globus gram job manager Reference Manual

10.17

Generated by Doxygen 1.4.4

Sat Feb 6 12:17:29 2010

Contents

1 GRAM Job Manager Reference Manual

The GRAM Job Manager program starts and monitors jobs on behalf of a GRAM client application. The job manager is typically started by the Gatekeeper program. It interfaces with a local scheduler to start jobs based on a job request RSL string.

- Job Manager Configuration
- RSL Validation File Format
- Job Execution Environment
- RSL Attributes
- Job Manager Scheduler Interface

2 globus gram job manager Page Index

2.1 globus gram job manager Related Pages

Here is a list of all related documentation pages:

3 globus gram job manager Page Documentation

3.1 Job Manager Configuration

The Job Manager is generally configured using the setup-globus-gram-job-manager setup script.

This section of the Job Manager manual describes all options which may be passed to the GRAM Job Manager in the configuration file `$GLOBUS_LOCATION/etc/globus-job-manager.conf`.

3.1.1 Configuration File Options

-save-logfile always|on-error|never
Create a file to store Job Manager log messages. This file will be created in the user's home directory. If the argument to this option is *on-error* , then the file will be removed if the job manager completes successfully. If the argument to this optionis *always* , the file will not be deleted by the job manager.

-k
Indicate that the job manager was built with the kerberos GSSAPI instead of GSI. This disables checks for a delegated GSI credential.

-home GLOBUS_LOCATION
Set the GLOBUS_LOCATION environment variable to the specified Globus-Location.

-type JOB_MANAGER_TYPE
Set the type of scheduler interface to use for this job manager. A similarly named Perl module in the `$GLOBUS_LOCATION/lib/perl/Globus/GRAM/JobManager/` directory is needed.

-history PATH
Set the path to the directory to store job history files.

-cache-location PATH
Set the path for the GASS cache. This path must be unique for each user. RSL substitutions (such as /builddir and mockbuild) may be used in this path. See RSL Substitions for a list of available RSL substitutions.

-extra-envvars VAR1,VAR2,...
The Job manager will attempt to get the listed VARs from its environment and set them the same for the job's environment. For example, -extra-envvars LD_LIBRARY_PATH,GLOBUS_TCP_PORT_RANGE

-scratch-dir-base PATH
Set the default scratch directory root for the job manager. Job-specific scratch directories will be created as a subdirectory of this directory. RSL substitutions (such as /builddir and mockbuild) may be used in this path. See RSL Substitions for a list of available RSL substitutions. If this option is not present in the configuration file, then the job manager will create scratch directories as subdirectories of the user's home directory.

-condor-arch ARCH
Set the condor architecture for this job manager to be the specified ARCH. This is required when the job manager type is *condor*.

-condor-os OSNAME
Set the condor operating system name for this job manager to be the specified OSNAME. This is required when the job manager type is *condor*.

-globus-gatekeeper-host HOST
Set the name of the gatekeeper host. This will be made available in the GLOBUS_GATEKEEPER_HOST RSL substitution.

-globus-gatekeeper-port PORT
Set the TCP port of the gatekeeper. This will be made available in the GLOBUS_GATEKEEPER_PORT RSL substitution.

-globus-gatekeeper-subject SUBJECT
Set the GSI subject name of the gatekeeper. This will be made available in the GLOBUS_GATEKEEPER_-SUBJECT RSL substitution.

-globus-host-manufacturer MANUFACTURER
Set the manufacturer name of the host machine. This will be made available in the GLOBUS_HOST_-MANUFACTURER RSL substitution.

-globus-host-cputype CPUTYPE
Set the cpu type of the host machine. This will be made available in the GLOBUS_HOST_CPUTYPE RSL substitution.

-globus-host-osname OSNAME
Set the operating system name of the host machine. This will be made available in the GLOBUS_HOST_-OSNAME RSL substitution.

-globus-host-osversion OSVERSION
Set the operating system version of the host machine. This will be made available in the GLOBUS_HOST_-OSVERSION RSL substitution.

-globus-tcp-port-range RANGE
Set the range of TCP port numbers which the job manager will use. This will also be made available in the GLOBUS_TCP_PORT_RANGE environment variable and RSL substitution.

-state-file-dir PATH
Set the path to store job manager state files (used for restarting a job manager which fails). If this is not set, then job state files will be stored in the `$GLOBUS_LOCATION/tmp/gram_job_state` directory.

-x509-cert-dir PATH
Set the path to the X.509 trusted certificate directory on the job execution hosts. If not present, then the trusted certificate directory used by the job manager (usually set by the Gatekeeper) will be used

-seg-module MODULE
Use the named module as a way to interact with the scheduler instead of polling for job status.

-audit-directory DIRECTORY
Store job auditing records in DIRECTORY. This directory should be sticky and group writable but not group readable. Audit records can be uploaded to a database by using the globus-gram-audit command.

-globus-toolkit-version VERSION-STRING
Use the string VERSION-STRING as the toolkit version in audit records.

