condor_negotiator` daemon is responsible for negotiation.

Negotiation occurs in cycles. During a negotiation cycle, the `condor_negotiator` daemon performs the following actions, in this order:

1. Construct a list of all possible resources in the pool
2. Obtain a list of all job submitters in the pool
3. Sort the list of job submitters based on EUP, with the highest priority user (lowest EUP) at the top of the list, and the lowest at the bottom.
4. Continue to perform all four steps until there are either no more resources to match, or no more jobs to match.

Once the `condor_negotiator` daemon has finished the initial actions, it will list every job for each submitter, in EUP order. Since jobs can be submitted from more than one machine, there is further sorting. When the jobs all come from a single machine, they are sorted in order of job priority. Otherwise, all the jobs from a single machine are sorted before sorting the jobs from the next machine.

In order to find matches, `condor_negotiator` will perform the following tasks for each machine in the pool that can execute jobs:

1. If `machine.requirements` is false or `job.requirements` is false, ignore the machine
2. If the machine is in the *Claimed* state, but not running a job, ignore the machine
3. If the machine is not running a job, add it to the potential match list with a reason of *No Preemption*
4. If the machine is running a job:
 - a. If the `machine.RANK` on the submitted job is higher than that of the running job, add this machine to the potential match list with a reason of *Rank*
 - b. If the EUP of the submitted job is better than the EUP of the running job, `PREEMPTION_REQUIREMENTS` is true, and the `machine.RANK` on the submitted job is higher than the running job, add this machine to the potential match list with a reason of *Priority*

The potential match list is sorted by:

1. `NEGOTIATOR_PRE_JOB_RANK`

-
2. *job.RANK*
 3. *NEGOTIATOR_POST_JOB_RANK*
 4. Reason for claim
 - *No Preemption*
 - *Rank*
 - *Priority*
 5. *PREEMPTION_RANK*

The job is then assigned to the top machine on the potential match list. That machine is then removed from the list of resources available in this negotiation cycle and the daemon goes on to find a match for the next job.

Cluster Considerations

If a cluster has multiple jobs and one of them cannot be matched, no other jobs in that cluster will be returned during the current negotiation cycle. This is based on an assumption that all the jobs in a cluster will be similar. The configuration variable **NEGOTIATE_ALL_JOBS_IN_CLUSTER** can be used to disable this behaviour. The definition of what makes up a cluster can be modified by use of the **SIGNIFICANT_ATTRIBUTES** setting.

Group Accounting

MRG Grid keeps a running tally of resource use. This accounting information is used to calculate priorities for the scheduling algorithms. Accounting is done on a per-user basis by default, but can also be on a per-group basis. When done on a per-group basis, any jobs submitted by the same group will be treated with the same priority.

When a job is submitted, the user can include an attribute that defines the accounting group. For example, the following line in a job's submit description file indicates that the job is part of the *group_physics* accounting group:

```
+AccountingGroup = "group_physics"
```

Example 5.1. Submit description file entry when using accounting groups

The value for the *AccountingGroup* attribute is a string. It must be enclosed in double quotation marks and can contain a maximum of 40 characters. The name should not be qualified with a domain, as parts of the system will add the **\$(UID_DOMAIN)** to the string. For example, the statistics for this accounting group might be displayed as follows:

User Name	EUP
-----	-----
group_physics@example.com	0.50
mcurie@example.com	23.11
pvonlenard@example.com	111.13
...	

Example 5.2. Accounting group statistics, showing the appending of the fully qualified domain

Condor normally removes entities automatically when they are no longer relevant, however administrators can also remove accounting groups manually, using the `-delete` option with the `condor_userprio` daemon. This action will only work if all jobs have already been removed from the accounting group, and the group is identified by its fully-qualified name. For example:

```
$ condor_userprio -delete group_physics@example.com
```

Example 5.3. Manually removing accounting groups

Group Quotas

In some cases, priorities based on each individual user might not be effective. *Group quotas* affect the negotiation for available resources within the pool. This may be the case when different groups own different amounts of resources, and the groups choose to combine their resources to form a pool. For example:

The physics department owns twenty workstations, and the chemistry department owns ten workstations. They have combined their resources to form a pool of thirty similar machines. The physics department wants priority on any twenty of the workstations. Likewise, the chemistry department wants priority on any ten workstations.

By creating group quotas, users are allocated not to specific machines, but to numbers of machines (a quota). Given thirty similar machines, group quotas allow the users within the physics group to have preference on up to twenty of the machines within the pool, and the machines can be any of the machines that are currently available.

Example 5.4. An effective use of group quotas

In order to set group quotas, the group must be identified in the job's submit description file, using the *AccountingGroup* attribute. Members of a group quota are called *group users*. When specifying a group user, you will need to include the name of the group, as well the username, using the following syntax:

```
+AccountingGroup = "group.user"
```

For example, if the user `mcurie` of the `group_physics` group was submitting a job in a pool that implements group quotas, the submit description file would be:

```
+AccountingGroup = "group_physics.mcurie"
```

Example 5.5. Submit description file entry when using group quotas

Group names are not case-sensitive and do not require the `group_` prefix. However, in order to avoid conflicts, group names must be different to user names. Adding the `group_` prefix to group names ensures against conflicts.

Quotas are configured in terms of slots per group. The combined quotas for all groups must be equal to or less than the amount of available slots in the pool. Any slots that are not allocated as part of a group quota are allocated to the *none* group. The *none* group contains only those users who do not submit jobs as part of a group.

Changes caused by group quotas to accounting and negotiation

When using group quotas, some changes occur in how accounting and negotiation are processed.

For jobs submitted by group users, accounting is performed per group user, rather than per group or individual user.

Negotiation is performed differently when group quotas are used. Instead of negotiating in the order described in [Negotiation](#), the `condor_negotiator` daemon will create a list of all jobs belonging to defined groups before it lists those jobs submitted by individual submitters. If there is more than one group in the negotiation cycle, the daemon will negotiate for the group using the smallest percentage of resources first, and the highest percentage last. However, the same algorithm still applies to individual submitters.

Managing configuration for group quotas

Configuring a pool can be slightly different when using group quotas. Each group can be assigned an initial value for user priority with the `GROUP_PRIO_FACTOR_` setting. Additionally, if a group is currently allocated the entire quota of machines, and a group user has a submitted job that is not running, the `GROUP_AUTOREGROUP_` setting, if true, will allow the job to be considered again within the same negotiation cycle, along with the individual users jobs.

```
• GROUP_NAMES = group_physics, group_chemistry
• GROUP_QUOTA_group_physics = 20
• GROUP_QUOTA_group_chemistry = 10
• GROUP_PRIO_FACTOR_group_physics = 1.0
• GROUP_PRIO_FACTOR_group_chemistry = 3.0
• GROUP_AUTOREGROUP_group_physics = FALSE
• GROUP_AUTOREGROUP_group_chemistry = TRUE
```

In this example, the physics group can access 20 machines and the chemistry group can access ten machines. The initial priority factor for users within the groups are 1.0 for the physics group and 3.0 for the chemistry group. The `GROUP_AUTOREGROUP_` settings indicate that the physics group will never be able to access more than 20 machines, while the chemistry group could potentially get more than ten machines.

Example 5.6. Example configuration for group quotas

Job Priority

In addition to user priorities, it is also possible to specify job priorities to control the order of job execution. Jobs can be assigned a priority level, of any integer, through the use of the `condor_prio` command. Jobs with a higher number will run with a higher priority. Job priority works only on a per user basis. It is effective when used by a single user to order their own jobs, but will not impact the order in which they run with other jobs in the pool.

1. To find out what jobs are currently running, use the `condor_q` with the name of the user to query:

```
$ condor_q user
```

```
-- 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   user          4/11 15:06     0+00:00:00 I  0  0.3 hello  
  
1 jobs; 1 idle, 0 running, 0 held
```

2. Job priority can be any integer. The default priority is 0. To change the priority use the **condor_prio** with the desired priority:

```
$ condor_prio -p -15 126.0
```

3. To check that the changes have been made, use the **condor_q** command again:

```
$ condor_q user  
-- 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   user          4/11 15:06     0+00:00:00 I -15 0.3 hello  
  
1 jobs; 1 idle, 0 running, 0 held
```

ClassAds

Job submission is simplified through the use of *ClassAds*. ClassAds are used to advertise the attributes of individual jobs and each slot on a machine. MRG Grid then uses the ClassAds to match jobs to slots.



Note

Slots are the logical equivalent of the physical cores on a machine. For example, a quad-core workstation would have four slots - with each slot being a dedicated allocation of memory (note however that hyperthreading will generally double the amount of slots available - a quad-core machine with hyperthreading would have eight slots).

ClassAds for slots advertise information such as:

- available RAM
- CPU type and speed
- virtual memory size
- current load average

Slots also advertise information about the conditions under which it is willing to run a job, and what type of job it would prefer. Additionally, machines can specify which jobs they would prefer to run. All this information is held by the ClassAd.

ClassAds for jobs advertise the type of machine they need to execute the job. For example, a job may require a minimum of 128MB of RAM, but would ideally like 512MB. This information is listed in the jobs ClassAd and slots that meet those requirements will be ranked for matching.

MRG Grid continuously reads all the ClassAds, ranking and matching jobs and slots. All requirements for both sets of ClassAds must be fulfilled before a match is made. ClassAds are generated automatically by the `condor_submit` daemon, but can also be manually constructed and edited.

This example uses the `condor_status` command to view ClassAds information from the machines available in the pool.

```
$ condor_status
```

Name	Arch	OpSys	State	Activity	LoadAv	Mem	ActvtyTime
adriana.cs	x86_64	LINUX	Claimed	Busy	1.000	64	0+01:10:00
alfred.cs.	x86_64	LINUX	Claimed	Busy	1.000	64	0+00:40:00
amul.cs.wi	x86_64	LINUX	Owner	Idle	1.000	128	0+06:20:04
anfrom.cs.	x86_64	LINUX	Claimed	Busy	1.000	32	0+05:16:22
anthrax.cs	x86_64	LINUX	Claimed	Busy	0.285	64	0+00:00:00
astro.cs.w	x86_64	LINUX	Claimed	Busy	0.949	64	0+05:30:00
aura.cs.wi	x86_64	LINUX	Owner	Idle	1.043	128	0+14:40:15

Example 6.1. Using `condor_status` to view ClassAds

The `condor_status` command has options that can be used to view the data in different ways. The most common options are:

`condor_status -available`

Shows only those machines that are currently available to run jobs.

`condor_status -run`

Shows only those machines that are currently running jobs.

`condor_status -l`

Lists the ClassAds for all machines in the pool.



Note

Use `$ man condor_status` for a complete list of options.

Constraints and preferences

Constraints and preferences for jobs are specified in the submit description file using **`requirements`** and **`rank`** expressions. For machines, this information is determined by the configuration.

The **`rank`** expression is used by a job to specify which requirements to use to rank potential machine matches.

This example uses the **`rank`** expression to specify preferences a job has for a machine.

A job ClassAd might contain the following expressions:

```
Requirements = Arch=="x86_64" && OpSys == "LINUX"
Rank          = TARGET.Memory + TARGET.Mips
```

In this case, the job requires a computer running a 64 bit Linux operating system. Among all such computers, the job prefers those with large physical memories and high MIPS (Millions of Instructions Per Second) ratings.

Example 6.2. Using the **`rank`** expression to set constraints and preferences for jobs

Any desired attribute can be specified for the **`rank`** expression. The `condor_negotiator` daemon will satisfy the required attributes first, then deliver the best resource available by matching the rank expression.

A machine may also specify constraints and preferences for the jobs that it will run.

This example using the machine configuration to set constraints and preferences a machine has for a job

A machine's configuration might contain the following:

```
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
```

This machine will always run a job submitted by members of the *ResearchGroup* but will never run jobs owned by users *rival* and *riffraff*. Jobs submitted by *Friends* are preferred to foreign jobs, and jobs submitted by the *ResearchGroup* are preferred to jobs submitted by *Friends*.

Example 6.3. Using machine configuration to set constraints and preferences

Querying ClassAd expressions

ClassAds can be queried from the shell prompt with the `condor_status` and `condor_q` tools. Some common examples are shown here:



Note

Use `$ man condor_status` and `$ man condor_q` for a complete list of options.

This example finds all computers that have more than 100MB of memory and their keyboard idle for longer than 20 minutes

```
$ condor_status -constraint 'KeyboardIdle > 20*60 && Memory > 100'
```

Name	Arch	OpSys	State	Activity	LoadAv	Mem	ActvtyTime
amul.cs.wi	x86_64	LINUX	Claimed	Busy	1.000	128	0+03:45:01
aura.cs.wi	x86_64	LINUX	Claimed	Busy	1.000	128	0+00:15:01
balder.cs.	x86_64	LINUX	Claimed	Busy	1.000	1024	0+01:05:00
beatrice.c	x86_64	LINUX	Claimed	Busy	1.000	128	0+01:30:02

[output truncated]

Machines	Owner	Claimed	Unclaimed	Matched	Preempting		
x86_64/LINUX		3	0	3	0	0	0
x86_64/LINUX		21	0	21	0	0	0
x86_64/LINUX		3	0	3	0	0	0
x86_64/LINUX		1	0	0	1	0	0
x86_64/LINUX		1	0	1	0	0	0
Total	29	0	28	1	0	0	

Example 6.4. Using the `condor_status` command with the `-constraint` option

This example uses a regular expression and a ClassAd function to list specific information.

A file called **ad** contains ClassAd information:

```
$ cat ad
MyType = "Generic"
FauxType = "DBMS"
Name = "random-test"
Machine = "f05.cs.wisc.edu"
MyAddress = "<128.105.149.105:34000>"
DaemonStartTime = 1153192799
UpdateSequenceNumber = 1
```

The **condor_advertise** daemon is used to insert the generic ClassAd information into the file:

```
$ condor_advertise UPDATE_AD_GENERIC ad
```

You can now use **condor_status** to constrain the search with a regular expression containing a ClassAd function:

```
$ condor_status -any -constraint 'FauxType=="DBMS" && regexp("random.*",
  Name, "i")'
```

MyType	TargetType	Name
Generic	None	random-test

Job queues can also be queried in the same way.

Example 6.5. Using a regex and a ClassAd function to list information

6.1. Writing ClassAd expressions

The primary purpose of a ClassAd is to make matches, where the possible matches contain constraints. To achieve this, the ClassAd mechanism will continuously carry out expression evaluations, where two ClassAds test each other for a potential match. This is performed by the **condor_negotiator** daemon. This section examines the semantics of evaluating constraints.

A ClassAd contains a set of *attributes*, which are unique names associated with expressions.

```
MyType = "Machine"
TargetType = "Job"
Machine = "froth.cs.wisc.edu"
Arch = "x86_64"
OpSys = "LINUX"
Disk = 35882
Memory = 128
KeyboardIdle = 173
LoadAvg = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60
```

Example 6.6. A typical ClassAd

ClassAd expressions are formed by *literals*, *attributes* and other sub-expressions combined with *operators* and *functions*. ClassAd expressions are not statically checked. For this reason, the expressions **UNDEFINED** and **ERROR** are used to identify expressions that contain names of attributes that have no associated value or that attempt to use values in a way that is inconsistent with their types.

Literals

Literals represent constant values in expressions. An expression that contains a literal will always evaluate to the value that the literal represents. The different types of literals are:

Integer

One or more digits (0-9). Additionally, the keyword *TRUE* represents *1* and *FALSE* represents *0*

Real

Two sequences of continuous digits separated by a *.* character

String

Zero or more characters enclosed within *"* characters. A ** character can be used as an escape character

Undefined

The keyword **UNDEFINED** represents an attribute that has not been given a value.

Error

The keyword **ERROR** represents an attribute with a value that is inconsistent with its type, or badly constructed.

Attributes

Every expression must have a name and a value, together the pair is referred to as an *attribute*. An attribute can be referred to in other expressions by its name.

Attribute names are sequences of letters, numbers and underscores. They can not start with a number. All characters in the name are significant, but they are not case sensitive.

A reference to an attribute must consist of the name of the attribute being referred to. References can also contain an optional *scope resolution prefix* of either **MY.** or **TARGET.**

The expression evaluation is carried out in the context of two ClassAds, creating a potential for ambiguities in the name space. The following rules define the semantics of attribute references made by *ClassAd A* which is being evaluated in relation to *ClassAd B*:

If the reference contains a scope resolution prefix:

- If the prefix is **MY.** the attribute will be looked up in *ClassAd A*. If the attribute exists in *ClassAd A*, the value of the reference becomes the value of the expression bound to the attribute name. If the attribute does not exist in *ClassAd A*, the value of the reference becomes **UNDEFINED**
- If the prefix is **TARGET.** the attribute is looked up in *ClassAd B*. If the attribute exists in *ClassAd B* the value of the reference becomes the value of the expression bound to the attribute name. If the attribute does not exist in *ClassAd B*, the value of the reference becomes **UNDEFINED**

If the reference does not contain a scope resolution prefix:

- If the attribute is defined in *ClassAd A* the value of the reference is the value of the expression bound to the attribute name in *ClassAd A*
- If the attribute is defined in *ClassAd B* the value of the reference is the value of the expression bound to the attribute name in *ClassAd B*
- If the attribute is defined in the ClassAd environment, the value from the environment is returned. This is a special environment, not the standard Unix environment. Currently, the only attribute of the environment is *CurrentTime*, which evaluates to the integer value returned by the `system call time(2)`
- If the attribute is not defined in any of the above locations, the value of the reference becomes **UNDEFINED**

If the reference refers to an expression that is itself in the process of being evaluated, it will cause a circular dependency. In this case, the value of the reference becomes **ERROR**

Operators

The unary negation operator of `-` takes the highest precedence in a string. In order, operators take the following precedence:

1. `-` (unary negation)
2. `*` and `/`
3. `+` (addition) and `-` (subtraction)
4. `<` `<=` `>=` and `>`
5. `==` `!=` `==?` and `!=?`
6. `&&`
7. `||`

The different types of operators are:

Arithmetic operators

The operators `*` `/` `+` and `-` operate arithmetically on integers and real literals

Arithmetic is carried out in the same type as both operands. If one operand is an integer and the other real, the type will be promoted from integer to real

Operators are strict with respect to both **UNDEFINED** and **ERROR**

If one or both of the operands are not numerical, the value of the operation is **ERROR**

Comparison operators

The comparison operators `==` `!=` `<=` `<` `>=` and `>` operate on integers, reals and strings

The operators `==?` and `!=?` behave similarly to `==` and `!=`, but are not strict. Semantically, `==?` tests if its operands have the same type and the same value. For example, `10 == UNDEFINED` and `UNDEFINED == UNDEFINED` both evaluate to **UNDEFINED**, but `10 ==? UNDEFINED` will evaluate to **FALSE** and `UNDEFINED ==? UNDEFINED` will evaluate to **TRUE**. The `!=?` operator tests for not identical conditions

String comparisons are case insensitive for most operators. The only exceptions are the operators `==?` and `!=?` which perform case sensitive comparisons when both sides are strings

Comparisons are carried out in the same type as both operands. If one operand is an integer and the other real, the type will be promoted from integer to real

Strings can not be converted to any other type, so comparing a string and an integer or a string and a real results in **ERROR**

The operators `==` `!=` `<=` `<` and `>=` `>` are strict with respect to both **UNDEFINED** and **ERROR**

Logical operators

The logical operators `&&` and `||` operate on integers and reals. The zero value of these types are considered **FALSE** and non-zero values **TRUE**

Logical operators are not strict, and exploit the "don't care" properties of the operators to eliminate **UNDEFINED** and **ERROR** values when possible. For example, **UNDEFINED** `&&` **FALSE** evaluates to **FALSE**, but **UNDEFINED** `||` **FALSE** evaluates to **UNDEFINED**

Any string operand is equivalent to an **ERROR** operand for a logical operator. For example **TRUE** `&&` **"string"** evaluates to **ERROR**

Pre-defined functions

ClassAd expressions can use predefined functions. Function names are not case sensitive. Function calls can also be nested or recursive.

This is a complete list of predefined functions. The format of each function is:

ReturnType

```
FunctionName(ParameterType1 parameter1, ParameterType2 parameter2, ...)
```

The possible types are as listed in [Literals](#). If the function can be any of these literal types, it is described as **AnyType**. Where the type is **Integer**, but only returns the value 1 or 0 (*True* or *False*), it is described as **Boolean**. Optional parameters are given in square brackets.

AnyType `ifThenElse(AnyType IfExpr, AnyType ThenExpr, AnyType ElseExpr)`

A conditional expression.

When *IfExpr* evaluates to **true**, return the value as given by *ThenExpr*

When **false**, return the value as given by *ElseExpr*

When **UNDEFINED**, return the value **UNDEFINED**

When **ERROR**, return the value **ERROR**

When *IfExpr* evaluates to 0.0, return the value as given by *ElseExpr*

When *IfExpr* evaluates to a non-0.0 or Real value, return the value as given by *ThenExpr*

When *IfExpr* evaluates to give a value of type **string**, return the value **ERROR**

Expressions are only evaluated as defined

If a number of arguments other than three are given, the function will return **ERROR**

Boolean `isUndefined(AnyType Expr)`

Returns *True* if *Expr* evaluates to **UNDEFINED**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean `isError(AnyType Expr)`

Returns *True*, if *Expr* evaluates to **ERROR**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean `isString(AnyType Expr)`

Returns *True* if *Expr* gives a value of type **string**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean `isInteger(AnyType Expr)`

Returns *True*, if *Expr* gives a value of type **Integer**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean `isReal(AnyType Expr)`

Returns *True* if *Expr* gives a value of type **Real**. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Boolean `isBoolean(AnyType Expr)`

Returns *True*, if *Expr* returns an integer value of *1* or *0*. Otherwise, returns *False*

If a number of arguments other than one is given, the function will return **ERROR**

Integer `int(AnyType Expr)`

Returns the integer value as defined by *Expr*

Where the type of the evaluated *Expr* is **Real** the value is rounded down 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. If the result is not an integer, **ERROR** is returned

Where the evaluated *Expr* is **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

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. If the result is not **real** **ERROR** is returned

Where the evaluated *Expr* is **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

String `string(AnyType Expr)`

Returns the string that results from the evaluation of *Expr*

A non-**string** value will be converted to a **string**

Where the evaluated *Expr* is **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

Integer floor(**AnyType** *Expr*)

When the type of the evaluated *Expr* is **Integer**, returns the integer that results from the evaluation of *Expr*

When the type of the evaluated *Expr* is anything other than **Integer**, function **real(Expr)** is called. Its return value is then used to return the largest integer that is not higher than the returned value

Where the **Real(Expr)** returns **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

Integer ceiling(**AnyType** *Expr*)

When the type of the evaluated *Expr* is **Integer**, returns the integer that results from the evaluation of *Expr*

When the type of the evaluated *Expr* is anything other than **Integer**, function **real(Expr)** is called. Its return value is then used to return the smallest integer that is not less than the returned value

Where the **Real(Expr)** returns **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

Integer round(**AnyType** *Expr*)

When the type of the evaluated *Expr* is **Integer**, returns the integer that results from the evaluation of *Expr*

When the type of the evaluated *Expr* is anything other than **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 values. In this case, the even valued integer is returned

Where the **Real(Expr)** returns **ERROR** or **UNDEFINED**, or the integer does not fit into 32 bits **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

Integer random([**AnyType** *Expr*])

When the type of the optional argument *Expr* evaluates to **Integer** or **Real**, the return value is the integer or real *r* randomly chosen from the interval $0 \leq r < x$

With no argument, the return value is chosen with *random(1.0)*

In all other cases, the function will return **ERROR**

If a number of arguments other than one is given, the function will return **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)**

If any argument evaluates to **ERROR** or **UNDEFINED**, **ERROR** is returned

String substr(**String** *s*, **Integer** *offset* [, **Integer** *length*])

Returns the substring *s*, from the position indicated by *offset*, with optional *length* characters

The first character within *s* is at offset 0. If the *length* argument is not used, the substring extends to the end of the string

If *offset* is negative, the value of **length** - *offset* is used for *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

If a number of arguments other than either two or three is given, the function will return **ERROR**

Integer strcmp(**AnyType** *Expr1*, **AnyType** *Expr2*)

Both arguments are converted to type **String** by **string(Expr)**

The return value is an integer that will be less than 0 if *Expr1* is less than *Expr2*

The return value will be equal to 0 if *Expr1* is equal to *Expr2*

The return value will be greater than 0 if *Expr1* is greater than *Expr2*

Case is significant in the comparison. Where either argument evaluates to **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than two is given, the function will return **ERROR**

Integer stricmp(**AnyType** *Expr1*, **AnyType** *Expr2*)

This function is the same as the **strcmp** function, except that letter case is not significant

String toUpper(**AnyType** *Expr*)

The argument is converted to type **String** by the **string(Expr)**

The return value is a string, with all lower case letters converted to upper case

If the argument evaluates to **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

String toLower(**AnyType** *Expr*)

The argument is converted to type **String** by the **string(Expr)**

The return value is a string, with all upper case letters converted to lower case

If the argument evaluates to **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

Integer Size(**AnyType** *Expr*)

Returns the number of characters in the string, after calling the **string(Expr)** function

If the argument evaluates to **ERROR** or **UNDEFINED**, **ERROR** is returned

If a number of arguments other than one is given, the function will return **ERROR**

The following functions contain string lists. *String delimiters* are used to define how the string list should be read. The characters in the string delimiter define the characters used to separate the elements within the string list.

This example uses the **stringListSize** function to demonstrate how a string delimiter of ", /" (a comma, followed by a space character, followed by a pipe) operates.

The function is given as follows:

```
StringListSize("1,2 3|4&5", ", |")
```

Firstly, the string is broken up according to the first delimiter - the comma character - resulting in the following two elements:

```
"1" and "2 3|4&5"
```

Now perform the same process, using the second delimiter - the space character - resulting in three elements:

```
"1", "2" and "3|4&5"
```

Finally, apply the third delimiter - the pipe character - resulting in a total of four elements:

```
"1", "2", "3" and "4&5"
```

Note that because the & character is not defined as a delimiter, the final group ("4&5") is considered only one element

Example 6.7. Using a string delimiter of ", /" with string lists



Note

The *string delimiter* is optional in the following functions. If no *string delimiter* is defined, the default string delimiter of " ," (a space character, followed by a comma) is used.

Integer stringListSize(**String** *list* [, **String** *delimiter*])

Returns the number of elements in the **String** *list*, as delimited by the **String** *delimiter*

If one or both of the arguments is not a string, returns **ERROR**

If a number of arguments other than one is given, the function will return **ERROR**

Integer stringListSum(**String** *list* [, **String** *delimiter*]) OR **Real** stringListSum(**String** *list* [, **String** *delimiter*])

Returns the sum of all items in the **String** *list*, as delimited by the **String** *delimiter*

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, the return value is real

If any item is not either an integer or real value, the return value is **ERROR**

Real stringListAve(**String** *list* [, **String** *delimiter*])

Sums and returns the real-valued average of all items in the **String** *list*, as delimited by the **String** *delimiter*

If any item is neither an integer nor a real value, the return value is **ERROR**

A list containing no items returns the value *0.0*

Integer stringListMin(**String** *list* [, **String** *delimiter*]) *OR* **Real** stringListMin(**String** *list* [, **String** *delimiter*])

Returns the minimum value from all items in the **String** *list*, as delimited by the **String** *delimiter*

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, the return value is a real

If any item is neither an integer nor a real value, the return value is **ERROR**

A list containing no items returns the value *UNDEFINED*

Integer stringListMax(**String** *list* [, **String** *delimiter*]) *OR* **Real** stringListMax(**String** *list* [, **String** *delimiter*])

Returns the maximum value from all items in the **String** *list*, as delimited by the **String** *delimiter*

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, the return value is a real

If any item is neither an integer nor a real value, the return value is **ERROR**

A list containing no items 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 **String** *delimiter*

Returns *FALSE* if item *x* is not in the **String** *list*

Comparison is performed with the **strcmp()** function

If the arguments are not strings, the return value is **ERROR**

Boolean stringListIMember(**String** *x*, **String** *list* [, **String** *delimiter*])

This function is the same as the **stringListMember** function, except that the comparison is done with the **stricmp()** function, so letter case is not significant

The following functions contain a regular expression (*regex*) and an options argument. The options argument is a string of special characters that modify the use of the regex. The only accepted options are:

Option	Description
<i>I</i> or <i>i</i>	Ignore letter case

Option	Description
<i>M</i> or <i>m</i>	Modifies the interpretation of the carat (^) and dollar sign (\$) characters, so that ^ matches the start of a string, as well as after each new line character and \$ matches before a new line character
<i>S</i> or <i>s</i>	Modifies the interpretation of the period (.) character to match any character, including the new line character.
<i>X</i> or <i>x</i>	Ignore white space and comments within the pattern. A comment is defined by starting with the # character, and continuing until the new line character.

Table 6.1. Options for use in functions

**Note**

For a complete list of regular expressions visit the [PCRE Library](#)¹

Boolean `regexp(String pattern, String target [, String options])`

Returns *TRUE* if the **String** *target* is a regular expression as described by *pattern*. Otherwise returns *FALSE*

If any argument is not a **String**, or if *pattern* does not describe a valid regular expression, returns **ERROR**

String `regexps(String pattern, String target, String substitute, [String options])`

The regular expression *pattern* is applied to *target*. If the **String** *target* is a regular expression as described by *pattern*, the **String** *substitute* is returned, with backslash expansion performed

If any argument is not a **String** returns **ERROR**

Boolean `stringListRegexpMember(String pattern, String list [, String delimiter] [, String options])`

Returns *TRUE* if any of the strings within the list is a regular expression as described by *pattern*. Otherwise returns *FALSE*

If any argument is not a **String**, or if *pattern* does not describe a valid regular expression, returns **ERROR**

To include the optional fourth argument *options*, a third argument of **String** *delimiter* is required. If a specific *delimiter* is not specified, the default value of " ," (a space character followed by a comma) will be used

Integer `time()`

Returns the current Unix epoch time, which is equal to the ClassAd attribute **CurrentTime**. This is the time, in seconds, since midnight on January 1, 1970

string interval(**Integer** *seconds*)

Uses seconds to return a string in the form of *days+hh:mm:ss* representing an interval of time.
Leading values of zero are omitted from the string

Policy Configuration

Machines in a pool can be configured through the `condor_startd` daemon to implement policies that perform actions such as:

- Start a remote job
- Suspend a job
- Resume a job
- Create a checkpoint and vacate a job
- Kill a job without creating a checkpoint

Policy configuration is the at the heart of the balancing act between the needs and wishes of machine owners and job submitters. This section will outline how to adjust the policy configuration for machines in your pool.



Note

If you are configuring the policy for a machine with multiple cores, and therefore multiple slots, each slot will have its own individual policy expressions. In this case, the word *machine* refers to a single slot, not to the machine as a whole.

This chapter assumes you know and understand ClassAd expressions. Ensure that you have read [Chapter 6, ClassAds](#) before you begin.

7.1. Machine states and transitioning

Every machine is assigned a *state*, which changes as machines become available to run jobs. The six possible states are:

Owner

The machine is not available to run jobs. This state normally occurs when the machine is being used by the owner. Additionally, machines begin in this state when they are first turned on

Unclaimed

The machine is available to run jobs, but is not currently doing so

Matched

The machine has been matched to a job by the negotiator, but the job has not claimed the machine

Claimed

The machine has been claimed by a job. The job may be currently executing, or waiting to begin execution

Preempting

The machine was claimed, but is now being pre-empted. This state is used to evict a running job from a machine, so that a new job can be started. This can happen for one of the following reasons:

- The owner has started using the machine

- Jobs with a higher priority are waiting to run
- 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 return or to be matched with a job. This state is only entered if the machine is specifically configured for backfill jobs

The following diagram demonstrates the machine states (with the exception of *Backfill* which is further described below) and the possible transitions between them.

Possible transitions between machine states

Owner to Unclaimed

This transition occurs when the machine becomes available to run a job. This occurs when the **START** expression evaluates to *TRUE*.

Unclaimed to Owner

This transition occurs when the machine is in use and therefore not available to run jobs. This occurs when the **START** expression evaluates to *FALSE*.

Unclaimed to Matched

This transition occurs when the resource is matched with a job.

Unclaimed to Claimed

This transition occurs if **condor_schedd** initiates the claiming procedure before the **condor_startd** receives notification of the match from the **condor_negotiator**.

Unclaimed to Backfill

This transition occurs only if the machine is configured to run backfill computations and the **START_BACKFILL** expression evaluates to *TRUE*.

Matched to Owner

This transition occurs if:

- the machine is no longer available to run jobs. This happens when the **START** expression evaluates to *FALSE*.
- the **MATCH_TIMEOUT** timer expires. This occurs when a machine has been matched but not claimed. The machine will eventually give up on the match and become available for a new match.
- **condor_schedd** has attempted to claim the machine but encountered an error.
- **condor_startd** receives a **condor_vacate** command while it is in the *Matched* state.

Matched to Claimed

This transition occurs when the machine is successfully claimed and the job is running.

Claimed to Pre-empting

From the *Claimed* state, the only possible destination is the *Pre-empting* state. This transition can be caused when:

- The job that has claimed the machine has completed and releases the machine
- The resource is in use. In this case, the **PREEMPT** expression evaluates to *TRUE*
- **condor_startd** receives a **condor_vacate** command.
- **condor_startd** is instructed to shut down.
- The machine is matched to a job with a higher priority than the currently running job.

Pre-empting to Claimed

This transition occurs when the resource is matched to a job with a better priority.

Pre-empting to Owner

This transition occurs if:

- the **PREEMPT** expression evaluated to *TRUE* while the machine was in the *Claimed* state
- **condor_startd** receives a **condor_vacate** command
- if the **START** expression evaluates to *FALSE* and the job it was running had finished being evicted when it entered the *Pre-empting* state.

Backfill to Owner

This transition occurs if:

- The **EVICT_BACKFILL** expression evaluates to *TRUE*
- **condor_startd** receives a **condor_vacate** command
- **condor_startd** is instructed to shut down.

Backfill to Matched

This transition occurs when a resource running a backfill computation is matched with a **condor_schedd** that wants to run a job.

Backfill to Claimed

This transition occurs if the **condor_schedd** completes the claiming protocol before the **condor_startd** receives notification of the match from the **condor_negotiator**.

Machine Activities

While a machine is in a particular state, it will also be performing an *activity*. The possible activities are:

- Idle
- Benchmarking
- Busy
- Suspended
- Retiring
- Vacating
- Killing

Each of these activities has a slightly different meaning, depending on which state they occur in. This list explains each of the possible activities for a machine in different states:

- *Owner*
 - *Idle*: This is the only possible activity for a machine in the *Owner* state. It indicates that the machine is not currently performing any work for MRG Grid, even though it may be working on other unrelated tasks.
- *Unclaimed*
 - *Idle*: This is the normal activity for machines in the *Unclaimed* state. The machine is available to run MRG Grid tasks, but is not currently doing so.
 - *Benchmarking*: This activity only occurs in the *Unclaimed* state. It indicates that benchmarks are being run to determine the speed of the machine. How often this activity occurs can be adjusted by changing the *RunBenchmarks* configuration variable.
- *Matched*
 - *Idle*: Although the machine is matched, it is still considered *Idle*, as it is not currently running a job.
- *Claimed*
 - *Idle*: The machine has been claimed, but the **condor_starter** daemon, and therefore the job, has not yet been started. The machine will briefly return to this state when the job finishes.
 - *Busy*: The **condor_starter** daemon has started and the job is running.
 - *Suspended*: The job has been suspended. The claim is still valid, but the job is not making any progress and MRG Grid is not currently generating a load on the machine.
 - *Retiring*: When an active claim is about to be pre-empted, it enters retirement while it waits for the current job to finish. The *MaxJobRetirementTime* configuration variable determines how long to wait. Once the job finishes or the retirement time expires, the *Preempting* state is entered.
- *Preempting*
 - *Vacating*: The job that was running is checkpointing, so it can exit gracefully.
 - *Killing*: The machine has requested the currently running job to exit immediately, without checkpointing.
- *Backfill*
 - *Idle*: The machine is ready to run a backfill job, but it has not yet started the backfill manager.
 - *Busy*: The machine is performing a backfill computation.
 - *Killing*: The machine is killing a backfill job to either return the resource to the owner, or to make room for a MRG Grid job.

7.2. The `condor_startd` daemon

This section discusses the `condor_startd` daemon. This daemon evaluates a number of expressions in order to determine when to transition between states and activities. The most important expressions are explained here.

The `condor_startd` daemon represents the machine or slot on which it is running. This daemon is responsible for publishing characteristics about the machine in the machine's ClassAd. To see the values for the attributes, run `condor_status -l hostname` from the shell prompt. On a machine with more than one slot, the `condor_startd` will regard the machine as separate slots, each with its own name and ClassAd.