-single
Use the single job manager per user/jobmanager type feature

3.2 RSL Attributes

This page contains a list of all RSL attributes which are supported by the GRAM Job Manager.

arguments
The command line arguments for the executable. Use quotes, if a space is required in a single argument.

count
The number of executions of the executable.

directory
Specifies the path of the directory the jobmanager will use as the default directory for the requested job.

dry_run
If dryrun = yes then the jobmanager will not submit the job for execution and will return success.

environment
The environment variables that will be defined for the executable in addition to default set that is given to the job by the jobmanager.

executable
The name of the executable file to run on the remote machine. If the value is a GASS URL, the file is transferred to the remote gass cache before executing the job and removed after the job has terminated.

file_clean_up
Specifies a list of files which will be removed after the job is completed.

file_stage_in
Specifies a list of ("remote URL" "local file") pairs which indicate files to be staged to the nodes which will run the job.

file_stage_in_shared
Specifies a list of ("remote URL" "local file") pairs which indicate files to be staged into the cache. A symlink from the cache to the "local file" path will be made.

file_stage_out
Specifies a list of ("local file" "remote URL") pairs which indicate files to be staged from the job to a GASS-compatible file server.

gass_cache
Specifies location to override the GASS cache location.

gram_my_job
Obsolete and ignored.

host_count
Only applies to clusters of SMP computers, such as newer IBM SP systems. Defines the number of nodes ("pizza boxes") to distribute the "count" processes across.

job_type
This specifies how the jobmanager should start the job. Possible values are single (even if the count $>$ 1, only start 1 process or thread), multiple (start count processes or threads), mpi (use the appropriate method (e.g. mpirun) to start a program compiled with a vendor-provided MPI library. Program is started with count nodes), and condor (starts condor jobs in the "condor" universe.)

library_path
Specifies a list of paths to be appended to the system-specific library path environment variables.

max_cpu_time
Explicitly set the maximum cputime for a single execution of the executable. The units is in minutes. The value will go through an atoi() conversion in order to get an integer. If the GRAM scheduler cannot set cputime, then an error will be returned.

max_memory
Explicitly set the maximum amount of memory for a single execution of the executable. The units is in Megabytes. The value will go through an atoi() conversion in order to get an integer. If the GRAM scheduler cannot set maxMemory, then an error will be returned.

max_time
The maximum walltime or cputime for a single execution of the executable. Walltime or cputime is selected by the GRAM scheduler being interfaced. The units is in minutes. The value will go through an atoi() conversion in order to get an integer.

max_wall_time
Explicitly set the maximum walltime for a single execution of the executable. The units is in minutes. The value will go through an atoi() conversion in order to get an integer. If the GRAM scheduler cannot set walltime, then an error will be returned.

min_memory
Explicitly set the minimum amount of memory for a single execution of the executable. The units is in Megabytes. The value will go through an atoi() conversion in order to get an integer. If the GRAM scheduler cannot set minMemory, then an error will be returned.

project
Target the job to be allocated to a project account as defined by the scheduler at the defined (remote) resource.

proxy_timeout
Obsolete and ignored. Now a job-manager-wide setting.

queue
Target the job to a queue (class) name as defined by the scheduler at the defined (remote) resource.

remote_io_url
Writes the given value (a URL base string) to a file, and adds the path to that file to the environment throught the GLOBUS_REMOTE_IO_URL environment variable. If this is specified as part of a job restart RSL, the job manager will update the file's contents. This is intended for jobs that want to access files via GASS, but the URL of the GASS server has changed due to a GASS server restart.

restart
Start a new job manager, but instead of submitting a new job, start managing an existing job. The job manager will search for the job state file created by the original job manager. If it finds the file and successfully reads it, it will become the new manager of the job, sending callbacks on status and streaming stdout/err if appropriate. It will fail if it detects that the old jobmanager is still alive (via a timestamp in the state file). If stdout or stderr was being streamed over the network, new stdout and stderr attributes can be specified in the restart RSL and the jobmanager will stream to the new locations (useful when output is going to a GASS server started by the client that's listening on a dynamic port, and the client was restarted). The new job manager will return a new contact string that should be used to communicate with it. If a jobmanager is restarted multiple times, any of the previous contact strings can be given for the restart attribute.

rsl_substitution
Specifies a list of values which can be substituted into other rsl attributes' values through the mechanism.

save_state
Causes the jobmanager to save it's job state information to a persistent file on disk. If the job manager exits or is suspended, the client can later start up a new job manager which can continue monitoring the job.

scratch_dir
Specifies the location to create a scratch subdirectory in. A SCRATCH_DIRECTORY RSL substitution will be filled with the name of the directory which is created.

stderr
The name of the remote file to store the standard error from the job. If the value is a GASS URL, the standard error from the job is transferred dynamically during the execution of the job.

stderr_position
Specifies where in the file remote standard error streaming should be restarted from. Must be 0.

stdin
The name of the file to be used as standard input for the executable on the remote machine. If the value is a GASS URL, the file is transferred to the remote gass cache before executing the job and removed after the job has terminated.

stdout
The name of the remote file to store the standard output from the job. If the value is a GASS URL, the standard output from the job is transferred dynamically during the execution of the job.

stdout_position
Specifies where in the file remote output streaming should be restarted from. Must be 0.

two_phase
Use a two-phase commit for job submission and completion. The job manager will respond to the initial job request with a WAITING_FOR_COMMIT error. It will then wait for a signal from the client before doing the actual job submission. The integer supplied is the number of seconds the job manager should wait before timing out. If the job manager times out before receiving the commit signal, or if a client issues a cancel signal, the job manager will clean up the job's files and exit, sending a callback with the job status as GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED. After the job manager sends a DONE or FAILED callback, it will wait for a commit signal from the client. If it receives one, it cleans up and exits as usual. If it times out and save_state was enabled, it will leave all of the job's files in place and exit (assuming the client is down and will attempt a job restart later). The timeoutvalue can be extended via a signal. When one of the following errors occurs, the job manager does not delete the job state file when it exits: GLOBUS_GRAM_PROTOCOL_ERROR_COMMIT_TIMED_OUT, GLOBUS_GRAM_PROTOCOL_ERROR_TTL_EXPIRED, GLOBUS_GRAM_PROTOCOL_ERROR_-JM_STOPPED, GLOBUS_GRAM_PROTOCOL_ERROR_USER_PROXY_EXPIRED. In these cases, it can not be restarted, so the job manager will not wait for the commit signal after sending the FAILED callback

username
Verify that the job is running as this user.

3.3 Job Manager Scheduler Interface

The GRAM Job Manager interfaces with the job filesystems and scheduler through scheduler-specific Perl modules.

GRAM provides several Perl modules which can be used to implement scheduler-specific interfaces to the GRAM Job Manager. These are:

Globus::GRAM::Error This module implements the GRAM error results as objects. Methods in this module will construct a GRAM error with the value matching the values in the GRAM Protocol library. A scheduler-specific JobManager module may return one of these objects from its methods to indicate errors to the Job Manager program.

Globus::GRAM::JobState This module defines the GRAM job state constants. A scheduler-specific JobManager module returns one of these values from its methods to indicate the managed job's current state.

Globus::GRAM::JobSignal This module defines the GRAM job signal constant values. The Job Manager uses these values to communicate which signal is being invoked in the manager's signal method.

Globus::GRAM::JobManager This module defines the actual implementatoin of the Job Manager scheduler interface. One writing a scheduler-specific GRAM interface will create a subclass of this object which overrides the default implementation's methods.

Globus::GRAM::JobDescription This module mimics the RSL job description using perl syntax. The job manager passes an object of this type to the JobManager modules's constructor. The job manager stores RSL and some configuration values in that JobDescription object. The manager accesses values stored in the JobDescription by invoking methods containing the RSL attribute's name (example: `$description->gram_my_job()`). Method names are handled as if they were based on the canonical RSL representation of the attribute name. For example, the **gram_my_job** may be equivalently referred to as **GramMyJob**, **grammyjob**, or **GRAM_My_Job**.

3.4 GRAM Job Manager Scheduler Tutorial

This tutorial describes the steps needed to build a GRAM Job Manager Scheduler interface package.

The audience for this tutorial is a person interested in adding support for a new scheduler interface to GRAM. This tutorial will assume some familiarty with GTP, autoconf, automake, and Perl. As a reference point, this tutorial will refer to the code in the LSF Job Manager package.

3.4.1 Writing a Scheduler Interface

This section deals with writing the perl module which implements the interface between the GRAM job manager and the local scheduler. Consult the Job Manager Scheduler Interface section of this manual for a more detailed reference on the Perl modules which are used here.

The scheduler interface is implemented as a Perl module which is a subclass of the Globus::GRAM::JobManager module. Its name must match the scheduler type string used when the service is installed. For the LSF scheduler, the name is *lsf*, so the module name is *Globus::GRAM::JobManager::lsf* and it is stored in the file `lsf.pm`. Though there are several methods in the JobManager interface, they only ones which absolutely need to be implemented in a scheduler module are submit, poll, cancel.

We'll begin by looking at the start of the lsf source module, lsf.in (the transformation to lsf.pm happens when the setup script is run. To begin the script, we import the GRAM support modules into the scheduler module's namespace, declare the module's namespace, and declare this module as a subclass of the Globus::GRAM::JobManager module. All scheduler packages will need to do this, substituting the name of the scheduler type being implemented where we see *lsf* below.

```
use Globus::GRAM::Error;
use Globus::GRAM::JobState;
use Globus::GRAM::JobManager;
use Globus::Core::Paths;

...

package Globus::GRAM::JobManager::lsf;

@ISA = qw(Globus::GRAM::JobManager);
```

Next, we declare any system-specifc values which will be substituted when the setup package scripts are run. In the LSF case, we need the know the paths to a few programs which interact with the scheduler:

```
my ($mpirun, $bsub, $bjobs, $bkill);

BEGIN
{
    $mpirun = '@MPIRUN@';
    $bsub   = '@BSUB@';
    $bjobs  = '@BJOBS@';
    $bkill  = '@BKILL@';
}
```

The values surrounded by the at-sign (such as `@MPIRUN@`) will be replaced by with the path to the named programs by the `find-lsf-tools` script described below.

3.4.1.1 Writing a constructor For scheduler interfaces which need to setup some data before calling their other methods, they can overload the `new` method which acts as a constructor. Scheduler scripts which don't need any per-instance initialization will not need to provide a constructor, the Globus::GRAM::JobManager constructor will do the job.