Normally, the `condor_negotiator` evaluates expressions in the machine ClassAd against job ClassAds to see if there is a match. By locally evaluating an expression, the machine only evaluates the expression against its own ClassAd. If the expression references parameters that can only be found in another ClassAd, then the expression can not be locally evaluated. In this case, the expression will usually evaluate locally to `UNDEFINED`.

The **START** expression

The most important expression to the `condor_startd` daemon is the **START** expression. This expression describes the conditions that must be met for a machine to run a job. This expression can reference attributes in the machine 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 machine locally evaluates the `IsOwner` expression to determine if it is capable of running jobs. The default `IsOwner` expression is a function of the **START** expression, so that `START =? = FALSE`. Any job ClassAd attributes appearing in the **START** expression, and subsequently in the `IsOwner` expression, are undefined in this context, and may lead to unexpected behavior. If the **START** expression is modified to reference job ClassAd attributes, the `IsOwner` expression should also be modified to reference only machine ClassAd attributes.

The **REQUIREMENTS** expression

The **REQUIREMENTS** expression is used for matching machines with jobs. When a machine is 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.

The **RANK** expression

A machine can be configured to prefer certain jobs over others, through the use of the **RANK** expression in the 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 to configure a machine so that it prefers to run jobs from the owner of that machine. Similarly, it is often used for a group of machines to prefer jobs submitted by the owners of those machines.

This example demonstrates a simple application of the **RANK** expression

In this example there is a small research group consisting of four machines and four owners:

- The machine called *tenorsax* is owned by the user *coltrane*
- The machine called *piano* is owned by the user *tyner*
- The machine called *bass* is owned by the user *garrison*
- The machine called *drums* is owned by the user *jones*

To implement a policy that gives priority to the machines in this research group, set the **RANK** expression to reference the **Owner** attribute, where it matches one of the people in the group:

```
RANK = Owner == "coltrane" || Owner == "tyner" \
|| Owner == "garrison" || Owner == "jones"
```

Boolean expressions evaluate to either *1* or *0* (*TRUE* or *FALSE*). In this case, if the remote job is owned by one of the preferred users, the **RANK** expression will evaluate to *1*. If the remote job is owned by any other user, it would evaluate to *0*. The **RANK** expression is evaluated as a floating point number, so it will prefer the group users because it evaluates to a higher number.

Example 7.1. A simple application of the **RANK** expression in the machine ClassAd

This example demonstrates a more complex application of the **RANK** expression. It uses the same basic scenario as [Example 7.1, “A simple application of the **RANK** expression in the machine ClassAd”](#), but gives the owner a higher priority on their own machine.

This example is on the machine called *bass*, which is owned by the user *garrison*. The following entry would need to be included in the local configuration file called **bass.local** on that machine:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
+ ((Owner == "garrison") * 10) + (Owner == "jones")
```

The parentheses in this expression are essential, because the *+* operator has higher default precedence than *==*. Using *+* instead of */ /* allows the system to match some terms and not others.

If a user not in the research group is running a job on the machine called *bass*, the **RANK** expression will evaluate to *0*, as all of the boolean terms evaluate to *0*. If the user *jones* submits a job, his job would match this machine and the **RANK** expression will evaluate to *1*. Therefore, the the job submitted by *jones* would pre-empt the running job. If the user *garrison* (the owner of the machine) later submits a job, the **RANK** expression will evaluate to *10* because the boolean that matches Jimmy gets multiplied by 10. In this case, the job submitted by *garrison* will pre-empt the job submitted by *jones*.

Example 7.2. A more complex application of the **RANK** expression in the machine ClassAd

The **RANK** expression can reference parameters other than **Owner**. If one machine has an enormous amount of memory and other do not have much at all, the **RANK** expression can be used to run jobs with larger memory requirements on the machine best suited to it, by using **RANK = ImageSize**. This preference will always service the largest of the jobs, regardless of which user has submitted them. Alternatively, a user could specify that their own jobs should run in preference to those with the largest **ImageSize** by using **RANK = (Owner == "user_name" * 1000000000000) + ImageSize**.

7.3. Conditions for state and activity transitions

This section lists all the possible state and activity transitions, with descriptions of the conditions under which each transition occurs.

Owner state

The *Owner* state represents a resource that is currently in use and not available to run jobs. When the **startd** is first spawned, the machine will enter the *Owner* state. The machine remains in the *Owner* state while the **IsOwner** expression evaluates to *TRUE*. If the **IsOwner** expression is *FALSE*, then the machine will transition to *Unclaimed*, indicating that it is ready to begin accepting jobs.

On a shared resource, the default value for the **IsOwner** is optimized to **START** *!=* *FALSE*. This causes the machine to remain in the *Owner* state as long as the **START** expression locally evaluates to *FALSE*. If the **START** expression locally evaluates to *TRUE* or cannot be locally evaluated (in which case, it will evaluate to *UNDEFINED*), the machine will transition to the *Unclaimed* state. For dedicated resources, the recommended value for the **IsOwner** expression is *FALSE*.



Note

The **IsOwner** expression should not reference job ClassAd attributes as this would result in an evaluation of *UNDEFINED*.

While in the *Owner* state, the **startd** polls the status of the machine. The frequency of this is determined by the **UPDATE_INTERVAL** configuration variable. The poll performs the following actions:

- Calculates load average
- Checks the access time on files
- Calculates the free swap space
- Notes if the **startd** has any critical tasks that need to be performed when the machine moves out of the *Owner* state

Whenever the machine is not actively running a job, it will transition back to the *Owner* state. Once a job is started, the value of **IsOwner** is no longer relevant and the job will either run to completion or be preempted.

Unclaimed state

The *Unclaimed* state represents a resource that is not currently in use by its owner or by MRG Grid.

Possible transitions from the *Unclaimed* state are:

1. *Owner:Idle*
2. *Matched:Idle*
3. *Claimed:Idle*
4. *Backfill:Idle*

When the `condor_negotiator` matches a machine with a job, it sends a notification of the match to each. Normally, the machine will enter the *Matched* state before progressing to *Claimed:Idle*. However, if the job receives the notification and initiates the claiming procedure before the machine receives the notification, the machine will transition directly to the *Claimed:Idle* state.

If the machine has been configured to perform backfill jobs it will evaluate the `START_BACKFILL` expression. When `START_BACKFILL` evaluates to *TRUE*, the machine will enter the *Backfill:Idle* state and begin running backfill jobs.

As long as the `IsOwner` expression is *TRUE*, the machine is in the *Owner* State. When the `IsOwner` expression is *FALSE*, the machine goes into the *Unclaimed* state. If the `IsOwner` expression is not present in the configuration files, then the default value is `START != FALSE`. This causes the machine to transition to the *Owner* state when the `START` expression locally evaluates to *TRUE*.

Effectively, there is very little difference between the *Owner* and *Unclaimed* states. The most obvious difference is how the resources are displayed in `condor_status` and other reporting tools. The only other difference is that benchmarking will only be run on a resource that is in the *Unclaimed* state. Whether or not benchmarking is run is determined by the `RunBenchmarks` expression. 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. Benchmarking performs and records two performance measures:

- MIPS (Millions of Instructions Per Second); and
- KFLOPS (thousands of Floating-point Operations Per Second).

When the benchmark is complete the machine returns to *Idle*.

This example runs benchmarking every four hours while the machine is in the *Unclaimed* state.

A macro called `BenchmarkTimer` is used in this example, which records the time since the last benchmark. When this time exceeds four hours, the benchmarks will be run again. A weighted average is used, so the more frequently the benchmarks run, the more accurate the data will be.

```
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks = $(BenchmarkTimer) >= (4 * $(HOUR))
```

Example 7.3. Setting benchmarks in the machine ClassAd

If `RunBenchmarks` is defined and set to anything other than *FALSE*, benchmarking will be run as soon as the machine transitions into the *Unclaimed* state. To completely disable benchmarks, set `RunBenchmarks` to *FALSE* or remove it from the configuration file.

Matched state

The *Matched* state occurs when the machine has been matched to a job by the negotiator, but the job has not yet claimed the machine. Machines are in this state for a very short period before transitioning.

When the machine is matched to a job, the machine will transition into the *Claimed:Idle* state. At any time while the machine is in the *Matched* state, if the `START` expression locally evaluates to *FALSE* the machine will enter the *Owner* state.

Machines in the *Matched* state will adjust the `START` expression so that the requirements evaluate to *FALSE*. This is to prevent the machine being matched again before it has been claimed.

The **startd** will start a timer when a machine transitions into the *Matched* state. This is to prevent the machine from staying in the *Matched* state for too long. The length of the timer can be adjusted with the **MATCH_TIMEOUT** configuration variable, which defaults to 120 seconds (2 minutes). If the job that was matched with the machine does not claim it within this period of time, the machine gives up, and transitions back into the *Owner* state. Normally, it would then transition straight back to the *Unclaimed* state to wait for a new match.

Claimed state

The *Claimed* state occurs when the machine has been claimed by a job. It is the most complex state, with the most possible transitions.

When the machine first enters the *Claimed* state it is in the *Idle* activity. If a job has claimed the machine and the claim will be activated, the machine will transition into the *Busy* activity and the job started. If a **condor_vacate** arrives, or the **START** expression locally evaluates to *FALSE*, it will enter the *Retiring* activity before transitioning to the *Pre-empting* state.

While in *Claimed:Busy*, the **startd** daemon will evaluate the **WANT_SUSPEND** expression to determine which other expression to evaluate. If **WANT_SUSPEND** evaluates to *TRUE*, the **startd** will evaluate the **SUSPEND** expression to determine whether or not to transition to *Claimed:Suspended*. Alternatively, if **WANT_SUSPEND** evaluates to *FALSE* the **startd** will evaluate the **PREEMPT** expression to determine whether or not to skip the *Suspended* state and move to *Claimed:Retiring* before transitioning to the *Pre-empting* state.

While a machine is in the *Claimed* state, the **startd** daemon will poll the machine for a change in state much more frequently than while in other states. The frequency can be adjusted by changing the **POLLING_INTERVAL** configuration variable.

The **condor_vacate** command affects the machine when it is in the *Claimed* state. There are a variety of events that may cause the **startd** daemon to try to suspend or kill a running job. Possible causes could be:

- The owner has resumed using the machine
- Load from other jobs
- The **startd** has been instructed to shut down
- The appearance of a higher priority claim to the machine by a different MRG Grid user.

The **startd** can be configured to handle interruptions in different ways. Activity on the machine could be ignored, or it could cause the job to be suspended or even killed. Desktop machines can benefit from a configuration that goes through successively more dramatic actions to remove the job. The least costly option to the job is to suspend it. If the owner is still using the machine after suspending the job for a short while, then **startd** will attempt to vacate the job. Vanilla jobs are sent a soft kill signal, such as **SIGTERM**, so that they can gracefully shut down. If the owner wants to resume using the machine, and vacating can not be completed, the **startd** will progress to kill the job. Killing is a quick death to a job. It uses a hard-kill signal that cannot be intercepted by the application. For vanilla jobs, vacating and killing are equivalent actions.

Pre-empting state

The *Pre-empting* state is used to evict a running job from a machine, so that a new job can be started. There are two possible activities while in the *Pre-empting* state. Which activity the machine is in is dependent on the value of the **WANT_VACATE** expression. If **WANT_VACATE** evaluates to *TRUE*,

the machine will enter the *Vacating* activity. Alternatively, if **WANT_VACATE** evaluates to *FALSE*, the machine will enter the *Killing* activity.

The main function of the *Pre-empting* state is to remove the **condor_starter** associated with the job. If the **condor_starter** associated with a given claim exits while the machine is still in the *Vacating* activity, then the job has successfully completed a graceful shutdown. For standard universe jobs, this means that a checkpoint was saved. For other jobs, it means that the application was given the opportunity to intercept the soft kill signal.

While in the *Pre-empting* state the machine advertises its **Requirements** expression as *FALSE*, to signify that it is not available for further matches. This is because it is about to transition to the *Owner* state, or because it has already been matched with a job that is currently pre-empting and further matches are not allowed until the machine has been claimed by the new job.

While the machine is in the *Vacating* activity, it continually evaluates the **KILL** expression. As soon as it evaluates to *TRUE*, the machine will enter the *Killing* activity.

When the machine enters the *Killing* activity it attempts to force the **condor_starter** to immediately kill the job. Once the machine has begun to kill the job, the **condor_startd** starts a timer. The length of the timer defaults to 30 seconds and can be adjusted by changing the **KILLING_TIMEOUT** macro. If the timer expires and the machine is still in the *Killing* activity, it is assumed that an error has occurred with the **condor_starter** and the startd will try to vacate the job immediately by sending **SIGKILL** to all of the children of the **condor_starter** and then to the **condor_starter** itself.

Once the **condor_starter** has killed 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 *Killing* activity. If the job was pre-empted because a better match was found, the machine will enter *Claimed:Idle*. If the pre-emption was caused by the machine owner, the machine will enter the *Owner* state.

Backfill state

The *Backfill* state is used whenever the machine is performing low priority background tasks. 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 not currently being used interactively or for regular MRG Grid jobs. 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*.

Once a machine is in the *Backfill* state, it will immediately attempt to spawn the backfill manager it has been configured to use. Once the backfill manager is running, the machine will enter the *Busy* activity to indicate that it is now performing a backfill computation.

On multi-slot machines the **condor_startd** will only spawn a single instance of the backfill manager, even if multiple slots are available to run backfill jobs. The first machine to enter *Backfill:Idle* will cause the backfill manager to start. If another slot subsequently enters the *Backfill* state and a backfill manager is already running the slot will immediately enter the *Busy* activity without starting another instance of the backfill manager.

There are several possible events that could cause the the **condor_startd** to kill the backfill manager and enter the *Killing* activity:

- The machine is matched or claimed for a MRG Grid job
- The owner has resumed using the machine

- The machine has received a `condor_vacate` command
- The `condor_startd` has been instructed to shut down

Once the backfill manager and all its children have exited the machine will enter the *Idle* activity. When this occurs, the machine will go into another state, depending on what caused the backfill manager to be killed.

While the machine is in the *Busy* activity, if the `EVICT_BACKFILL` expression evaluates to *TRUE* and the backfill manager has been successfully killed, the machine will return to the *Owner:Idle*. This will also occur if the backfill manager was killed as a result of the `condor_vacate` command, or if the `condor_startd` is shut down.

7.4. Defining a policy

When a transition occurs, MRG Grid records the time that the new activity or state was entered. These times can be used to write expressions for customized transitions. To define a policy, set expressions in the configuration file (see section 3.3 on Configuring Condor for an introduction to Condor's configuration files). The expressions are evaluated in the context of the machine's ClassAd and a job ClassAd. The expressions can therefore reference attributes from either ClassAd.

Default macros

The following default macros assist with writing human-readable expressions.

MINUTE

Defaults to *60*

HOURL

Defaults to $(60 * \$(MINUTE))$

StateTimer

Amount of time in the current state

Defaults to $(CurrentTime - EnteredCurrentState)$

ActivityTimer

Amount of time in the current activity

Defaults to $(CurrentTime - EnteredCurrentActivity)$

ActivationTimer

Amount of time the job has been running on this machine

Defaults to $(CurrentTime - JobStart)$

NonCondorLoadAvg

The difference between the system load and the MRG Grid load (equates to the load generated by everything except MRG Grid)

Defaults to $(LoadAvg - CondorLoadAvg)$

BackgroundLoad

Amount of background load permitted on the machine and still be able to start a job

Defaults to *0.3*

HighLoad

If the **NonCondorLoadAvg** goes over this, the CPU is considered too busy, and eviction of the job should start

Defaults to *0.5*

StartIdleTime

Amount of time the keyboard must be idle before starting a job

Defaults to *15 * \$(MINUTE)*

ContinueIdleTime

Amount of time the keyboard must to be idle before resumption of a suspended job

Defaults to *5 * \$(MINUTE)*

MaxSuspendTime

Amount of time a job may be suspended before more drastic measures are taken.

Defaults to *10 * \$(MINUTE)*

MaxVacateTime

Amount of time a job may spend attempting to checkpoint before giving up and killing it

Defaults to *10 * \$(MINUTE)*

KeyboardBusy

A boolean expression that evaluates to *TRUE* when the keyboard is being used

Defaults to *KeyboardIdle < \$(MINUTE)*

CPUIIdle

A boolean expression that evaluates to *TRUE* when the CPU is idle

Defaults to *\$(NonCondorLoadAvg) <= \$(BackgroundLoad)*

CPUBusy

A boolean expression that evaluates to *TRUE* when the CPU is busy

Defaults to *\$(NonCondorLoadAvg) >= \$(HighLoad)*

MachineBusy

The CPU or the Keyboard is busy

Defaults to *\$(CPUBusy) || \$(KeyboardBusy)*

CPUIsBusy

A boolean value set to the same value as **CPUBusy**

CPUBusyTime

the time in seconds since **CPUBusy** became *TRUE*. Evaluates to *0* if **CPUBusy** is *FALSE*

It is preferable to suspend jobs instead of killing them. This is especially true when the job uses little memory, when the keyboard is not being used or when the job is running in the vanilla universe. By default, these macros will gracefully vacate jobs that have been running for more than ten minutes, or are vanilla universe jobs:

```
WANT_SUSPEND      = ( $(SmallJob) || $(KeyboardNotBusy) || $(IsVanilla) )
WANT_VACATE       = ( $(ActivationTimer) > 10 * $(MINUTE) ||
$(IsVanilla) )
```

Expression Definitions

This list gives examples of typical expressions.

START

Start a job if the keyboard has been idle long enough and the load average is low enough or if the machine is currently running a job. Note that MRG Grid will only run one job at a time, but it may pre-empt the currently running job in favour of the new one:

```
START    = ( (KeyboardIdle > $(StartIdleTime)) \
&& ( $(CPUIIdle) || (State != "Unclaimed" \
&& State != "Owner")) )
```

SUSPEND

Suspend a job if the keyboard is in use. Alternatively, suspend if the CPU has been busy for more than two minutes and the job has been running for more than 90 seconds:

```
SUSPEND  = ( $(KeyboardBusy) || \
( (CpuBusyTime > 2 * $(MINUTE)) \
&& $(ActivationTimer) > 90 ) )
```

CONTINUE

Continue a suspended job if the CPU is idle, the Keyboard has been idle for long enough, and the job has been suspended more than 10 seconds:

```
CONTINUE = ( $(CPUIIdle) && ($(ActivityTimer) > 10) \
&& (KeyboardIdle > $(ContinueIdleTime)) )
```

PREEMPT

There are two conditions that signal pre-emption. The first condition is if the job is suspended, but it has been suspended too long. The second condition is if suspension is not desired and the machine is busy:

```
PREEMPT  = ( ((Activity == "Suspended") && \
$(ActivityTimer) > $(MaxSuspendTime)) \
|| (SUSPEND && (WANT_SUSPEND == False)) )
```

MaxJobRetirementTime

Do not give jobs any time to retire on their own when they are about to be pre-empted:

```
MaxJobRetirementTime = 0
```

KILL

Kill jobs that take too long to exit gracefully:

```
KILL    = $(ActivityTimer) > $(MaxVacateTime)
```

Example Policies

The following examples show how to use the default macros detailed in this chapter to create commonly used policies.



Warning

If you intend to change any of the settings as described in this chapter, make sure you follow the instructions carefully and always test your changes before implementing them. Mistakes in policy configuration can have a severe negative impact on both the owners of machines in your pool, and the users who submit jobs to those machines.

This example shows to set up a machine for running test jobs from a specified user.

The machine needs to behave normally unless the user *coltrane* submits a job. When this occurs, the job should start execution immediately, regardless of what else is happening on the machine at that time.

Jobs submitted by *coltrane* should not be suspended, vacated or killed. This is reasonable because *coltrane* will only be submitting very short running programs for testing purposes.