If you do need to overloaded this method, be sure to call the JobManager module's constructor to allow it to do its initialization, as in this example:

```
sub new
{
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new(@_);

    ## Insert scheduler-specific startup code here

    return $self;
}
```

The job interface methods are called with only one argument, the scheduler object itself. That object contains the a Globus::GRAM::JobDescription object (`$self->{JobDescription}`) which includes the values from the RSL string associated with the request, as well as a few extra values:

job_id The string returned as the value of JOB_ID in the return hash from submit. This won't be present for methods called before the job is submitted.

uniq_id A string associated with this job request by the job manager program. It will be unique for all jobs on a host for all time.

cache_tag The GASS cache tag related to this job submission. Files in the cache with this tag will be cleaned by the cleanup_cache() method.

Now, let's look at the methods which will interface to the scheduler.

3.4.1.2 Submitting Jobs All scheduler modules must implement the submit method. This method is called when the job manager wishes to submit the job to the scheduler. The information in the original job request RSL string is available to the scheduler interface through the `JobDescription` data member of it's hash.

For most schedulers, this is the longest method to be implemented, as it must decide what to do with the job description, and convert them to something which the scheduler can understand.

We'll look at some of the steps in the LSF manager code to see how the scheduler interface is implemented.

In the beginning of the submit method, we'll get our parameters and look up the job description in the manager-specific object:

```
sub submit
{
    my $self = shift;
    my $description = $self->{JobDescription};
```

Then we will check for values of the job parameters that we will be handling. For example, this is how we check for a valid job type in the LSF scheduler interface:

```
if(defined($description->jobtype())
{
    if($description->jobtype !~ /^(mpi|single|multiple)$/)
    {
        return Globus::GRAM::Error::JOBTYPE_NOT_SUPPORTED;
    }
    elsif($description->jobtype() eq 'mpi' && $mpirun eq "no")
    {
        return Globus::GRAM::Error::JOBTYPE_NOT_SUPPORTED;
    }
}
```

The lsf module supports most of the core RSL attributes, so it does more processing to determine what to do with the values in the job description.

Once we've inspected the JobDescription we'll know what we need to tell the scheduler about so that it'll start the job properly. For LSF, we will construct a job description script and pass that to the `bsub` command. This script is a bourne shell script with some special comments which LSF uses to decide what constraints to use when scheduling the job.

First, we'll open the new file, and write the file header:

```
    $lsf_job_script = new IO::File($lsf_job_script_name, '>');

    $lsf_job_script->print<<EOF;
#! /bin/sh
#
# LSF batch job script built by Globus Job Manager
#
EOF
```

Then, we'll add some special comments to pass job constraints to LSF:

```
if(defined($queue))
{
    $lsf_job_script->print("#BSUB -q $queue\n");
}
if(defined($description->project()))
{
    $lsf_job_script->print("#BSUB -P " . $description->project() . "\n");
}
```

Before we start the executable in the LSF job description script, we will quote and escape the job's arguments so that they will be passed to the application as they were in the job submission RSL string:

At the end of the job description script, we actually run the executable named in the JobDescription. For LSF, we support a few different job types which require different startup commands. Here, we will quote and escape the strings in the argument list so that the values of the arguments will be identical to those in the initial job request string. For this Bourne-shell syntax script, we will double-quote each argument, and escaping the backslash (\), dollar-sign ($), double-quote ("), and single-quote (') characters. We will use this new string later in the script.

```
    @arguments = $description->arguments();

    foreach(@arguments)
    {
        if(ref($_))
        {
            return Globus::GRAM::Error::RSL_ARGUMENTS;
        }
    }
    if($arguments[0])
    {
        foreach(@arguments)
        {
             $_ =~ s/\\/\\\\/g;
             $_ =~ s/\$/\\\$/g;
             $_ =~ s/"/\\\"/g;
             $_ =~ s/'/\\\'/g;

             $args .= '"' . $_ . '" ';
        }
    }
    else
    {
        $args = "";
    }
```

To end the LSF job description script, we will write the command line of the executable to the script. Depending on the job type of this submission, we will need to start either one or more instances of the executable, or the mpirun program which will start the job with the executable count in the `JobDescription`:

```
if($description->jobtype() eq "mpi")
{
    $lsf_job_script->print("$mpirun -np " . $description->count() . " ");

    $lsf_job_script->print($description->executable()
                           . " $args \n");
}
elsif($description->jobtype() eq 'multiple')
{
    for(my $i = 0; $i < $description->count(); $i++)
    {
        $lsf_job_script->print($description->executable() . " $args &\n");
    }
    $lsf_job_script->print("wait\n");
}
else
{
    $lsf_job_script->print($description->executable() . " $args\n");
}
```

Next, we submit the job to the scheduler. Be sure to close the script file before trying to redirect it into the submit command, or some of the script file may be buffered and things will fail in strange ways!

When the submission command returns, we check its output for the scheduler-specific job identifier. We will use this value to be able to poll or cancel the job.

The return value of the script should be either a GRAM error object or a reference to a hash of values. The Globus::GRAM::JobManager documentation lists the valid keys to that hash. For the submit method, we'll return the job identifier as the value of JOB_ID in the hash. If the scheduler returned a job status result, we could return

that as well. LSF does not, so we'll just check for the job ID and return it, or if the job fails, we'll return an error object:

```
    $lsf_job_script->close();

    $job_id = (grep(/is submitted/,
                  split(/\n/, `$bsub < $lsf_job_script_name`)))[0];
    if($? == 0)
    {
        $job_id =~ m/<([^>]*)>/;
        $job_id = $1;

        return { JOB_ID => $job_id };
    }

    return Globus::GRAM::Error::INVALID_SCRIPT_REPLY;
}
```

That finishes the submit method. Most of the functionality for the scheduler interface is now written. We just have a few more (much shorter) methods to implement.

3.4.1.3 Polling Jobs All scheduler modules must also implement the poll method. The purpose of this method is to check for updates of a job's status, for example, to see if a job has finished.

When this method is called, we'll get the job ID (which we returned from the submit method above) as well as the original job request information in the object's JobDescription. In the LSF script, we'll pass the job ID to the `bjobs` program, and that will return the job's status information. We'll compare the status field from the `bjobs` output to see what job state we should return.

If the job fails, and there is a way to determine that from the scheduler, then the script should return in its hash both

```
JOB_STATE => Globus::GRAM::JobState::FAILED
```

and

```
ERROR => Globus::GRAM::Error::<ERROR_TYPE>->value
```

Here's an excerpt from the LSF scheduler module implementation:

```
sub poll
{
    my $self = shift;
    my $description = $self->{JobDescription};
    my $job_id = $description->jobid();
    my $state;
    my $status_line;

    $self->log("polling job $job_id");

    # Get first line matching job id
    $_ = (grep(/$job_id/, `$bjobs $job_id 2>/dev/null`))[0];

    # Get 3th field (status)
    $_ = (split(/\s+/))[2];

    if(/PEND/)
    {
        $state = Globus::GRAM::JobState::PENDING;
    }
    elsif(/USUSP|SSUSP|PSUSP/)
    {
```

```
        $state = Globus::GRAM::JobState::SUSPENDED
    }
    ...
    return {JOB_STATE => $state};
}
```

3.4.1.4 Cancelling Jobs All scheduler modules must also implement the cancel method. The purpose of this method is to cancel a running job.

As with the `poll` method described above, this method will be given the job ID as part of the JobDescription object held by the manager object. If the scheduler interface provides feedback that the job was cancelled successfully, then we can return a JOB_STATE change to the FAILED state. Otherwise we can return an empty hash reference, and let the poll method return the state change next time it is called.

To process a cancel in the LSF case, we will run the bkill command with the job ID.

```
sub cancel
{
    my $self = shift;
    my $description = $self->{JobDescription};
    my $job_id = $description->jobid();

    $self->log("cancel job $job_id");

    system("$bkill $job_id >/dev/null 2>/dev/null");

    if($? == 0)
    {
        return { JOB_STATE => Globus::GRAM::JobState::FAILED }
    }
    return Globus::GRAM::Error::JOB_CANCEL_FAILED;

}
```

3.4.1.5 End of the script It is required that all perl modules return a non-zero value when they are parsed. To do this, make sure the last line of your module consists of:

```
1;
```

3.4.2 Setting up a Scheduler

Once we've written the job manager script, we need to get it installed so that the gatekeeper will be able to run our new service. We do this by writing a setup script. For LSF, we will write the script `setup-globus-job-manager-lsf.pl`, which we will list in the LSF package as the **Post_Install_Program**.

To set up the Gatekeeper service, our LSF setup script does the following:

1. Perform system-specific configuration.
2. Install the GRAM scheduler Perl module and register as a gatekeeper service.
3. **(Optional)** Install an RSL validation file defining extra scheduler-specific RSL attributes which the scheduler interface will support.
4. Update the GPT metadata to indicate that the job manager service has been set up.

3.4.2.1 System-Specific Configuration First, our scheduler setup script probes for any system-specific information needed to interface with the local scheduler. For example, the LSF scheduler uses the `mpirun`,

`bsub`, `bqueues`, `bjobs`, and `bkill` commands to submit, poll, and cancel jobs. We'll assume that the administrator who is installing the package has these commands in their path. We'll use an autoconf script to locate the executable paths for these commands and substitute them into our scheduler Perl module. In the LSF package, we have the `find-lsf-tools` script, which is generated during bootstrap by autoconf from the `find-lsf-tools.in` file:

```
## Required Prolog

AC_REVISION($Revision: 1.5 $)
AC_INIT(lsf.in)

# checking for the GLOBUS_LOCATION

if test "x$GLOBUS_LOCATION" = "x"; then
    echo "ERROR Please specify GLOBUS_LOCATION" >&2
    exit 1
fi

...

## Check for optional tools, warn if not found

AC_PATH_PROG(MPIRUN, mpirun, no)
if test "$MPIRUN" = "no" ; then
    AC_MSG_WARN([Cannot locate mpirun])
fi

...

## Check for required tools, error if not found

AC_PATH_PROG(BSUB, bsub, no)
if test "$BSUB" = "no" ; then
    AC_MSG_ERROR([Cannot locate bsub])
fi

...

## Required epilog - update scheduler specific module

prefix='$(GLOBUS_LOCATION)'
exec_prefix='$(GLOBUS_LOCATION)'
libexecdir=${prefix}/libexec

AC_OUTPUT(
    lsf.pm:lsf.in
)
```

If this script exits with a non-zero error code, then the setup script propagates the error to the caller and exits without installing the service.