```
START      = ($(START)) || Owner == "coltrane"
SUSPEND    = ($(SUSPEND)) && Owner != "coltrane"
CONTINUE   = $(CONTINUE)
PREEMPT    = ($(PREEMPT)) && Owner != "coltrane"
KILL       = $(KILL)
```

There are no specific settings for the **CONTINUE** or **KILL** expressions. Because the jobs submitted by *coltrane* will never be suspended, the **CONTINUE** expression is irrelevant. Similarly, because the jobs can not be pre-empted, **KILL** is irrelevant.

Example 7.4. Test-job Policy

This example shows how to set up a machine to only run jobs at certain times of the day.

This is achieved through the **ClockMin** and **ClockDay** attributes. These are special attributes which are automatically inserted by the **condor_startd** into its ClassAd, so they can always be referenced in policy expressions. **ClockMin** defines the number of minutes that have passed since midnight. **ClockDay** defines the day of the week, where Sunday = 0, Monday = 1, and so on to Saturday = 7.

To make the policy expressions easier to read, use macros to define the time periods when you want jobs to run or not run. Regular work hours at your site could be defined as being from 0800 until 1700, Monday through Friday.

```
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
  (ClockDay > 0 && ClockDay < 6) )
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
  (ClockDay == 0 || ClockDay == 6) )
```

Once these macros are defined, MRG Grid can be instructed to only start jobs after hours:

```
START = $(AfterHours) && $(CPUIIdle) && KeyboardIdle > $(StartIdleTime)
```

Consider the machine busy during work hours, or if the keyboard or CPU are busy:

```
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
```

Avoid suspending jobs during work hours, so that in the morning, if a job is running, it will be immediately pre-empted, instead of being suspended for some length of time:

```
WANT_SUSPEND = $(AfterHours)
```

By default, the **MachineBusy** macro is used to define the **SUSPEND** and **PREEMPT** expressions. If you have changed these expressions, you will need to add **\$(WorkHours)** to your **SUSPEND** and **PREEMPT** expressions as appropriate.

Example 7.5. Time of Day Policy

This example shows to set up a pool of machines that include desktop machines and dedicated cluster machines, requiring different policies.

In this scenario, keyboard activity should not have any effect on the dedicated machines. It might be necessary to log into the dedicated machines to debug a problem, or change settings, and this should not interrupt the running jobs. 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 create a standard desktop policy and a standard non-desktop policy. The appropriate policy is then copied into the local configuration file for each machine. This example, however, defines one standard policy in `condor_config` with a toggle that can be set in the local configuration file.

If `IsDesktop` is configured, make it an attribute of the machine ClassAd:

```
STARTD_EXPRS = IsDesktop
```

If a desktop machine, only consider starting jobs if the load average is low enough or the machine is currently running a Condor job, and the user is not active:

```
START = ( ($(CPUIIdle) || (State != "Unclaimed" && State != "Owner")) \
  && (IsDesktop != True || (KeyboardIdle > $(StartIdleTime))) )
```

Suspend instead of vacating or killing for small or vanilla universe jobs:

```
WANT_SUSPEND = ( $(SmallJob) || $(JustCpu) \
  || $(IsVanilla) )
```

When pre-empting, vacate instead of killing for jobs that have been running for longer than 10 minutes, or vanilla universe jobs:

```
WANT_VACATE = ( $(ActivationTimer) > 10 * $(MINUTE) \
  || $(IsVanilla) )
```

Suspend jobs if the CPU has been busy for more than 2 minutes and the job has been running for more than 90 seconds. Also suspend jobs if this is a desktop and the user is active:

```
SUSPEND = ( ((CpuBusyTime > 2 * $(MINUTE)) && $(ActivationTimer) > 90) \
  || ( IsDesktop == True && $(KeyboardBusy) ) )
```

Continue jobs on a desktop machine if the CPU is idle, the job has been suspended more than 5 minutes and the keyboard has been idle for long enough:

```
CONTINUE = ( $(CPUIIdle) && $(ActivityTimer) > 300 \
  && (IsDesktop != True || (KeyboardIdle > $(ContinueIdleTime))) )
```

Pre-empt jobs if it has been suspended too long or the conditions to suspend the job has been met, but suspension is not desired:

```
PREEMPT = ( (Activity == "Suspended") && \
  $(ActivityTimer) > $(MaxSuspendTime) \
  || (SUSPEND && (WANT_SUSPEND == False)) )
```

The following expression determines retirement time. Replace `0` with the desired amount of retirement time for dedicated machines. 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:

```
KILL = $(ActivityTimer) > $(MaxVacateTime)
```

With this policy in `condor_config`, the local configuration files for desktops can now be configured with the following line:

```
IsDesktop = True
```

In all other cases, the default policy described above will ignore keyboard activity.

Example 7.6. Desktop/Non-Desktop Policy

This example shows how to prevent and disable pre-emption.

Pre-emption can result in jobs being killed. When this happens, the jobs remain in the queue and will be automatically rescheduled. It is highly recommend designing jobs that work well in this environment, rather than simply disabling pre-emption. Planning for pre-emption makes jobs more robust in the face of other sources of failure. The easiest way to do this is to use the standard universe, which provides the ability to produce checkpoints. If a job is incompatible with the requirements of the standard universe, the job can still gracefully shutdown and restart by intercepting the soft kill signal.

However, there can be cases where it is appropriate to force MRG Grid to never kill jobs within an upper time limit. This can be achieved with the following policy.

Allow a job to run uninterrupted for up to two days before forcing it to vacate:

```
MAXJOBRETIREMENTTIME = $(HOUR) * 24 * 2
```

Construction of this expression can be more complex. For example, it could specify a different retirement time for different users or different types of jobs. Additionally, the job might come with its own definition of **MAXJOBRETIREMENTTIME**, but this can only cause less retirement time to be used, never more than what the machine offers.

The longer the retirement time that is given, the slower reallocation of resources in the pool can become if there are long-running jobs. However, by preventing jobs from being killed, you may decrease the number of cycles that are wasted on non-checkpointable jobs that are killed.

Note that the use of **MAXJOBRETIREMENTTIME** limits the killing of jobs, but it does not prevent the pre-emption of resource claims. Therefore, it is technically not a way of disabling pre-emption, but simply a way of forcing pre-empting claims to wait until an existing job finishes or runs out of time.

To limit the pre-emption of resource claims, the following policy can be used. Some of these settings apply to the execute node and some apply to the central manager, so this policy should be configured so that it is read by both.

Disable pre-emption by machine activity:

```
PREEMPT = False
```

Disable pre-emption by user priority:

```
PREEMPTION_REQUIREMENTS = False
```

Disable pre-emption by machine rank by ranking all jobs equally:

```
RANK = 0
```

When disabling claim pre-emption, it is advised to also optimize negotiation:

```
NEGOTIATOR_CONSIDER_PREEMPTION = False
```

Without any pre-emption of resource claims, once the **condor_negotiator** gives the **condor_schedd** a match to a machine, the **condor_schedd** may hold onto this claim indefinitely, as long as the user keeps supplying more jobs to run. To avoid this behavior, force claims to be retired after a specified period of time by setting the **CLAIM_WORKLIFE** variable. This enforces a time limit, beyond which no new jobs may be started on an existing claim. In this case, the **condor_schedd** daemon is forced to go back to the **condor_negotiator** to request a new match. The execute machine configuration would include a line that forces the schedd to renegotiate for new jobs after 20 minutes:

```
CLAIM_WORKLIFE = 1200
```

It is not advisable to set **NEGOTIATOR_CONSIDER_PREEMPTION** to *False*, as it can potentially lead to some machines never being matched to jobs.

Example 7.7. Disabling Pre-emption

This example shows how to create a policy around job suspension.

When jobs with a higher priority are submitted, the executing jobs might be pre-empted. These jobs can 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.

A policy can be created that will allow jobs to be suspended instead of evicted. The policy utilizes two slots: *slot1* only runs jobs identified as high priority jobs; *slot2* is set to run jobs according to the usual policy and to suspend them when *slot1* is claimed. A policy for a machine with more than one physical CPU may be adapted from this example. Instead of having two slots, you would have twice times the number of physical CPUs. Half of the slots would be for high priority jobs and the other half would be for suspendable jobs.

Tell MRG Grid that the machine has two slots, even though it only has a single CPU:

```
NUM_CPUS = 2
```

slot1 is the high-priority slot, while *slot2* is the background slot:

```
START = (SlotID == 1) && $(SLOT1_START) || \
  (SlotID == 2) && $(SLOT2_START)
```

Only start jobs on *slot1* if the job is marked as a high-priority job:

```
SLOT1_START = (TARGET.IsHighPrioJob == TRUE)
```

Only start jobs on *slot2* if there is no job on *slot1*, and if the machine is otherwise idle. Note that the *Busy* activity is only in the *Claimed* state, and only when there is an active job:

```
SLOT2_START = ( (slot1_Activity != "Busy") && \
  (KeyboardIdle > $(StartIdleTime)) && \
  ($(CPUIidle) || (State != "Unclaimed" && State != "Owner"))) )
```

Suspend jobs on *slot2* if there is keyboard activity or if a job starts on *slot1*:

```
SUSPEND = (SlotID == 2) && \
  ( (slot1_Activity == "Busy") || ($(KeyboardBusy)) )
```

```
CONTINUE = (SlotID == 2) && \
  (KeyboardIdle > $(ContinueIdleTime)) && \
  (slot1_Activity != "Busy")
```

Note that in this example, the job ClassAd attribute **IsHighPrioJob** has no special meaning. It is an invented name chosen for this example. To take advantage of the policy, a user must submit high priority jobs with this attribute defined. The following line appears in the job's submit description file as:

```
+IsHighPrioJob = True
```

Example 7.8. Job Suspension

The Virtual Machine Universe

Virtual Machines can be operated under MRG Grid using Xen (with libvirt). MRG Grid requires some configuration before being used with virtual machine. This chapter contains information on getting started.

Before you begin configuring MRG Grid to work with virtual machines, you will need to install the virtualization package according to the vendor's instructions.

For Xen, there are four requirements for MRG Grid to fully support it:

1. A Xen kernel must be running on the executing machine. The running Xen kernel acts as Dom0. All virtual machines, called DomUs, will be started under this kernel
2. The `libvirtd` service must be installed and running. This service is provided by the `libvirt` package
3. A recent version of the `mkisofs` utility must be available. This utility is used to create CD-ROM disk images, and is provided by the `mkisofs` package
4. The `pygrub` program must be available. This program executes virtual machines whose disks contain the kernel they will run. This program is provided by the `xen`

8.1. Configuring MRG Grid for the virtual machine universe

The configuration files for MRG Grid include various configuration settings for virtual machines. Some settings are required, while others are optional. This section discusses only the required settings.

Initial setup

1. Specify the type of virtualization software that is installed, using the `VM_TYPE` setting:

```
VM_TYPE = xen
```

2. Specify the location of `condor_vm-gahp` and its configuration file, using the `VM_GAHP_SERVER` settings:

```
VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp
```

Configuring `condor_vm-gahp`

These options belong in the local configuration file.

1. Initially, specify the type of virtualization software that is installed, using the `VM_TYPE` setting:

```
VM_TYPE = xen
```

2. Specify your kernel version. You can check which kernel version you have by running the `uname -r` command:

```
VM_VERSION = your kernel version
```

3. Although this information is required, it does not alter the behavior of `condor_vm-gahp`. The information is added to the machine ClassAd. If your virtualization software supports features that are desirable for job matching, it can be specified in the `RANK` expression.

Xen-specific configuration

Additional configuration is necessary for Xen.

1. Specify the location of the control script:

```
XEN_SCRIPT = $(SBIN)/condor_vm_xen.sh
```

2. Specify the default kernel image with the `XEN_DEFAULT_KERNEL` configuration variable. This is the kernel image to be used in cases where one explicitly specified in a job submission. In most cases, this is the default kernel from which the system was booted:

```
XEN_DEFAULT_KERNEL = /boot/vmlinuz-2.6.18-1.2798.fc6xen
```

3. Although it is not required, it may be necessary to set the default initrd image for Xen to use on Unix-based platforms. Unlike the kernel image, the default initrd image should not be set to the same one used to boot the system. In this case, create a new initrd image by running `mkinitrd` from the shell prompt and loading the `xennet` and `xenblk` drivers into it.
4. Specify the `XEN_BOOTLOADER`. The bootloader allows you to select a kernel instead of specifying the Dom0 configuration, and allows the use of the `xen_kernel = included` specification when submitting a job to the VM universe. A typical bootloader is `pygrub`:

```
XEN_BOOTLOADER = /usr/bin/pygrub
```

5. A typical configuration file for Xen is:

```
VM_TYPE = xen
VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp
VM_GAHP_LOG = $(LOG)/VMGahpLog
MAX_VM_GAHP_LOG = 1000000
VM_GAHP_DEBUG = D_FULLDEBUG
VM_VERSION = X.Y.Zxen
VM_MEMORY = 1024
VM_MAX_MEMORY = 1024
XEN_SCRIPT = $(SBIN)/condor_vm_xen.sh
XEN_BOOTLOADER = /usr/bin/pygrub
XEN_DEFAULT_KERNEL = /boot/vmlinuz-X.Y.Zxen
XEN_DEFAULT_INITRD = /boot/initrd-X.Y.Zxen.img
```


Restarting MRG Grid with virtualization settings

1. Once the configuration options have been set, restart the `condor_startd` daemon on the host. You can do this by running `condor_restart`. This should be performed on the central manager machine:

```
$ condor_restart -startd machine_name
```



Note

If the `condor_startd` daemon is currently servicing jobs it will let them finish running before restarting. If you want to force the `condor_startd` daemon to restart and kill any running jobs, add the `-fast` option to the `condor_restart` command.

2. The `condor_startd` daemon will pause while it performs the following checks:

- Exercise the virtual machine capabilities of `condor_vm-gahp`
- Query the properties
- Advertise the machine to the pool as VM-capable

If these steps complete successfully, `condor_status` will record the virtual machine type and version number. These details can be displayed by running the following command from the shell prompt:

```
$ condor_status -vm machine_name
```

If this command does not display output after some time, it is likely that `condor_vm-gahp` is not able to execute the virtualization software. The problem could be caused by configuration of the virtual machine, the local installation, or a variety of other factors. Check the `VMGahpLog` log file for diagnostics.

3. When using Xen for virtualization, the VM Universe is only available when MRG Grid is started with the root user or administrator. These privileges are required to create a virtual machine on top of a Xen kernel, as well as to use the `virsh` utility that controls creation and management of Xen guest virtual machines.

High Availability

MRG Grid can be configured to provide high availability. If a machine is not functioning - either because of scheduled downtime or due to a system failure - other machines can take on key functions. There are two specialized cases for the use of high availability with MRG Grid:

- Availability of the job queue - the machine running the `condor_schedd` daemon; and
- Availability of the central manager - the machine running the `condor_negotiator` and `condor_collector` daemons.

This chapter discusses how to set up high availability for both these scenarios.

9.1. High availability of the job queue

The `condor_schedd` daemon controls the job queue. If the job queue is not functioning then the entire pool will be unable to run jobs. This situation can be made worse if one machine is a dedicated submission point for jobs. When a job on the queue is executed, a `condor_shadow` process runs on the machine it was submitted from. The purpose of this process is to handle all input and output functionality for the job. However, if the machine running the queue becomes non-functional, `condor_shadow` can not continue communication and no jobs can continue processing.

Without high availability, the job queue would persist, but further jobs would be made to wait until the machine running the `condor_schedd` daemon became available again. By enabling high availability, management of the job queue can be transferred to other designated schedulers and reduce the chance of an outage. If jobs are required to stop without finishing, they can be restarted from the beginning, or can continue execution from the most recent checkpoint.

To enable high availability, the configuration is adjusted to specify alternate machines that can be used to run the `condor_schedd` daemon. To prevent multiple instances of `condor_schedd` running, a lock is placed on the job queue. When the machine running the job queue fails, the lock is lifted and `condor_schedd` is transferred to another machine. Configuration variables are also used to determine the intervals at which the lock expires, and how frequently polling for expired locks should occur.

Configuring high availability for the job queue

1. Add the following lines to the local configuration of all machines that are able to run the `condor_schedd` daemon and become the single pool submission point:

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
VALID_SPOOL_FILES = SCHEDD.lock
```

The `MASTER_HA_LIST` macro identifies the `condor_schedd` daemon as a daemon that should be kept running.

2. Each machine with this configuration must have access to the job queue lock. This synchronizes which single machine is currently running the `condor_schedd`. `SPOOL` identifies the location of the job queue, and needs to be accessible by all High Availability schedulers. This is typically accomplished by placing the `SPOOL` directory in a file system that is mounted on all schedulers.

`HA_LOCK_URL` identifies the location of the job queue lock. Like `SPOOL`, this needs to be accessible by all High Availability Schedulers, and is often found in the same location.

Always add `SCHEDD.lock` to the `VALID_SPOOL_FILES` variables. This is to prevent `condor_preem` deleting the lock file because it is not aware of it.

When a machine that is able to run the `condor_schedd` daemon is started, the `condor_master` daemon will poll the lock. If no lock is currently held, it will assume that no `condor_schedd` is currently running. It will then acquire the lock and start the `condor_schedd` daemon. If a lock is currently held by another machine, the `condor_schedd` daemon will not be started.

The machine running the `condor_schedd` daemon renews the lock periodically. If the machine fails to renew the lock, because the machine is not functioning the lock will become *stale*. The lock can also be released if `condor_off` or `condor_off -schedd` is executed. When another machine that is capable of running `condor_schedd` becomes aware that the lock is stale, it will attempt to acquire the lock and start the `condor_schedd`.

Remote job submission

1. When submitting jobs remotely, the scheduler needs to be identified, using a command such as `$ condor_submit -remote schedd_name myjob.submit`
2. The command above assumes a single `condor_schedd` running on a single machine. When high availability is configured, there are multiple possible `condor_schedd` daemons, with any one of them providing a single submission point.
3. So that jobs can be successfully submitted in a high availability situation, adjust the `SCHEDD_NAME` variable in the local configuration of each potential High Availability Scheduler. They will need to have the same value on each machine that could potentially be running the `condor_schedd` daemon. Ensure that the value chosen ends with the `@` character. This will prevent MRG Grid from modifying the value set for the variable.

```
SCHEDD_NAME = had-schedd@
```

4. The command to submit a job is now `$ condor_submit -remote had-schedd@ myjob.submit`

9.2. High availability of the central manager

The `condor_negotiator` and `condor_collector` daemons are critical to a pool functioning correctly. Both daemons usually run on the same machine, referred to as the *central manager*. If a central manager machine fails, MRG Grid will not be able to match new jobs or allocate new resources. Configuring high availability in a pool reduces the chance of an outage.

High availability allows one of multiple machines within the pool to function as the central manager. While there can be many active `condor_collector` daemons, only a single, active `condor_negotiator` will be running. The machine with the `condor_negotiator` daemon running is the active central manager. All machines running a `condor_collector` daemon are idle central managers. All submit and execute machines are configured to report to all potential central manager machines.

Every machine that can potentially be a central manager needs to run the high availability daemon `condor_had`. The daemons on each of the machines will communicate to monitor the pool and

ensure that a central manager is always available. If the active central manager stops functioning, the `condor_had` daemons will detect the failure. The daemons will then select one of the idle machines to become the new active central manager.

If the outage is caused by a network partition, the idle `condor_had` daemons on each side of the partition will choose a new active central manager. As long as the partition exists, there will be an active central manager on each side. When the partition is removed and the network repaired, the `condor_had` daemons will be re-organized and ensure that only one central manager is active.

It is recommended that a single machine is considered the primary central manager. If the primary central manager stops functioning, a secondary central manager can take over. When the primary central manager recovers, it will reclaim central management from the secondary machine. This is particularly useful in situations where the primary central manager is a reliable machine that is expected to have very short periods of instability. An alternative configuration allows the secondary central manager to remain active after the failed central manager machine is restarted.

The high availability mechanism on the central manager operates by monitoring communication between machines. Note that there is a significant difference in communications between machines when:

1. The machine is completely down - crashed or switched off
2. The machine is functioning, but the `condor_had` daemon is not running

The high availability mechanism operates only when the machine is down. If the daemons are simply not running, the system will not select a new active central manager.

The central manager machine records state information, including information about user priorities. Should the primary central manager fail, a pool with high availability enabled would lose this information. Operation would continue, but priorities would be re-initialized. To prevent this occurring, the `condor_replication` daemon replicates the state information on all potential central manager machines. The `condor_replication` daemon needs to be running on the active central manager as well as all potential central managers.

The high availability of central manager machines is enabled through the configuration settings. It is disabled by default. All machines in a pool must be configured appropriately in order to make the high availability mechanism work.

The stabilization period is the time it takes for the `condor_had` daemons to detect a change in the pool state and recover from this change. It is computed using the following formula:

```
stabilization period = 12 * [number of central managers] *  
$(HAD_CONNECTION_TIMEOUT)
```

Configuring high availability on potential central manager machines

The following is the procedure for configuring machines that are potential central managers.

1. Firstly, remove any parameters from the `NEGOTIATOR_HOST` and `CONDOR_HOST` macros:

```
NEGOTIATOR_HOST=  
CONDOR_HOST=
```

2.



Note

The following settings must be the same on all potential central manager machines:

In order to make writing other expressions simpler, define a variable for each potential central manager in the pool.

```
CENTRAL_MANAGER1 = cm1.example.com
CENTRAL_MANAGER2 = cm2.example.com
```

3. List all the potential central managers in the pool:

```
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)
```

4. Define a macro for the port number that **condor_had** will listen on. The port number must match the port number used when defining **HAD_LIST**. This port number is arbitrary, but ensure that there are no port number collisions with other applications:

```
HAD_PORT = 51450
HAD_ARGS = -p $(HAD_PORT)
```

5. Define a macro for port number that **condor_replication** will listen on. The port number must match the port number specified for the replication daemon in **REPLICATION_LIST**. The port number is arbitrary, but ensure that there are no port number collisions with other applications:

```
REPLICATION_PORT = 41450
REPLICATION_ARGS = -p $(REPLICATION_PORT)
```

6. Specify a list of addresses for the replication list. It must contain the same addresses as those listed in **HAD_LIST**. Additionally, for each hostname specify the port number of the **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)
```

7. Specify a list of addresses for the high availability list. It must contain the same addresses in the same order as the list under **COLLECTOR_HOST**. Additionally, for each hostname specify the port number of the **condor_had** daemon running on that host. The first machine in the list will be considered the primary central manager if **HAD_USE_PRIMARY** is set to **TRUE**:

```
HAD_LIST = $(CENTRAL_MANAGER1):$(HAD_PORT),$(CENTRAL_MANAGER2):
$(HAD_PORT)
```

8. Specify the high availability daemon connection time. Recommended values are:

- 2 if the central managers are on the same subnet
- 5 if security is enabled
- 10 if the network is very slow, or to reduce the sensitivity of the high availability daemons to network failures

```
HAD_CONNECTION_TIMEOUT = 2
```

9. Select whether or not to use the first central manager in the **HAD_LIST** as a primary central manager:

```
HAD_USE_PRIMARY = true
```

10. Specify which machines have root or administrator privileges within the pool. This is normally set to the machine where the MRG Grid administrator works, provided all users who log in to that machine are trusted:

```
HOSTALLOW_ADMINISTRATOR = $(COLLECTOR_HOST)
```

11. Specify which machines have access to the **condor_negotiator**. These are trusted central managers. The default value is appropriate for most pools:

```
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
```

- 12.



Note

The following settings can vary between machines. They are master specific parameters:

Specify the location of executable files:

```
HAD = $(SBIN)/condor_had
REPLICATION = $(SBIN)/condor_replication
```

13. List the daemons that the master central manager should start. It should contain at least the following five daemons:

```
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
DC_DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
```

The **DC_DAEMON_LIST** should also include any other daemons running on the node.

14. Specify whether or not to enable the replication feature:

```
HAD_USE_REPLICATION = true
```

15. Name of the file to be replicated:

```
STATE_FILE = $(SPOOL)/Accountantnew.log
```

16. Specify how long (in seconds) to wait in between attempts to replicate the file:

```
REPLICATION_INTERVAL = 300
```

17. Specify how long (in seconds) transferer daemons have to complete the download/upload process:

```
MAX_TRANSFERER_LIFETIME = 300
```

18. Specify how long (in seconds) for the **condor_had** to wait in between sending ClassAds to the **condor_collector**:

```
HAD_UPDATE_INTERVAL = 300
```

19. Specify the master negotiator controller and the back-off constant:

```
MASTER_NEGOTIATOR_CONTROLLER = HAD  
MASTER_HAD_BACKOFF_CONSTANT = 360
```



Important

If the backoff constant value is too small, it can result in the **condor_negotiator** churning. This occurs when a constant cycling of the daemons stopping and starting prevents the **condor_negotiator** from being able to run long enough to complete a negotiation cycle. Churning causes an inability for any job to start processing. Increasing the **MASTER_HAD_BACKOFF_CONSTANT** variable can help solve this problem.

20. Specify the maximum size (in bytes) of the log file:

```
MAX_HAD_LOG = 640000
```

21. Specify the debug level:

```
HAD_DEBUG = D_COMMAND
```

22. Specify the location of the log file for **condor_had**:

```
HAD_LOG = $(LOG)/HADLog
```

23. Specify the maximum size (in bytes) of the replication log file:

```
MAX_REPLICATION_LOG = 640000
```


24. Specify the debug level for replication:

```
REPLICATION_DEBUG = D_COMMAND
```

25. Specify the location of the log file for **condor_replication**:

```
REPLICATION_LOG = $(LOG)/ReplicationLog
```

Configuring high availability on other machines in the pool

Machines that are not potential central managers also require configuration for high availability to work correctly. The following is the procedure for configuring machines that are in the pool, but are not potential central managers.

1. Firstly, remove any parameters from the **NEGOTIATOR_HOST** and **CONDOR_HOST** macros:

```
NEGOTIATOR_HOST=
CONDOR_HOST=
```

2. Define a variable for each potential central manager:

```
CENTRAL_MANAGER1 = cm1.example.com
CENTRAL_MANAGER2 = cm2.example.com
```

3. Specify a list of all potential central managers:

```
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)
```

4. Specify which machines have access to the **condor_negotiator**. These are trusted central managers. The default value is appropriate for most pools:

```
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
```

Using a high availability pool without replication

1. Set the **HAD_USE_REPLICATION** configuration variable to *FALSE*. This will disable replication at the configuration level.
2. Remove *REPLICATION* from both the **DAEMON_LIST** and **DC_DAEMON_LIST** in the configuration file.

Disabling high availability on the central manager

1. To disable the high availability mechanism on central managers, remove the *HAD*, *REPLICATION*, and *NEGOTIATOR* settings from the **DAEMON_LIST** configuration variable on all machines except the primary machine. This will leave only one **condor_negotiator** remaining in the pool.
2. To shut down a high availability mechanism that is currently running run the following commands from a host with root or administrator privileges on all central managers:
 - a. **condor_off -all -neg**

b. `condor_off -all -subsystem -replication`

c. `condor_off -all -subsystem -had`

These commands will kill all the currently running `condor_had`, `condor_replication` and `condor_negotiator` daemons.

3. Run the command `condor_on -neg` on the machine where the single `condor_negotiator` is going to operate.

Cloud Computing

The Elastic Compute Cloud - or *EC2* - allows the execution of jobs in a virtual computing environment provided by [Amazon Web Services](http://aws.amazon.com/ec2/)¹ (AWS). EC2 can be considered as extra computing capacity you can leverage when needed as an extension to an existing pool.

EC2 takes advantage of virtual machine technology. A virtual machine image is referred to as an Amazon Machine Image (AMI). EC2 users can build an AMI specific to the type of application they wish to run. EC2 also provides command tools and APIs to manage AMIs.

The starting, monitoring and cleaning up of EC2 resources occurs at the local level. The application is installed in an AMI stored in S3, and, once started, is responsible for the life-cycle of the job and the termination of the AMI instance.

AMI instances running in EC2 do not have persistent storage directly available. It is advisable to program the AMI to transfer the output from a job out of the running instance before it is shut down.

An AMI is fixed at creation time. This means that it cannot be customized before an instance of it is started in EC2. However, an AMI instance can be customized using EC2's *user-data* functionality. This allows for an AMI instance to receive instantiation specific input and customize its operation.

MRG Grid uses EC2 in two different ways:

- MRG/EC2 Basic
- MRG/EC2 Enhanced

In order to use MRG/EC2 Basic and MRG/EC2 Enhanced, an Amazon Web Services (AWS) account is required. For information on how to obtain an AWS account and other information on EC2, including billing rates and terms and conditions, visit the [Amazon Web Services website](http://aws.amazon.com/)².

You will need to create an Amazon Machine Image (AMI). The AMI contains information about your operating system, binaries, installed software and configuration information and allows you to be able to run applications in the cloud.



Note

More information about configuring the AMI can be found in the [Amazon Getting Started Guide](#)³



Note

You must purchase *Red Hat Enterprise MRG Grid Amazon EC2 Execute Node* products at Amazon to run and be entitled to RHEL+MRG at EC2. Hourly pricing and additional information for these products are located at <http://www.redhat.com/solutions/cloud/>

10.1. MRG/EC2 Basic

With MRG/EC2 Basic an AMI can be submitted as a job to EC2. This is useful when deploying a complete application stack into EC2. The AMI contains the operating system and all the required

¹ <http://aws.amazon.com/ec2/>

² <http://aws.amazon.com/>

packages. EC2 will boot the image and the image can initialize the application stack on boot. MRG/EC2 Basic knowledge is also important when using MRG/EC2 Enhanced.

When setting up MRG Grid for use with EC2 for the first time, the following steps are important:

1. Make changes to your local condor configuration file
 2. Prepare the job submission file for EC2 use
 3. Set up a security group on EC2 (this step is optional)
 4. Submit the job
 5. Check that the job is running in EC2
 6. Check the image using **ssh** (this step is optional)
1. MRG Grid is configured to work with EC2 by default. The necessary configuration settings are in the global configuration file. There is one additional setting you may wish to add to the local configuration:

```
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE_AMAZON = 10
```

This setting will limit the number of EC2 jobs that can be submitted at any time. AWS has an upper limit of 20. Setting the maximum to less than 20 can help avoid problems.

2. The following is an example of a simple job submission file for MRG/EC2 Basic:

```
# Note to submit an AMI as a job we need the grid universe
Universe = grid
grid_resource = amazon

# Executable in this context is just a label for the job
Executable = my_amazon_ec2_job
transfer_executable = false

# Keys provided by AWS
amazon_public_key = cert-ABCDEFGHIJKLMNPOQRSTUVWXYZ.pem
amazon_private_key = pk-AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPP.pem

# The AMI ID
amazon_ami_id = ami-123a456b
amazon_user_data = Hello EC2!

# The keypair file if needed for using ssh etc
amazon_keypair_file = /tmp/keypair

# The security group for the job
amazon_security_groups = MY_SEC_GRP
```

```
queue
```

MRG/EC2 Basic requires the grid universe and the *amazon* grid resource. The executable is a label that will show up in the job details when using commands such as `condor_q`. It is not an executable file.

The AMI ID of the image needs to be specified in the job submission file. User data can also be passed to the remote job if it is required. Applications that require user data can access it using a Representational State Transfer (REST) based interface. Information on how to access image instance data, including user data, is available from [Amazon Web Services Developer Guide](#)⁴.

EC2 will provide a keypair for access to the image if required. The `amazon_keypair_file` command specifies where this will be stored.

EC2 allows users to specify one or more security groups. Security groups can specify what type of access is available. This can include opening specific ports - e.g. port 22 for `ssh` access.

Advanced options

This step is optional. EC2 provides several options for instance types:

- `m1.small`: i386 instance with 1 compute unit
- `m1.large`: x86_64 instance with 4 compute unit
- `m1.xlarge`: x86_64 instance with 8 compute units
- `c1.medium`: i386 instance with 5 compute units
- `c1.xlarge`: x86_64 instance with 20 compute units

The default instance type is `m1.small` and assumes an i386 architecture. For example, if the AMI you are deploying is x86_64 then you will need to set the following value in your job submission:

```
amazon_instance_type = m1.large
```

For more information on instance types see the [Amazon EC2 Developer Guide](#)⁵.



Note

You could be using the wrong instance type if you see a message like this in your job ClassAd when you run `condor_q -l`:

```
HoldReason = "The requested instance type's architecture (i386)
does not match the architecture in the manifest for ami-
bda347d4 (x86_64) "
```

3. *This step is optional.* If `ssh` or other access is required, EC2 provides APIs and commands to create and modify a security group. Download the AMI command line utilities here from the

[Amazon Web Services Developer Site](#)⁶. Documentation on the APIs and command line utilities are also on the [Amazon Web Services Developer Site](#)⁷.

To use the command line utilities provided by AWS, you will need to set some environment variables.

Set **EC2_HOME** to point to the location of the tools. The EC2 tools are normally downloaded in a zip file, using version numbers:

```
export EC2_HOME=/home/myuser/ec2-api-tools-X.Y-ZZZZ
```

EC2 requires X509 certificates. These can be downloaded from your AWS account and set using the following variables:

```
export EC2_CERT=/home/myuser/keys/cert-  
MPMCULQDTBLIBUEPGBVK2LIEV6AN6GB.pem  
export EC2_PRIVATE_KEY=/home/myuser/keys/pk-  
MPMCULQDTBLIBUEPGBVK2LIEV6AN6GB.pem
```

The EC2 commands require Java, so **JAVA_HOME** must also be set:

```
export JAVA_HOME=/etc/alternatives/jre_1.5.0
```

Use the following commands from the bin directory to create a security group and allow **ssh** access to the AMI. The following are examples. For more information see the documentation at the [Amazon Web Services Developer Site](#)⁸. To create a new group called **MY_SEC_GRP** and a short description:

```
./ec2-add-group MY_SEC_GRP -d "My Security Group"
```

Open port 22 and allow **ssh** access:

```
./ec2-authorize MY_SEC_GRP -p 22
```

4. Submit the job using **condor_submit**, as normal.
5. You can check on the status of EC2 jobs, just as regular MRG Grid jobs, by using the **condor_q** and **condor_q -l** commands. When the image has been successfully loaded in EC2 and the job is running, the **condor_q -l** command will show the address of the AMI using the label *AmazonRemoteVirtualMachineName*:

```
$ condor_q -l
```

```
AmazonRemoteVirtualMachineName =
    "ec2-99-111-222-44.compute-1.amazonaws.com"
```



Note

There are tools available for managing running APIs. One of these is the Mozilla™ Firefox™ plugin *Elasticfox*⁹.

6. *This step is optional.* If you are using **ssh** and have opened the appropriate port, **ssh** can also be used to access the running image with a remote shell. The keypair file specified in the job is required:

```
$ ssh -i /tmp/keypair root@ec2-99-111-222-44.compute-1.amazonaws.com
```

This example contains a script for a job to be executed by an AMI. Edit the `/etc/rc.local` file in the AMI and place this code at the end.

This example reads data from the `user-data` field, creates a file called `output.txt` and transfers that file out of the AMI before shutting down.