3.4.2.2 Registering as a Gatekeeper Service Next, the setup script installs it's perl module into the perl library directory and registers an entry in the Globus Gatekeeper's service directory. The program `globus-job-manager-service` (distributed in the job manager program setup package) performs both of these tasks. When run, it expects the scheduler perl module to be located in the `$GLOBUS_LOCATION/setup/globus` directory.

```
$libexecdir/globus-job-manager-service -add -m lsf -s jobmanager-lsf;
```

3.4.2.3 Installing an RSL Validation File If the scheduler script implements RSL attributes which are not part of the core set supported by the job manager, it must publish them in the job manager's data directory. If the scheduler script wants to set some default values of RSL attributes, it may also set those as the default values in the validation file.

The format of the validation file is described in the RSL Validation File Format section of the documentation. The validation file must be named *scheduler-type*.rvf and installed in the `$GLOBUS_LOCATION/share/globus_gram_job_manager` directory.

In the LSF setup script, we check the list of queues supported by the local LSF installation, and add a section of acceptable values for the *queue* RSL attribute:

```
open(VALIDATION_FILE,
     ">$ENV{GLOBUS_LOCATION}/share/globus_gram_job_manager/lsf.rvf");

# Customize validation file with queue info
open(BQUEUES, "bqueues -w |");

# discard header
$_ = <BQUEUES>;
my @queues = ();

while(<BQUEUES>)
{
    chomp;

    $_ =~ m/^(\S+)/;

    push(@queues, $1);
}
close(BQUEUES);

if(@queues)
{
    print VALIDATION_FILE "Attribute: queue\n";
    print VALIDATION_FILE join(" ", "Values:", @queues);

}
close VALIDATION_FILE;
```

3.4.2.4 Updating GPT Metadata Finally, the setup package should create and finalize a `Grid::GPT::Setup`. The value of $package must be the same value as the gpt_package_metadata *Name* attribute in the package's metadata file. If either the `new()` or `finish()` methods fail, then it is considered good practice to clean up any files created by the setup script. From `setup-globus-job-manager-lsf.pl`:

```
my $metadata =
    new Grid::GPT::Setup(
        package_name => "globus_gram_job_manager_setup_lsf");

...

$metadata->finish();
```

3.4.3 Packaging

Now that we've written a job manager scheduler interface, we'll package it using GPT to make it easy for our users to build and install. We'll start by gathering the different files we've written above into a single directory `lsf`.

- lsf.in
- find-lsf-tools.in
- setup-globus-job-manager.pl

3.4.3.1 Package Documentation If there are any scheduler-specific options defined for this scheduler module, or if there any any optional setup items, then it is good to provide a documentation page which describes these. For LSF, we describe the changes since the last version of this package in the file `globus_gram_job_manager_-lsf.dox`. This file consists of a doxygen mainpage. See www.doxygen.org for information on how to write documentation with that tool.

3.4.3.2 configure.in Now, we'll write our configure.in script. This file is converted to the configure shell script by the bootstrap script below. Since we don't do any probes for compile-time tools or system characteristics, we just call the various initialization macros used by GPT, declare that we may provide doxygen documentation, and then output the files we need substitions done on.

```
AC_REVISION($Revision: 1.5 $)
AC_INIT(Makefile.am)

GLOBUS_INIT
AM_PROG_LIBTOOL

dnl Initialize the automake rules the last argument
AM_INIT_AUTOMAKE($GPT_NAME, $GPT_VERSION)

LAC_DOXYGEN("../", "*.dox")

GLOBUS_FINALIZE

AC_OUTPUT(
        Makefile
        pkgdata/Makefile
        pkgdata/pkg_data_src.gpt
        doxygen/Doxyfile
        doxygen/Doxyfile-internal
        doxygen/Makefile
)
```

3.4.3.3 Package Metadata Now we'll write our metadata file, and put it in the pkgdata subdirectory of our package. The important things to note in this file are the package name and version, the post_install_program, and the setup sections. These define how the package distribution will be named, what command will be run by `gpt-postinstall` when this package is installed, and what the setup dependencies will be written when the Grid::GPT::Setup object is finalized.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gpt_package_metadata SYSTEM "package.dtd">

<gpt_package_metadata Format_Version="0.02" Name="globus_gram_job_manager_setup_lsf" >

  <Aging_Version Age="0" Major="1" Minor="0" />
  <Description >LSF Job Manager Setup</Description>
  <Functional_Group >ResourceManagement</Functional_Group>
  <Version_Stability Release="Beta" />
  <src_pkg >

    <With_Flavors build="no" />
    <Source_Setup_Dependency PkgType="pgm" >
      <Setup_Dependency Name="globus_gram_job_manager_setup" >
        <Version >
          <Simple_Version Major="3" />
        </Version>
      </Setup_Dependency>
      <Setup_Dependency Name="globus_common_setup" >
        <Version >
          <Simple_Version Major="2" />
        </Version>
      </Setup_Dependency>
    </Source_Setup_Dependency>
```

```
    <Build_Environment >
      <cflags >@GPT_CFLAGS@</cflags>
      <external_includes >@GPT_EXTERNAL_INCLUDES@</external_includes>
      <pkg_libs > </pkg_libs>
      <external_libs >@GPT_EXTERNAL_LIBS@</external_libs>
    </Build_Environment>

    <Post_Install_Message >
      Run the setup-globus-job-manager-lsf setup script to configure an
      lsf job manager.
    </Post_Install_Message>

    <Post_Install_Program >
      setup-globus-job-manager-lsf
    </Post_Install_Program>

    <Setup Name="globus_gram_job_manager_service_setup" >
      <Aging_Version Age="0" Major="1" Minor="0" />
    </Setup>

  </src_pkg>

</gpt_package_metadata>
```

3.4.3.4 Automake Makefile.am The automake Makefile.am for this package is short because there isn't any compilation needed for this package. We just need to define what needs to be installed into which directory, and what source files need to be put inot our source distribution. For the LSF package, we need to list the `lsf.in`, `find-lsf-tools`, and `setup-globus-job-manager-lsf.pl` scripts as files to be installed into the setup directory. We need to add those files plus our documentation source file to the EXTRA_LIST variable so that they will be included in source distributions. The rest of the lines in the file are needed for proper interaction with GPT.