```
-- /etc/rc.local --
#!/bin/sh

USER_DATA=`curl http://169.254.169.254/2007-08-29/user-data`

ARGUMENTS="${USER_DATA%;*}"
RESULTS_FILE="${USER_DATA#*;}"

mkdir /tmp/output
cd /tmp/output

/bin/echo "$ARGUMENTS" > output.txt

cd /tmp
tar czf "$RESULTS_FILE" /tmp/output

curl --ftp-pasv -u user:password -T "$RESULTS_FILE" ftp://server/output

shutdown -h -P now
```

Example 10.1. Creating a script to run an MRG/EC2 Basic job in an AMI

10.2. MRG/EC2 Enhanced

The MRG/EC2 Enhanced feature is an extension of MRG/EC2 Basic that allows vanilla universe jobs to be run in Amazon's EC2 service. MRG/EC2 Enhanced uses generic AMIs to execute vanilla universe jobs. Jobs executed with MRG/EC2 Enhanced act like any other vanilla universe job, except the execution node is in EC2 instead of a local condor pool.

MRG/EC2 Enhanced uses security provided by the AWS X509 certificates as well as AWS access/secret access keys. Credentials for these measures can be supplied by an administrator on the MRG Grid configuration files, or on a job-by-job basis in the individual submit files. Credentials in the configuration files will be checked first, and will override any provided in the submit file.

To use MRG/EC2 Enhanced, you will need an Amazon Web Services (AWS) account with access to the following features:

- EC2
- SQS (Simple Queue Service)
- S3 (Simple Storage Service)

This chapter provides instructions on how to download and install the necessary RPMs and Amazon Machine Images (AMIs) for the use and operation of the MRG Grid MRG/EC2 Enhanced feature.

Configuring an Amazon Machine Image

1. On the AMI, use **yum** to install the **condor-ec2-enhanced** package:

```
# yum install condor-ec2-enhanced
```

2. Create a private key file called *private_key*:

```
$ openssl genrsa -out private_key 1024
```

Create a public key file called *public_key*:

```
$ openssl rsa -in private_key -out public_key -pubout
```



Note

These keys are generated using openssl, and are not the same as the AWS keys needed elsewhere.

Copy the contents of **private_key** into the file **/root/.ec2/rsa_key** on the AMI. The private key must match the public key set in **set_rsapublickey** for a given route or job.

3. The following changes can be specified in any condor configuration file, however it is recommended that they are added to the local configuration file at **/var/lib/condor/condor_config.local**:

Specify the location of the **condor_startd** hooks:

```
EC2ENHANCED_HOOK_FETCH_WORK = $(LIBEXEC)/hooks/hook_fetch_work.py
EC2ENHANCED_HOOK_REPLY_FETCH = $(LIBEXEC)/hooks/hook_reply_fetch.py
```


4. Specify the location of the starter hooks:

```
EC2ENHANCED_HOOK_PREPARE_JOB = $(LIBEXEC)/hooks/hook_prepare_job.py
EC2ENHANCED_HOOK_UPDATE_JOB_INFO = $(LIBEXEC)/hooks/
hook_update_job_status.py
EC2ENHANCED_HOOK_JOB_EXIT = $(LIBEXEC)/hooks/hook_job_exit.py
```

5. Specify the job hook keywords:

```
STARTD_JOB_HOOK_KEYWORD = EC2ENHANCED
STARTER_JOB_HOOK_KEYWORD = EC2ENHANCED
```

6. Set the delay for fetching work and the update interval:

```
FetchWorkDelay = 10
STARTER_UPDATE_INTERVAL = 30
```

7. The **caroniad** daemon is used in the MRG/EC2 Enhanced AMI instance to retrieve and process MRG Grid jobs. In order to do this **caroniad** communicates with Condor hooks that may or may not be running on the same machine. The daemon is configured by editing the configuration file located at `/etc/opt/grid/caroniad.conf`. The parameters that need to be changed are:

- **ip**

caroniad will listen on this IP address.



Note

By default, the hooks and **caroniad** will run on the same machine. In this case, the loopback IP address is sufficient.

- **port**

The port **caroniad** should listen on

- **queued_connections**

The number of outstanding connections

- **lease_time**

The amount of time that a job can run without performing an update. If a job has not performed an update within this time frame, it is assumed that an error has occurred and the job will be released or re-sent. This value must be longer than the value specified for **STARTER_UPDATE_INTERVAL**.

- **lease_check_interval**

The interval to wait between checks to see if a job has had an error

8. The hook configuration file located at `/etc/opt/grid/job-hooks.conf` also needs to be configured to communicate with `caroniad`. Adjust the `ip` and `port` parameters to the IP address and port that the hooks should use to communicate with `caroniad`.
9. Configure MRG Grid to start on boot using the `chkconfig` command as the root user at the shell prompt:

```
# chkconfig --level 2345 condor on
```

Perform the same action for `condor-ec2-enhanced`:

```
# chkconfig --level 2345 condor-ec2-enhanced on
```

10. Package the AMI. This step will vary depending on how you are building your AMI. If you have changed an existing AMI you should use the following commands (please see the [Amazon Getting Started Guide](#)¹⁰ for more information on how to use these commands):

On the AMI instance run:

```
$ ec2-bundle-vol
$ ec2-upload-bundle
```

After uploading the bundle it must be registered. On the local machine, register the bundle using the command:

```
$ ec2-register
```

The registration process will return an AMI ID. This ID will be needed when submitting jobs.

Download and install the MRG/EC2 Enhanced RPMs

1. The MRG/EC2 Enhanced RPMs can be downloaded using `yum`. You will need to ensure that you are connected to the Red Hat Network.



Important

For further information on installing Red Hat Enterprise MRG components, see the *MRG Grid Installation Guide*.

2. On the submit machine, use `yum` to install the `condor-ec2-enhanced-hooks` package:

```
# yum install condor-ec2-enhanced-hooks
```

Configuring the submit machine

1. In order for the local pool to take advantage of the newly created MRG/EC2 Enhanced image, some changes need to be made to the configuration of a submit node in the pool. A sample configuration file for the submit machine is located at `/usr/share/doc/condor-ec2-enhanced-hooks-1.0/example/condor_config.example`. Copy the required parts of this file to the submit nodes local configuration file, and edit the following lines to include the AMI ID you received during the registration process:

```
set_amazonamiid = "ami-123a456b";
```

2. Specify the default settings for all routes, including instructions to remove a routed job if it is held or idle for over 6 hours:

```
JOB_ROUTER_DEFAULTS = \
[ \
  MaxIdleJobs = 10; \
  MaxJobs = 200; \
\
  set_PeriodicRemove = (JobStatus == 5 && \
    HoldReason != "Spooling input data files") || \
    (JobStatus == 1 && (CurrentTime - QDate) > 3600*6); \
  set_requirements = true; \
  set_WantAWS = false; \
]
```

3. Define each routes for sending jobs. Specify a name, a list of requirements and the amazon details:



Note

Just one route is shown here. The example configuration file at `/usr/share/doc/ec2-enhanced-hooks-1.0/example/condor_config.example` goes into further detail.

```
JOB_ROUTER_ENTRIES = \
[ GridResource = "condor localhost $(COLLECTOR_HOST)"; \
  Name = "Amazon Small"; \
  requirements=target.WantAWS is true && (target.Universe is vanilla || \
    target.Universe is 5) && (target.WantArch is "INTEL" || target.WantArch \
    is UNDEFINED) && (target.WantCpus <= 1 || target.WantCpus is UNDEFINED) \
    && (target.WantMemory < 1.7 || target.WantMemory is UNDEFINED) && \
    (target.WantDisk < 160 || target.WantDisk is UNDEFINED); \
  set_gridresource = "amazon"; \
]
```

```
set_amazonpublickey = "<path_to_AWS_public_key>"; \  
set_amazonprivatekey = "<path_to_AWS_private_key>"; \  
set_amazonaccesskey = "<path_to_AWS_access_key>"; \  
set_amazonsecretkey = "<path_to_AWS_secret_key>"; \  
set_rsapublickey = "<path_to_RSA_public_key>"; \  
set_amazoninstancetype = "ml.small"; \  
set_amazons3bucketname = "<S3_bucket_name>"; \  
set_amazonsqsqueue_name = "<SQS_queue_name>"; \  
set_amazonamiid = "<EC2_AMI_ID>"; \  
set_remote_jobuniverse = 5; \  
] \  

```

4. Add the **JOB_ROUTER** to the list of daemons to run:

```
DAEMON_LIST = $(DAEMON_LIST) JOB_ROUTER
```

5. Define the polling period for the job router. It is recommended that this value be set to a low value during testing, and a higher value when running on a large scale. This will ensure tests run faster, but prevent using too much CPU when in production:

```
JOB_ROUTER_POLLING_PERIOD = 10
```

6. Set the maximum number of history rotations:

```
MAX_HISTORY_ROTATIONS = 20
```

7. Configure the job router hooks:

```
JOB_ROUTER_HOOK_TRANSLATE_JOB = $(LIBEXEC)/hooks/hook_translate.py  
JOB_ROUTER_HOOK_UPDATE_JOB_INFO = $(LIBEXEC)/hooks/  
hook_retrieve_status.py  
JOB_ROUTER_HOOK_JOB_EXIT = $(LIBEXEC)/hooks/hook_job_finalize.py  
JOB_ROUTER_HOOK_JOB_CLEANUP = $(LIBEXEC)/hooks/hook_cleanup.py
```

8. Restart MRG Grid with the new configuration:

```
$ service condor restart  
Stopping condor daemon:      [ OK ]  
Starting condor daemon:      [ OK ]
```

Submitting a job to MRG/EC2 Enhanced

1. A job that uses MRG/EC2 Enhanced is similar to a usual vanilla universe job. However, some keys need to be added to the job submit file. This submit file will cause the job to be routed to the Amazon Small route using administrator defined credentials:

```

universe = vanilla
executable = /bin/date
output = date.out
log = ulog
requirements = Arch == "INTEL"
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_executable = false
+WantAWS = True
+WantArch = "INTEL"
+WantCPUs = 1
queue

```



Important

The **requirements** attribute for the job will need to be set to match the hardware of the AMI the job will run on. For example, if the submit machine is x86_64 and the requirements are not specified, then the above job will not execute because the Amazon Small AMI type is 32-bit, not 64-bit.

- The following fields are available for routing the job to the correct AMI. If only **wantAWS** is defined, then the job will be routed to the small AMI type by default.

WantAWS

Must be either *TRUE* or *FALSE*. Use EC2 for executing the job. Defaults to false

WantArch

Must be either *INTEL* or *X86_64*. Designates the architecture desired for the job. Defaults to Intel

WantCpus

Must be an integer. Designates the number of CPUs desired for the job

WantMemory

Must be a float. Designates the amount of RAM desired for the job, in gigabytes

WantDisk

Must be an integer. Designates the amount of disk space desired for the job, in gigabytes

- User credentials for accessing EC2 can be supplied for the submit machine by the site administrator. If this is not the case, the submit file can be used to supply the required information, by adding the following entries:

```

+AmazonAccessKey = "<path>/access_key"
+AmazonSecretKey = "<path>/secret_access_key"
+AmazonPublicKey = "<path>/cert.pem"
+AmazonPrivateKey = "<path>/pk.pem"
+RSAPublicKey = "<path>/rsa_key.pub"

```

These credentials will only be used if the submit machine does not already have credentials defined in `condor_config` for the route that the job will use.

Concurrency Limits

MRG Grid provides the ability to limit the number of jobs that run concurrently. *Concurrency limits* can be used to limit job access to software licences, database connections, shares of overall load on a server, or the number of concurrently run jobs by a particular user or group of users.

Configuring MRG Grid to operate with concurrency limits is achieved by specifying **concurrency_limits** in the job submit file. The limits referenced in the submit file are defined in the configuration file. A job submit file can also reference more than one limit. The **condor_negotiator** then uses this information when it matches the job to a resource. Firstly, it checks that the limits have not been reached, and then stores the job's limits in the machine ClassAd to which it has been matched.

Configuration variables for concurrency limits are located in the **condor_negotiator** daemon's configuration file. The important configuration variables for concurrency limits are:

*_LIMIT

In this case, the ***** is the name of the limit. This variable sets the allowable number of concurrent jobs for jobs that reference this limit in their submit file. Any number of ***_LIMIT** variables can be set, as long as they all have different names

CONCURRENCY_LIMIT_DEFAULT

All limits that are not specified with ***_LIMIT**, will use the default limit

This example demonstrates the use of the ***_LIMIT** and **CONCURRENCY_LIMIT_DEFAULT** configuration variables

In the following configuration file, **Y_LIMIT** is set to 2 and **CONCURRENCY_LIMIT_DEFAULT** to 1. In this case, any job that includes the line **concurrency_limits = y** in the submit file will have a limit of 2. All other jobs that have a limit other than **y** will be limited to 1:

```
CONCURRENCY_LIMIT_DEFAULT = 1
Y_LIMIT = 2
```

The ***_LIMIT** variable can also be set without the use of **CONCURRENCY_LIMIT_DEFAULT**. With the following configuration, any job that includes the line **concurrency_limits = x** in the submit file will have a limit of 5. All other jobs that have a limit other than **x** will not be limited:

```
X_LIMIT = 5
```

Example 11.1. Using *_LIMIT and CONCURRENCY_LIMIT_DEFAULT

Creating a job submit file with concurrency limits

1. The **concurrency_limits** attribute references the ***_LIMIT** variables:

```
universe = vanilla
executable = /bin/sleep
arguments = 28
concurrency_limits = Y, x, z
```

```
queue 1
```

2. When the job has been submitted, **condor_submit** will sort the given concurrency limits and convert them to lowercase:

```
$ condor_submit job
Submitting job(s).
1 job(s) submitted to cluster 28.

$ condor_q -long 28.0 | grep ConcurrencyLimits
ConcurrencyLimits = "x,y,z"
```

3. Concurrency limits can also be adjusted with **condor_config_val**. In this case, three configuration variables need to be set. Set the **ENABLE_RUNTIME_CONFIG** variable to *TRUE*:

```
ENABLE_RUNTIME_CONFIG = TRUE
```

Allow access from a specific machine to the **CONFIG** access level. This allows you to change the limit from that machine:

```
HOSTALLOW_CONFIG = $(CONDOR_HOST)
```

List the configuration variables that can be changed. The following example allows all limits to be changed, and new limits to be created:

```
NEGOTIATOR.SETTABLE_ATTRS_CONFIG = *_LIMIT
```

4. Once the configuration is set, change the limits from the shell prompt:

```
$ condor_config_val -negotiator -rset "X_LIMIT = 3"
```

5. After the limits have been changed, reconfigure the **condor_negotiator** to pick up the changes:

```
$ condor_reconfig -negotiator
```

6. Information about all concurrency limits can be viewed at the shell prompt by using the **condor_userprio** command with the **-l** option:

```
$ condor_userprio -l | grep ConcurrencyLimit
ConcurrencyLimit.p = 0
```

```
ConcurrencyLimit.q = 2  
ConcurrencyLimit.x = 6  
ConcurrencyLimit.y = 1  
ConcurrencyLimit.z = 0
```

This command displays the current number of jobs using each limit. In the example used above, 6 jobs using the X limit, 2 are using the Q limit, and 0 are using the Z and P limits. The limits with zero users are returned because they have been used at some point in the past. If a limit has been configured but never used, it will not appear in the list.



Note

If, for example, ten jobs are currently using the X limit, and **x_LIMIT** is changed to five, those ten jobs will continue to run. However, no new matches will be accepted against the X limit until the number of running jobs drops below five.

Dynamic provisioning

Dynamic provisioning, also referred to as *partitionable startd* or *dynamic slots*, allows users to mark slots as partitionable. This means that more than one job can occupy a single slot at any one time. Typically, slots have a fixed set of resources, such as associated CPUs, memory and disk space. By partitioning the slot, those resources become more flexible and able to be better utilized.



Important

Dynamic provisioning is a new feature that provides some powerful configuration possibilities and should be used with care. Specifically, while pre-emption 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 will be split up into 1GB dynamic slots. In this instance a job requiring 2GB of memory would be starved, and unable to run.

This example shows how more than one job can be matched to a single slot through dynamic provisioning.

In this example, Slot1 has the following resources:

- `cpu=10`
- `memory=10240`
- `disk=BIG`

JobA is allocated to the slot. JobA has the following requirements:

- `cpu=3`
- `memory=1024`
- `disk=10240`

The portion of the slot that is being used is referred to as Slot1.1, and the slot now 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 ... until the entire resources available have been consumed by jobs.

Example 12.1. Matching multiple jobs to a single slot

To enable dynamic provisioning, set the `SLOT_TYPE_X_PARTITIONABLE` configuration variable to `TRUE`. The `x` component of the variable changes, depending on which slot is being configured.

In a pool that uses dynamic provisioning, jobs can have extra desirable resources specified in their submit files:

- `request_cpus`
- `request_memory` (in megabytes)

- request_disk (in kilobytes)

This example gives a truncated job submit description file for use when submitting a job to a pool with dynamic provisioning.

```
JobA:
universe = vanilla
executable = ...
...
request_cpus = 3
request_memory = 1024
request_disk = 10240
...
queue
```

Example 12.2. Job submit file for a dyanmic provisioning pool

For each type of slot - the original, partionable slot and the new smaller, dynamic slots - an attribute is added to identify it. The original slot will have an attribute stating **PartitionableSlot=TRUE** and the dynamic slots will have an attribute **DynamicSlot=TRUE**. These attributes can 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. It will eventually show as having no available resources, which will prevent it being matched to new jobs. Because it has been effectively broken up into smaller slots, these will show as running jobs directly. These dynamic slots can also be pre-empted in the same way as ordinary slots.

Low-latency scheduling

Low-latency scheduling allows jobs to begin execution immediately, without going through the standard scheduling process. This decreases the amount of time before a job can begin execution, but bypasses the scheduling process. This can increase the possibility that a job will not be able to execute on the first node that tries to run it. This type of scheduling is performed using the MRG Messaging component of Red Hat Enterprise MRG, instead of the Condor daemons. The machines in the pool capable of executing jobs - execute nodes - communicate directly with a MRG Messaging broker. The advantage of this is that any machine capable of sending messages to the broker can submit jobs to the pool.



Note

For more information on MRG Messaging, the broker, and the AMQP protocol, see the *MRG Messaging User Guide*

Installing the condor-low-latency packages

1.



Important

You will require the MRG Messaging broker from the Red Hat Network in order to use low-latency scheduling. For instructions on downloading and configuring the MRG Messaging packages, see the *MRG Messaging Installation Guide*.

You will require the following packages, in addition to the MRG Messaging components:

- condor-low-latency
- condor-job-hooks
- condor-job-hooks-common

Use **yum** to install these components:

```
# yum install condor-low-latency
# yum install condor-job-hooks
# yum install condor-job-hooks-common
```

2. Configure MRG Grid to use the new job hooks by opening the **condor_config** file in your preferred text editor and adding the following lines:

```
# Startd hooks
LOW_LATENCY_HOOK_FETCH_WORK = $(LIBEXEC)/hooks/hook_fetch_work.py
LOW_LATENCY_HOOK_REPLY_FETCH = $(LIBEXEC)/hooks/hook_reply_fetch.py

# Starter hooks
```

```
LOW_LATENCY_JOB_HOOK_PREPARE_JOB = $(LIBEXEC)/hooks/hook_prepare_job.py
LOW_LATENCY_JOB_HOOK_UPDATE_JOB_INFO =
$(LIBEXEC)/hooks/hook_update_job_status.py
LOW_LATENCY_JOB_HOOK_JOB_EXIT = $(LIBEXEC)/hooks/hook_job_exit.py

STARTD_JOB_HOOK_KEYWORD = LOW_LATENCY
```

3. Set the **FetchWorkDelay** setting. This setting controls how often the condor-low-latency feature will look for jobs to execute, in seconds:

```
FetchWorkDelay = 10 * (Activity == "Idle")
STARTER_UPDATE_INTERVAL = 30
```

4. The daemon that controls the communication between MRG Messaging and MRG Grid is called the **caro** daemon. It can be configured by editing the file located at **/etc/opt/grid/carod.conf**. This file controls the active broker other options such as the exchange name, message queue and IP information.

The Condor job hooks are configured by editing the file located at **/etc/opt/grid/job-hooks.conf**. This file specifies the port and IP information that the job hooks can use to contact the **caro** daemon. The IP and port information in this file must match the information used in the **carod** configuration file.

5. When all the components are configured, start the MRG Messaging broker.