```
include $(top_srcdir)/globus_automake_pre
include $(top_srcdir)/globus_automake_pre_top

SUBDIRS = pkgdata doxygen

setup_SCRIPTS = \
    lsf.in \
    find-lsf-tools \
    setup-globus-job-manager-lsf.pl

EXTRA_DIST = $(setup_SCRIPTS) globus_gram_job_manager_lsf.dox

include $(top_srcdir)/globus_automake_post
include $(top_srcdir)/globus_automake_post_top
```

3.4.3.5 Bootstrap The final piece we need to write for our package is the `bootstrap` script. This script is the standard bootstrap script for a globus package, with an extra line to generate the `fine-lsf-tools` script using autoconf.

```
#!/bin/sh

# checking for the GLOBUS_LOCATION

if test "x$GLOBUS_LOCATION" = "x"; then
    echo "ERROR Please specify GLOBUS_LOCATION" >&2
    exit 1
fi

if [ ! -f ${GLOBUS_LOCATION}/libexec/globus-bootstrap.sh ]; then
    echo "ERROR: Unable to locate \${GLOBUS_LOCATION}/libexec/globus-bootstrap.sh"
    echo "       Please ensure that you have installed the globus-core package and"
```

```
    echo "       that GLOBUS_LOCATION is set to the proper directory"
    exit
fi

. ${GLOBUS_LOCATION}/libexec/globus-bootstrap.sh

autoconf find-lsf-tools.in > find-lsf-tools
chmod 755 find-lsf-tools

exit 0
```

3.4.4 Building, Testing, and Debugging

With this all done, we can now try to build our now package. To do so, we'll need to run

```
% ./bootstrap
% ./globus-build
```

If all of the files are written correctly, this should result in our package being installed into `$GLOBUS_LOCATION`. Once that is done, we should be able to run gpt-postinstall to configure our new job manager.

Now, we should be able to run the command

```
% globus-personal-gatekeeper -start -jmtype lsf
```

to start a gatekeeper configured to run a job manager using our new scripts. Running this will output a contact string (referred to as <contact-string> below), which we can use to connect to this new service. To do so, we'll run globus-job-run to submit a test job:

```
% globus-job-run <contact-string> /bin/echo Hello, LSF
Hello, LSF
```

3.4.4.1 When Things Go Wrong If the test above fails, or more complicated job failures are occurring, then you'lll have to debug your scheduler interface. Here are a few tips to help you out.

Make sure that your script is valid Perl. If you run

```
perl -I$GLOBUS_LOCATION/lib/perl \
    $GLOBUS_LOCATION/lib/perl/Globus/GRAM/JobManager/lsf.pm
```

You should get no output. If there are any diagnostics, correct them (in the lsf.in file), reinstall your package, and rerun the setup script.

Look at the `Globus Toolkit Error FAQ` and see if the failure is perhaps not related to your scheduler script at all.

Enable logging for the job manager. By default, the job manager is configured to log only when it notices a job failure. However, if your problem is that your script is not returning a failure code when you expect, you might want to enable logging always. To do this, modify the job manager configuration file to contain "-save-logfile always" in place of "-save-log on_error".

Adding logging messages to your script: the JobManager object implements a `log` method, which allows you to write messages to the job manager log file. Do this as your methods are called to pinpoint where the error occurs.

Save the job description file when your script is run. This will allow you to run the `globus-job-manager-script.pl` interactively (or in the Perl debugger). To save the job description file, you can do

```
$self->{JobDescription}->save("/tmp/job_description.$$");
```

in any of the methods you've implemented.

3.5 RSL Validation File Format

The idea behind the RSL Validation file is to provide a mechanism for job manager scheduler interfaces to implement RSL extensions, while having the job manager still be able to decide whether a job request contains a valid RSL string.

In addition to indicating what RSL attributes are valid, the RSL validation file contains information about when the RSL attribute may be used, it's default value, and the format of the attribute's value.

3.5.1 File Format

3.5.1.1 Comments The RSL validation file may contain comments. Comments are indicated in the file by a line beginning with the # character. Comments continue until end-of-line. Comments are discarded by the RSL Validation File parser.

Example:

```
# I am a comment. Ignore me.
```

3.5.1.2 ValidationRecords The RSL validition file consists of a set of *ValidationRecords*. Each *ValidationRecord* consists of a number of *ValidationProperties* associated with a single RSL attribute. *ValidationRecord*s are separated by a blank line in the RSL validation file.

3.5.1.3 Validation Properties A *ValidationProperty* is defined in the RSL validation file by a *PropertyName* and a *PropertyValue*. The syntax of a validation property is simply the *PropertyName* followed by a colon character, followed by a *PropertyValue*.

If a *PropertyValue* begins with a double-quote, then it may span multiple lines. Double-quote characters within a multiline *PropertyValue* must be escaped by a preceding backslash character. An unquoted *PropertyValue* ends at the first newline character.

Example:

```
Attribute: directory
Description: "Specifies the path of the directory the jobmanager will
              use as the default directory for the requested job."
```

3.5.1.4 Defined PropertyNames The following *PropertyName*s are understood by this version of the RSL validation file parser. Any unknown *PropertyName* will cause the *ValidationProprty* to be ignored by the parser.

Attribute
The name of the RSL parameter to which this record refers.

Description
A textual description of what the RSL parameter means.

Default
The default value of the RSL parameter if it is not found in the RSL. The default value is only used if the DefaultWhen value matches the current validation mode.

Values
A string containing whitespace-separated list of enumerated values which are valid for this RSL attribute. For example, for the "dryrun" parameter, this may be **"yes no"**.

RequiredWhen
Some subset of the "when strings" (see below), indicating when the RSL parameter is required for the RSL to be valid.

DefaultWhen
Some subset of the "when strings" (see below) indicating when then RSL's default value will be used if the RSL attribute is not present in the RSL.

ValidWhen
Some subset of the "when strings" (see below) indicating when then RSL attribute may be used.

3.5.1.5 "When" Strings

The set of when strings understood by the RSL validation file parser are

GLOBUS_GRAM_JOB_SUBMIT
The RSL attribute pertains to job submission.

GLOBUS_GRAM_JOB_MANAGER_RESTART
The RSL attribute pertains to job manager restart.

GLOBUS_GRAM_JOB_MANAGER_STDIO_UPDATE
The RSL attribute pertains to the STDIO_UPDATE signal.

3.6 Job Execution Environment

3.6.1 Environment Variables

The GRAM Job Manager provides a minimal environment for user's jobs. The following environment variables will be set by the job manager. Variables marked with an asterisk (∗) will only be set if a related job manager configuration option or RSL attribute is provided. Local schedulers may set additional environment variables.

HOME
The user's home directory.

LOGNAME
The user's login name.

X509_USER_PROXY
The path to the job manager's delegated credential. (GSI only).

GLOBUS_GRAM_JOB_CONTACT
The job manager's contact string for this job.

GLOBUS_LOCATION
The path to the Globus installation on the job manager host.

X509_CERT_DIR∗
The path to a trusted certificate directory. This variable will only be set if the -x509-cert-dir argument is given to the job manager.

GLOBUS_GASS_CACHE_DEFAULT∗
The path to the job's GASS cache (if the gass_cache RSL attribute is present).

GLOBUS_TCP_PORT_RANGE∗
A system-specific range of TCP ports which may be used by the job. Globus I/O will automatically honor this range. Only present if the related configuration option is present in the job manager configuration file.

GLOBUS_REMOTE_IO_URL*
The path to a file containing a URL string of a GASS server which the job may access (if the remote_io_url attribute is present).

GLOBUS_NEXUS_NO_GSI
Disables GSI in Nexus's TCP protocol, for compatibility with Nexus 4.6 and earlier.

3.6.2 RSL Substitions

In addition to the environment variables described above, a number of RSL substitutions are made available by the job manager. These substitions may be added to the environment by the job RSL if needed.

HOME
The user's home directory.

LOGNAME
The user's login name.

GLOBUS_ID
The subject name of the security credentials under which the job is running.

GLOBUS_GRAM_JOB_CONTACT
The job manager's contact string for this job.

GLOBUS_HOST_MANUFACTURER
The manufacturer part of the host configuration string (derived from config.guess)

GLOBUS_HOST_CPUTYPE
The CPU type part of the host configuration string (derived from config.guess)

GLOBUS_HOST_OSNAME
The operating system name part of the host configuration string (derived from config.guess)

GLOBUS_HOST_OSVERSION
The operating system version number part of the host configuration string (derived from config.guess)

GLOBUS_GATEKEEPER_HOST
The name of the host on which the gatekeeper is running.

GLOBUS_GATEKEEPER_PORT
The TCP port which on which the gatekeeper is accepting connections.

GLOBUS_GATEKEEPER_SUBJECT
The subject name of the security credentials under which the gatekeeper is running.

GLOBUS_LOCATION
The path to the Globus installation on the job manager host.

GLOBUS_CACHED_STDOUT
The name of the local file in the cache where stdout is being stored. This may be used as the value of the stdout RSL attribute to cause one copy of output to be stored in the cache. A stdio_update signal may be used to retrieve the output when the job is finishing.

GLOBUS_CACHED_STDERR

The name of the local file in the cache where stderr is being stored. This may be used as the value of the stderr RSL attribute to cause one copy of output to be stored in the cache. A stdio_update signal may be used to retrieve the output when the job is finishing.

SCRATCH_DIRECTORY

The path of the scratch directory for this job. (Only set if the scratch_dir RSL attribute is used).

GLOBUS_CONDOR_ARCH

The condor name of the architecture which the job manager is handling jobs for.

GLOBUS_CONDOR_OS

The condor name of the operating system which the job manager is handling jobs for.