```
# service qpidd start
Starting qpidd daemon: [ OK ]
```

6. Start the Condor low latency daemon as a service:

```
# service condor-low-latency start
Starting condor-low-latency service: [ OK ]
```

7. Submitting a job using condor-low-latency scheduling is similar to submitting a regular Condor job, with the main difference being that instead of using a file for submission the job's attributes are defined in the application headers field of a MRG Messaging message. There are however some differences between the job description fields. To ensure the fields are correct, a normal Condor job submission file can be translated into the appropriate fields for the application headers by using the **condor_submit** command with the **-dump** option:

```
$ condor_submit myjob.submit -dump output_file
```

This command would produce a file named **output_file**. This file contains the information contained in the **myjob.submit** in a format suitable for placing directly into the the application header of a message. This method only works when queuing a single message at a time.

-
8. When submitting jobs in messages using this method, it is only possible to submit one job for every message. To submit multiple jobs of the same type, multiple messages - each containing one job - will need to be sent to the broker.

Any messages submitted this way must have a `reply-to` field set, or the jobs will not run. They must also include a unique message ID.

If data needs to be submitted with the job, it will need to be compressed and the archive placed in the body of the message. Similarly, results of the job will be placed in the body of the message to signify completion.

Application Program Interfaces (APIs)

The MRG Grid Web Service (WS) API provides a method of interaction for application developers. The MRG Grid daemons implement the SOAP (Simple Object Access Protocol) XML protocol. Job submission and management is provided through a web interface. A two phase commit mechanism is used to ensure reliability and fault-tolerance.

This chapter discusses the interaction between a client using the API and the `condor_schedd` and `condor_collector` daemons. It will explain the transactions and methods used for job submission, queue management and ClassAd management functions.

14.1. Using the MRG Grid API

The MRG Grid daemons communicate using the SOAP XML protocol. An application using this protocol needs to contain code that can handle the communication. The XML WSDL (Web Services Description Language) required by MRG Grid is included in the distribution, and can be found at `$(RELEASE_DIR)/lib/webservice`. The WSDL must be run through a toolkit to produce the language-specific routines required for communication. The application can be compiled as follows:

1. Condor must be configured to enable responses to SOAP calls. The WS interface listens on the `condor_schedd` daemon's command port. To obtain a list of all the `condor_schedd` daemons in the pool that have a WS interface, use this command at the shell prompt:

```
$ condor_status -schedd -constraint "HasSOAPInterface=?=TRUE"
```

2. To determine the port number to use:

```
$ condor_status -schedd -constraint "HasSOAPInterface=?=TRUE" -l |  
grep MyAddress
```

3. To authorize access to the SOAP client, it is also important to set the `ALLOW_SOAP` and `DENY_SOAP` configuration variables.

Transactions

All applications that use the API to interact with the `condor_schedd` daemon use *transactions*. The lifetime of a transaction is limited by the API, and can be further limited by the client application or the `condor_schedd` daemon.

Transactions are controlled by methods. They are initiated with a `beginTransaction()` method and completed with either a `commitTransaction()` or an `abortTransaction()` method.

Some operations will have access to more information when they are performed within a transaction. As an example of this, a `getJobAds()` query would have access to information about pending jobs within the transaction. Because these jobs are not committed they would not be visible outside of the transaction. However, transactions are designed to be *ACID*, or Atomic, Consistent, Isolated, and Durable. For this reason, information outside of a transaction should not be queried in order to make a decision within the transaction.

If required, the API can also accept null transactions. A null transaction can be created by inserting the programming language's equivalent of *null* in place of the transaction identifier. In a SOAP message, the following line achieves this:

```
<transaction xsi:type="ns1:Transaction" xsi:nil="true"/>
```

Submitting jobs

A job must be described with a ClassAd. The job ClassAd is then submitted to the `condor_schedd` within a transaction using the `submit()` method. To simplify the creation of a job ClassAd, the `createJobTemplate()` method can be called. This method returns a ClassAd structure that can then be modified to suit.



Important

For jobs that will be executed on Windows platforms, explicitly set the job ClassAd `NTDomain` attribute. The owner of the job will authenticate to this NT domain. This attribute is required but is not set by the `createJobTemplate()` function.

A necessary part of the job ClassAd are the `ClusterId` and `ProcId` attributes, which uniquely identify the cluster and the job. When the `newCluster()` method is called, it is assigned a `ClusterId`. Every job submitted is then assigned a `ProcId`, starting at 0 and incrementing by one for every job. When `newCluster()` is called again, it is assigned the next `ClusterId` and the job numbering starts again at 0.

This example demonstrates the *ClusterId* and *ProcId* attributes.

The following list contains an ordered set of method calls, showing the assigned *ClusterId* and *ProcId* values:

1. A call to `newCluster()` assigns a *ClusterId* of 6
2. A call to `newJob()` assigns a *ProcId* of 0 as this is the first job within the cluster
3. A call to `submit()` results in a job submission numbered 6.0
4. A call to `newJob()`, assigns a *ProcId* of 1
5. A call to `submit()` results in a job submission numbered 6.1
6. A call to `newJob()`, assigns a *ProcId* of 2
7. A call to `submit()` results in a job submission numbered 6.2
8. A call to `newCluster()`, assigns a *ClusterId* of 7
9. A call to `newJob()`, assigns a *ProcId* of 0 as this is the first job within the cluster.
10. A call to `submit()` results in a job submission numbered 7.0
11. A call to `newJob()` assigns a *ProcId* of 1
12. A call to `submit()` results in a job submission numbered 7.1

Example 14.1. Demonstrating the *ClusterId* and *ProcId* attributes

There is always a chance that a call to `submit()` will fail. Mostly this occurs when the job is in the queue but something required by the job has not been sent and the job will not be able to be run successfully. Sending the information required could potentially resolve this problem. To assist in determining what requirements a job has, the `discoverJobRequirements()` method can be called with a job ClassAd, and will return with a list of requirements for the job.

File transfer

Often, a job submission requires the job's executable and input files to be transferred from the machine where the application is running to the machine where the `condor_schedd` is running. The executable and input files must be sent directly to the `condor_schedd` daemon and placed in a spool location. This can be achieved with the `declareFile()` and `sendFile()` methods.

The `declareFile()` and `sendFile()` methods work together to transfer files to the `condor_schedd`. The `declareFile()` method causes `condor_schedd` to check if the file exists in the spool location. This prevents sending a file that already exists. The `sendFile()` method then sends the required file, or parts of a file, as base64 encoded data.

The `declareFile()` method requires the name of the file and its size in bytes. It also accepts optional information that relates to the hash (encryption) information for the file. When the hash type is specified as `NOHASH`, the `condor_schedd` daemon can not reliably determine if the file exists.

Retrieving files is most useful when a job is completed. When a job is completed and waiting to be removed, the `listSpool()` method provides a list of all the files for that job in the spool location. The `getFile()` method then retrieves a file.

Once the `closeSpool()` method has been called, the `condor_schedd` daemon removes the job from the queue and the spool files are no longer available. There is no requirement for the application to invoke the `closeSpool()` method, which results in jobs potentially remaining in the queue forever. The configuration variable `SOAP_LEAVE_IN_QUEUE` can help to mitigate this problem. It is a boolean value, and when it evaluates to *False*, the job will be reoved from the queue, and its information moved into the history log.

This example demonstrates the use of the `SOAP_LEAVE_IN_QUEUE` configuration variable

The following line inserted in the configuration file will result in a job being reoved from the queue once it has been completed for 24 hours:

```
SOAP_LEAVE_IN_QUEUE = ((JobStatus==4) && ((ServerTime - CompletionDate) <
(60 * 60 * 24)))
```

Example 14.2. Use of the `SOAP_LEAVE_IN_QUEUE` configuration variable

14.2. Methods

Method	Description	Parameters	Return Value
<code>beginTransaction</code>	Begin a transaction.	<i>duration</i> - The expected duration of the transaction.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the new transaction.
<code>commitTransaction</code>	Commits a transaction.	<i>transaction</i> - The transaction to be committed.	If the function succeeds, the return value is <i>SUCCESS</i> .
<code>abortTransaction</code>	Abort a transaction.	<i>transaction</i> - The transaction to be aborted.	If the function succeeds, the return value is <i>SUCCESS</i> .
<code>extendTransaction</code>	Request an extension in duration for a specific transaction.	<i>transaction</i> - The transaction to be extended and <i>duration</i> - The duration of the extension.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the transaction with the extended duration.

Table 14.1. Methods for transaction management

beginTransaction

Begin a transaction. For example:

```
StatusAndTransaction beginTransaction(int duration);
```

commitTransaction

Commits a transaction. For example:

```
Status commitTransaction(Transaction transaction);
```

abortTransaction

Abort a transaction. For example:

```
Status abortTransaction(Transaction transaction);
```

extendTransaction

Request an extension in duration for a specific transaction. For example:

```
StatusAndTransaction extendTransaction( Transaction transaction, int
    duration);
```

Example 14.3. Examples of methods for transaction management

Method	Description	Parameters	Return Value
submit	Submit a job.	<i>transaction</i> - The transaction in which the submission takes place; <i>clusterId</i> - The cluster identifier; <i>jobId</i> - The job identifier; <i>jobAd</i> - The ClassAd describing the job.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the transaction with the job requirements.
createJobTemplate	Request a job ClassAd, given some of the job requirements. This ClassAd will be suitable for use when submitting the job.	<i>clusterId</i> - The cluster identifier; <i>jobId</i> - The job identifier; <i>owner</i> - The name to be associated with the job; <i>type</i> - The universe under which the job will run; <i>command</i> - The command to execute once the job has started; <i>arguments</i> - The command-line arguments for <i>command</i> ;	If the function succeeds, the return value is <i>SUCCESS</i> .

Method	Description	Parameters	Return Value
		<i>requirements</i> - The requirements expression for the job. <i>type</i> can be any one of the following: <i>VANILLA</i> = 5, <i>SCHEDULER</i> = 7, <i>MPI</i> = 8, <i>GRID</i> = 9, <i>JAVA</i> = 10, <i>PARALLEL</i> = 11, <i>LOCALUNIVERSE</i> = 12 or <i>VM</i> = 13.	
discoverJobRequirements	Discover the requirements of a job, given a ClassAd.	<i>jobAd</i> - The ClassAd of the job.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the job requirements.

Table 14.2. Methods for job submission

submit

Submit a job. For example:

```
StatusAndRequirements submit(Transaction transaction, int clusterId, int
    jobId, ClassAd jobAd);
```

createJobTemplate

Request a job ClassAd, given some of the job requirements. This ClassAd will be suitable for use when submitting the job. For example:

```
StatusAndClassAd createJobTemplate(int clusterId, int jobId, String owner,
    UniverseType type, String command, String arguments, String requirements);
```

discoverJobRequirements

Discover the requirements of a job, given a ClassAd. For example:

```
StatusAndRequirements discoverJobRequirements( ClassAd jobAd);
```

Example 14.4. Examples of methods for job submission

Method	Description	Parameters	Return Value
declareFile	Declare a file to be used by a job.	<i>transaction</i> - The transaction in which the file is declared; <i>clusterId</i> - The cluster identifier; <i>jobId</i> - The identifier of the job that will use the file; <i>name</i> - The	If the function succeeds, the return value is <i>SUCCESS</i> .

Method	Description	Parameters	Return Value
		name of the file; <i>size</i> - The size of the file; <i>hashType</i> - The type of hash mechanism used to verify file integrity; <i>hash</i> - An optionally zero-length string encoding of the file hash. <i>hashType</i> can be either <i>NOHASH</i> or <i>MD5HASH</i>	
sendFile	Send a file that a job may use.	<i>transaction</i> - The transaction in which this file is send; <i>clusterId</i> - The cluster identifier; <i>jobId</i> - An identifier of the job that will use the file; <i>name</i> - The name of the file being sent; <i>offset</i> - The starting offset within the file being sent; <i>length</i> - The length from the offset to send; <i>data</i> - The data block being sent. This could be the entire file or a sub-section of the file as defined by offset and length.	If the function succeeds, the return value is <i>SUCCESS</i> .
getFile	Get a file from a job's spool.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster in which to search; <i>jobId</i> - The job identifier the file is associated with; <i>name</i> - The name of the file to retrieve; <i>offset</i> - The starting offset within the file being retrieved; <i>length</i> - The length from the offset to retrieve.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the file or a sub-section of the file as defined by offset and length.

Method	Description	Parameters	Return Value
closeSpool	Close a job's spool. All the files in the job's spool can be deleted.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster identifier which the job is associated with; <i>jobId</i> - The job identifier for which the spool is to be removed.	If the function succeeds, the return value is <i>SUCCESS</i> .
listSpool	List the files in a job's spool.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster in which to search; <i>jobId</i> - The job identifier to search for.	If the function succeeds, the return value is <i>SUCCESS</i> and contains a list of files and their respective sizes.

Table 14.3. Methods for file transfer

declareFile

Declare a file to be used by a job. For example:

```
Status declareFile(Transaction transaction, int clusterId, int jobId,
    String name, int size, HashType hashType, String hash);
```

sendFile

Send a file that a job may use. For example:

```
Status sendFile(Transaction transaction, int clusterId, int jobId, String
    name, int offset, Base64 data);
```

getFile

Get a file from a job's spool. For example:

```
StatusAndBase64 getFile(Transaction transaction, int clusterId, int jobId,
    String name, int offset, int length);
```

closeSpool

Close a job's spool. All the files in the job's spool can be deleted. For example:

```
Status closeSpool(Transaction transaction, int clusterId, int jobId);
```

listSpool

List the files in a job's spool. For example:

```
StatusAndFileInfoArray listSpool(Transaction transaction, int clusterId,
    int jobId);
```

Example 14.5. Examples of methods for file transfer

Method	Description	Parameters	Return Value
newCluster	Create a new job cluster.	<i>transaction</i> - The transaction in which this cluster is created.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the cluster ID.
removeCluster	Remove a job cluster, and all the jobs within it.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster to remove; <i>reason</i> - The reason for the removal.	If the function succeeds, the return value is <i>SUCCESS</i> .

Method	Description	Parameters	Return Value
newJob	Creates a new job within the most recently created job cluster.	<i>transaction</i> - The transaction in which this job is created; <i>clusterId</i> - The cluster identifier of the most recently created cluster.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the job ID.
removeJob	Remove a job, regardless of the job's state.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster identifier to search in; <i>jobId</i> - The job identifier to search for; <i>reason</i> - The reason for the release; <i>forceRemoval</i> - Set if the job should be forcibly removed.	If the function succeeds, the return value is <i>SUCCESS</i> .
holdJob	Put a job into the Hold state, regardless of the job's current state.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster in which to search; <i>jobId</i> - The job identifier to search for; <i>reason</i> - The reason for the release; <i>emailUser</i> - Set if the submitting user should be notified; <i>emailAdmin</i> - Set if the administrator should be notified; <i>systemHold</i> - Set if the job should be put on hold.	If the function succeeds, the return value is <i>SUCCESS</i> .
releaseJob	Release a job that has been in the Hold state.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster in which to search; <i>jobId</i> - The	If the function succeeds, the return value is <i>SUCCESS</i> .

Method	Description	Parameters	Return Value
		job identifier to search for; <i>reason</i> - The reason for the release; <i>emailUser</i> - Set if the submitting user should be notified; <i>emailAdmin</i> - Set if the administrator should be notified.	
getJobAds	Find an array of job ClassAds.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>constraint</i> - A string constraining the number of ClassAds to return.	If the function succeeds, the return value is <i>SUCCESS</i> and contains all job ClassAds matching the given constraint.
getJobAd	Finds a specific job ClassAd.	<i>transaction</i> - An optionally nullable transaction, this call does not need to occur in a transaction; <i>clusterId</i> - The cluster in which to search; <i>jobId</i> - The job identifier to search for.	If the function succeeds, the return value is <i>SUCCESS</i> and contains the requested job ClassAd.
requestReschedule	Request a condor_reschedule from the condor_schedd daemon.		If the function succeeds, the return value is <i>SUCCESS</i> .

Table 14.4. Methods for job management

newCluster

Create a new job cluster. For example:

```
StatusAndInt newCluster(Transaction transaction);
```

removeCluster

Remove a job cluster, and all the jobs within it. For example:

```
Status removeCluster(Transaction transaction, int clusterId, String  
    reason);
```

newJob

Creates a new job within the most recently created job cluster. For example:

```
StatusAndInt newJob(Transaction transaction, int clusterId);
```

removeJob

Remove a job, regardless of the job's state. For example:

```
Status removeJob(Transaction transaction, int clusterId, int jobId, String  
    reason, boolean forceRemoval);
```

holdJob

Put a job into the Hold state, regardless of the job's current state. For example:

```
Status holdJob(Transaction transaction, int clusterId, int jobId, string  
    reason, boolean emailUser, boolean emailAdmin, boolean systemHold);
```

releaseJob

Release a job that has been in the Hold state. For example:

```
Status releaseJob(Transaction transaction, int clusterId, int jobId, String  
    reason, boolean emailUser, boolean emailAdmin);
```

getJobAds

Find an array of job ClassAds. For example:

```
StatusAndClassAdArray getJobAds(Transaction transaction, String  
    constraint);
```

requestReschedule

Request a `condor_reschedule` from the `condor_schedd` daemon. For example:

```
Status requestReschedule();
```

Example 14.6. Examples of methods for job management

Method	Description	Parameters	Return Value
insertAd		<i>type</i> - The type of ClassAd to insert; <i>ad</i> - The ClassAd to insert. <i>type</i> can be any one of: <i>STARTD_AD_TYPE</i> , <i>QUILL_AD_TYPE</i> , <i>SCHEDD_AD_TYPE</i> , <i>SUBMITTOR_AD_TYPE</i> , <i>LICENSE_AD_TYPE</i> , <i>MASTER_AD_TYPE</i> , <i>CKPTSRVR_AD_TYPE</i> , <i>COLLECTOR_AD_TYPE</i> , <i>STORAGE_AD_TYPE</i> , <i>NEGOTIATOR_AD_TYPE</i> , <i>HAD_AD_TYPE</i> or <i>GENERIC_AD_TYPE</i> .	If the function succeeds, the return value is <i>SUCCESS</i> .
queryStartdAds	Search for condor_startd ClassAds.	<i>constraint</i> - A string constraining the number ClassAds to return.	A list of all the condor_startd ClassAds matching the given constraint.
queryScheddAds	Search for condor_schedd ClassAds.	<i>constraint</i> - A string constraining the number ClassAds to return.	A list of all the condor_schedd ClassAds matching the given constraint.
queryMasterAds	Search for condor_master ClassAds.	<i>constraint</i> - A string constraining the number ClassAds to return.	A list of all the condor_master ClassAds matching the given constraint.
querySubmittorAds	Search for submitter ClassAds.	<i>constraint</i> - A string constraining the number ClassAds to return.	A list of all the submitter ClassAds matching the given constraint.
queryLicenseAds	Search for license ClassAds.	<i>constraint</i> - A string constraining the number ClassAds to return.	A list of all the license ClassAds matching the given constraint.
queryStorageAds	Search for storage ClassAds.	<i>constraint</i> - A string constraining the number ClassAds to return.	A list of all the storage ClassAds matching the given constraint.
queryAnyAds	Search for any ClassAds.	<i>constraint</i> - A string constraining the number ClassAds to return.	A list of all the ClassAds matching the given constraint.

Table 14.5. Methods for ClassAd management

insertAd

For example:

```
Status insertAd(ClassAdType type, ClassAdStruct ad);
```

queryStartdAds

Search for **condor_startd** ClassAds. For example:

```
ClassAdArray queryStartdAds(String constraint);
```

queryScheddAds

Search for **condor_schedd** ClassAds. For example:

```
ClassAdArray queryScheddAds(String constraint);
```

queryMasterAds

Search for **condor_master** ClassAds. For example:

```
ClassAdArray queryMasterAds(String constraint);
```

querySubmittorAds

Search for submitter ClassAds. For example:

```
ClassAdArray querySubmittorAds(String constraint);
```

queryLicenseAds

Search for license ClassAds. For example:

```
ClassAdArray queryLicenseAds(String constraint);
```

queryStorageAds

Search for storage ClassAds. For example:

```
ClassAdArray queryLicenseAds(String constraint);
```

queryAnyAds

Search for any ClassAds. For example:

```
ClassAdArray queryAnyAds(String constraint);
```

Example 14.7. Examples of methods for ClassAd management

Method	Description	Return Value
<code>getVersionString</code>	Determine the Condor version.	Returns the Condor version as a string.
<code>getPlatformString</code>	Determine the platform information.	Returns the platform information as string.

Table 14.6. Methods for version information

getVersionString

Determine the Condor version. For example:

```
StatusAndString getVersionString();
```

getPlatformString

Determine the platform information. For example:

```
StatusAndString getPlatformString();
```

Example 14.8. Examples of methods for version information

Many methods return a status, [Table 14.7, “StatusCode return values”](#) lists the possible return values:

Value	Identifier	Definition
0	<i>SUCCESS</i>	No errors returned.
1	<i>FAIL</i>	An error occurred that is not specific to another error code
2	<i>INVALIDTRANSACTION</i>	No such transaction exists
3	<i>UNKNOWNCLUSTER</i>	The specified cluster is not the currently active one
4	<i>UNKNOWNJOB</i>	The specified job does not exist, or can not be found.
5	<i>UNKNOWNFILE</i>	The specified file does not exist, or can not be found.
6	<i>INCOMPLETE</i>	The request is incomplete.
7	<i>INVALIDOFFSET</i>	The specified offset is invalid.
8	<i>ALREADYEXISTS</i>	For this job, the specified file already exists

Table 14.7. StatusCode return values

Frequently Asked Questions

15.1. Installing MRG Grid

Q: How do I download MRG Grid

A: MRG Grid is available through the Red Hat Network. For full instructions on downloading and installing MRG Grid, read the *MRG Grid Installation Guide* available from the [Red Hat Enterprise MRG documentation page](#)¹.

Q: What platforms are supported?

A: MRG Grid is supported under most recent versions of both Red Hat Enterprise Linux and Microsoft Windows. Full information is available from the [Red Hat Enterprise MRG hardware page](#)². Note however that not all features are currently supported under Windows.

Q: Can I access the source code?

A: Yes! The source code is made available in the source RPM distributed by Red Hat. MRG Grid source code is distributed under the [Apache ASL 2.0 license](#)³.

15.2. Running MRG Grid jobs

Q: I receive too much email. What should I do with it all?

A: You should not ignore all the mail sent to you, but you can dramatically reduce the amount you get. When jobs are submitted, ensure they contain the following line:

```
Notification = Error
```

This will make sure that you only receive an email if an error has occurred. Note that this means you will not receive emails when a job completes successfully.

Q: My job starts but exits right away with *signal 9*. What's wrong?

A: This error occurs most often when a shared library is missing. If you know which file is missing, you can re-install it on all machines that might execute the job. Alternatively, re-link your program so that it contains all the information it requires.

Q: None or only some of my jobs are running, even though there's resources available in the pool. How can I fix this?

A: Firstly, you will need to discover where the problem lies. Try these steps to work out what is wrong:

1. Run `condor_q -analyze` and see what output it gives you
2. Look at the User Log file. This is the file that you specified as `log = path/to/filename.log` in the submit file. From this file you should be able to tell if the jobs are starting to run, or if they are exiting before they begin.

3. Look at the SchedLog on the submit machine after it has performed the negotiation for the user. If a user doesn't have a high enough priority to access more resources, then this log will contain a message that says *Lost priority, no more jobs*.
4. Check the ShadowLog on the submit machine for warnings or errors. If jobs are successfully being matched with machines, they still might be failing when they try to execute. This can be caused by file permission problems or similar errors.
5. Look at the NegotiatorLog during the negotiation for the user. Look for messages about priority or errors such as *No more machines*.

Another common problem that will stop jobs running is if the submit machine does not have adequate swap space. This will produce an error in the `SCHEDD_LOG` file:

```
[date] [time] Swap space estimate reached! No more jobs can be run!
[date] [time] Solution: get more swap space, or set RESERVED_SWAP = 0
[date] [time] 0 jobs matched, 1 jobs idle
```

The amount of swap space on the submit machine is calculated by the system. Serious errors can occur in a situation where a machine has a lot of physical memory and little or no swap space. Because physical memory is not considered, Condor might calculate that it has little or no swap space, and so it will not run the submitted jobs.

You can check how much swap space has been calculated as being available, by running the following command from the shell prompt:

```
$ condor_status -schedd [hostname] -long | grep VirtualMemory
```

If the value in the output is 0, then you will need to tell the system that it has some swap space. This can be done in two ways:

1. Configure the machine with some more actual swap space; or
2. Disable the check. Define the amount of reserved swap space for the submit machine as 0, and change the `RESERVED_SWAP` configuration variable to 0. You will need to perform `condor_restart` on the submit machine to pick up the changes.

-
- Q:** I submitted a job, but now my requirements expression has extra things in it that I didn't put there. How did they get there and why do I need them?
- A:** This occurs automatically, and are extensions that are required by Condor. This is a list of the things that are automatically added:
- If `arch` and `opsys` are not specified in the submit description file, they will be added. It will insert the same platform details as the machine from which the job was submitted.
 - The expression `Memory * 1024 > ImageSize` is automatically added. This makes sure that the job runs on a machine with at least as much physical memory as the memory footprint of the job.

-
- If the **Disk** \geq **DiskUsage** is not specified, it will be added. This makes sure that the job will only run on a machine with enough disk space for the job's local input and output.
 - A pool administrator can request that certain expressions are added to submit files. This is done using the following configuration variables:
 - **APPEND_REQUIREMENTS**
 - **APPEND_REQ_VANILLA**
 - **APPEND_REQ_STANDARD**

Q: What signals get sent to my jobs when they are pre-empted or killed, or when I remove them from the queue? Can I tell Condor which signals to send?

A: The signal jobs are sent can be set in the submit description file, by adding either of the following lines:

```
remove_kill_sig = SIGWHATEVER
```

```
kill_sig = SIGWHATEVER
```

If no signal is specified, the **SIGTERM** signal will be used. In the case of a hard kill, the **SIGKILL** signal is sent instead.

Q: Why does the time output from **condor_status** appear as [?????]?

A: Collecting time data from an entire pool of machines can cause errant timing calculations if the system clocks of those machines differ. If a time is calculated as negative, it will be displayed as [?????]. This can be fixed by synchronizing the time on all machines in the pool, using a tool such as NTP (Network Time Protocol).

Q: Condor commands are running very slowly. What is going on?

A: Some Condor commands will react slowly if they expect to find a **condor_collector** daemon, but can not find one. If you are not running a **condor_collector** daemon, change the **COLLECTOR_HOST** configuration variable to nothing:

```
COLLECTOR_HOST=
```

Q: If I submit jobs under NFS, they fail a lot. What's going on?

A: If the directory you are using when you run **condor_submit** is automounted under NFS (Network File System), Condor might try to unmount the volume before the job has completed.

To fix the problem, use the **initialdir** command in your submit description file with a reference to the stable access point. For example, if the NFS automounter is configured to mount a volume at **/a/myserver.company.com/vol1/user** whenever the directory **/home/user** is accessed, add this line to the submit description file:

```
initialdir = /home/user
```

Q: Why is my Java job completing so quickly?

A: The java universe executes the Java program's `main()` method and waits for it to return. When it returns, Condor considers your job to have been completed. This can happen inadvertently if the `main()` method is starting threads for processing. To avoid this, ensure you `join()` all threads spawned in the `main()` method.

15.3. Running MRG Grid on Windows platforms

Q: My pool uses a mixture of Unix/Linux and Windows machines. Will MRG Grid still work properly?

A: Yes! The central manager can be either Unix/Linux or Windows. Jobs can be submitted from either, and run on either platform.

Q: My Windows program works fine when executed on its own, but it does not work when submitted to the pool. What's going wrong?

A: Some Windows programs will not run properly because it can not find the `.dll` file that it depends on. To avoid this problem, try the following:

- Use a static link for the program, instead of a dynamic link
- Use a script for the job that will set up the necessary environment
- Use a machine where the job runs correctly, and submit the job with `getenv = true` in the submit description file to copy the current environment
- Send the required `.dll` files with the job by adding `transfer_input_files` to the submit description file

Q: Why won't the `condor_master` start, saying *In StartServiceCtrlDispatcher, Error number: 1063?*

A: Under Windows, the `condor_master` daemon is started as a service. To start the daemon, type the following command at the command prompt:

```
> net start condor
```

You can also start the service by going to the **Windows Control Panel**, opening the **Service Control Manager** and selecting the `condor_master` daemon.

Q: When I submit a job from a Windows machine, I receive an error about a credential. What does this mean?

A: Jobs submitted from a Windows machine require a stored password to perform some operations. If this password is not stored, it will give an error saying *ERROR: No credential stored for username@machinename.*

To store a password, type the following command at the command prompt:

```
> condor_store_cred add
```

Q: When a job executes on a Windows machine, if it has been submitted from a Unix/Linux machine, it doesn't work properly. How do I fix this?

A: This can happen sometimes if a file transfer has not been performed. To fix the problem, add the line **TRANSFER_FILES = ALWAYS** to the job submit description file.

Q: Why does my job start but then exit right away with *status 128*?

A: This happens when the machine executing the job is missing a required *.dll* file. To determine what *.dll* files your program requires, open Windows Explorer, right-click the program and select **Quickview**. Click **Import List** to see the required files. Once you know what files are required, include them and add the **TRANSFER_INPUT_FILES** line to the job submit file.

Q: Does the **USER_JOB_WRAPPER** configuration variable work on Windows machines?

A: No. This configuration variable does not work on Windows machines, due to differences in the way Windows and Unix/Linux handle batch scripts.

Q: Why do the Condor daemons exit with an error saying *10038 (WSAENOTSOCK)*?

A: This can be caused if a machine has installed a non-standard Winsock Layered Service Provider (LSP). These are commonly installed as part of anti-virus or other security-related software. There are freely available tools to detect and remove LSPs from Windows machines.

15.4. Grid computing

Q: My log files contain errors saying *PERMISSION DENIED*. What does that mean?

A: This can happen if the configuration variables **HOSTALLOW_*** and **HOSTDENY_*** are not configured correctly. Check these parameters and set **ALLOW_*** and **DENY_*** as appropriate.

Q: What happens if the central manager crashes?

A: If the central manager crashes, jobs that are already running will continue as normal. Queued jobs will remain in the queue but will not begin running until the central manager is restarted and begins matchmaking again.

Q: The condor daemons are running, but I get no output when I run **condor_status**. What is wrong?

A: Check the collector log. You should see a message similar to this:

```
DaemonCore: PERMISSION DENIED to host 128.105.101.15:9618 for command 0
(UPDATE_STARTD_AD)
```

This type of error is caused when permissions are configured correctly. Try the following:

- Ensure that DNS inverse lookup works on your machines (when you type in an IP address, your machine can find the domain name). If it is not working, either fix the DNS problem itself, or set the **DEFAULT_DOMAIN_NAME** setting in the configuration file
- Use numeric IP addresses instead of domain names when setting the **HOSTALLOW_WRITE** and **HOSTDENY_WRITE** configuration macros
- If the problem is caused by being too restrictive, try using wildcards when defining the address. For example, instead of using:

```
HOSTALLOW_WRITE = condor.your.domain.com
```

try using:

```
HOSTALLOW_WRITE = *.your.domain.com
```

Q: How do I stop my job moving to different CPUs?

A: You will need to define which slot you want the job to run on. You can do this using either **numactl** or **taskset**. If you are running jobs from within your own program, use **sched_setaffinity** and **pthread_{,attr_}setaffinity** to achieve the same result.

Q: I have a High Availability setup, but sometimes the **schedd** keeps on trying to start but exits with a *status 0*. Why is this happening?

A: In an High-Available Scheduler setup with 2 nodes (Node A and Node B), Condor will start on Node A and brings up the **schedd**, before it starts on Node B. On node B, the **schedd** continually attempts to start and exits with *status 0*.

This can be caused by the two nodes using different HA **schedd** names. In this case, the **schedd** on Node B will continually try to start, but will not be able to because of lock conflicts.

This problem can be solved by using the same name for the **schedd** on both nodes. This will make the **schedd** on Node B realize that one is already running, and it doesn't need to start. Change the **SCHEDD_NAME** configuration entry on both nodes so that the name is identical.

Note that this configuration will allow other schedulers to run on other nodes besides the HA **SCHEDD_NAME**. So you can have HA (on two nodes) and other **schedds** elsewhere.

More Information

Reporting Bugs

Follow these instructions to enter a bug report:

1. You will need a [Bugzilla](#)¹ account. You can create one at [Create Bugzilla Account](#)².
2. Once you have a Bugzilla account, log in and click on [Enter A New Bug Report](#)³.
3. You will need to identify the product (Red Hat Enterprise MRG), the version (1.1), and whether the bug occurs in the software (component=grid) or in the documentation (component=Grid_Installation_Guide).

Further Reading

- Red Hat Enterprise MRG and MRG Grid Product Information
 - <http://www.redhat.com/mrg>
- MRG Grid Installation Guide and other Red Hat Enterprise MRG manuals
 - http://redhat.com/docs/en-US/Red_Hat_Enterprise_MRG
- University of Wisconsin's Condor Manual
 - <http://www.cs.wisc.edu/condor/manual/>

Appendix A. Revision History

Revision 21 BZ #479198 BZ #473111	Mon Jan 12 2009	Lana Brindley lbrindle@redhat.com
Revision 20 BZ #479053	Wed Jan 7 2009	Lana Brindley lbrindle@redhat.com
Revision 19 BZ #477801 BZ #477805	Wed Jan 7 2009	Lana Brindley lbrindle@redhat.com
Revision 18 BZ #477070 Removed issuenum in Book_Info.xml Changed edition to 1	Mon Dec 22 2008	Michael Hideo mhideo@redhat.com
Revision 0.15 BZ #474939 BZ #474938	Mon Dec 8 2008	Lana Brindley lbrindle
Revision 0.14 Further minor updates	Fri Dec 5 2008	Lana Brindley lbrindle
Revision 0.13 Further minor updates Restructure of EC2 Chapter	Tue Nov 25 2008	Lana Brindley lbrindle
Revision 0.12 Minor updates prior to releasing document to Quality Engineering	Mon Nov 24 2008	Lana Brindley lbrindle
Revision 0.11 Completion of EC2 chapter	Mon Nov 24 2008	Lana Brindley lbrindle@redhat.com
Revision 0.10 Split EC2 chapter into EC2 and EC2 Enhanced - BZ #471695	Fri Nov 21 2008	Lana Brindley lbrindle@redhat.com
Revision 0.9	Thu Nov 20 2008	Lana Brindley lbrindle@redhat.com

Appendix A. Revision History

Added remote configuration chapter - BZ #471707

Revision 0.8 Wed Nov 19 2008 Lana Brindley lbrindle@redhat.com
Changes and updates arising from technical review

Revision 0.7 Fri Nov 7 2008 Lana Brindley lbrindle@redhat.com
Configuration

Revision 0.6 Mon Nov 3 2008 Lana Brindley lbrindle@redhat.com
Concurrency limits - BZ #459937
Dynamic provisioning - BZ #468942
Low-latency scheduling - BZ #454455
FAQs
More Information

Revision 0.5 Wed Oct 29 2008 Lana Brindley lbrindle@redhat.com
Added download and configuration information to EC2 chapter
APIs

Revision 0.4 Tue Oct 28 2008 Lana Brindley lbrindle@redhat.com
EC2
Removed future chapters from current build

Revision 0.3 Tue Oct 21 2008 Lana Brindley lbrindle@redhat.com
Policy Configuration
Virtual Machine Universe
High Availability

Revision 0.2 Wed Oct 1 2008 Lana Brindley lbrindle@redhat.com
Front matter
Preface
Overview
Configuration (not completed)
Jobs
Users
ClassAds
Policy Configuration (not completed)

Revision 0.1 Wed Aug 6 2008 Lana Brindley lbrindle@redhat.com
Initial Document Creation