

GT 4.2.1 GRAM2: Developer Guide

GT 4.2.1 GRAM2: Developer Guide

Table of Contents

1. Introduction	1
2. Public interface	2
1. Scheduler Event Generator	2
2. Client	2
3. GRAM Proctocol	2
3. Tutorials	3
1. GRAM Job Manager Scheduler Tutorial	3
4. Usage scenarios	17
5. Debugging	18
6. Troubleshooting	19
7. Related Documentation	20

Chapter 1. Introduction

There is no content available at this time.

Chapter 2. Public interface

1. Scheduler Event Generator

- [General information](#)¹
- [SEG protocol](#)²
- [SEG API](#)³
- [SEG Tests](#)⁴

2. Client

[Client API](#)⁵

3. GRAM Proctocol

- [General information](#)⁶
- [GRAM Protocol Functions](#)⁷
- [Job States](#)⁸
- [Signals](#)⁹
- [GRAM Errors](#)¹⁰
- [GRAM Protocol Message Format](#)¹¹

¹ http://www.globus.org/api/c-globus-4.2.1/globus_scheduler_event_generator/html/main.html

² http://www.globus.org/api/c-globus-4.2.1/globus_scheduler_event_generator/html/seg_protocol.html

³ http://www.globus.org/api/c-globus-4.2.1/globus_scheduler_event_generator/html/group_seg_api.html

⁴ http://www.globus.org/api/c-globus-4.2.1/globus_scheduler_event_generator_test/html/main.html

⁵ http://www.globus.org/api/c-globus-4.2.1/globus_gram_client/html/main.html

⁶ http://www.globus.org/api/c-globus-4.2.1/globus_gram_protocol/html/main.html

⁷ http://www.globus.org/api/c-globus-4.2.1/globus_gram_protocol/html/group_globus_gram_protocol_functions.html

⁸ http://www.globus.org/api/c-globus-4.2.1/globus_gram_protocol/html/group_globus_gram_protocol_job_state.html

⁹ http://www.globus.org/api/c-globus-4.2.1/globus_gram_protocol/html/group_globus_gram_protocol_job_signal.html

¹⁰ http://www.globus.org/api/c-globus-4.2.1/globus_gram_protocol/html/group_globus_gram_protocol_error.html

¹¹ http://www.globus.org/api/c-globus-4.2.1/globus_gram_protocol/html/globus_gram_protocol.html

Chapter 3. Tutorials

1. 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 familiarity with GTP, autoconf, automake, and Perl. As a reference point, this tutorial will refer to the code in the LSF Job Manager package.

1.1. 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](#)¹ 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-specific values which will be substituted when the setup package scripts are run. In the LSF case, we need to know the paths to a few programs which interact with the scheduler:

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

¹ #

```
}
```

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.

1.1.1. 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 overload 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.

1.1.2. 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.

1.1.3. 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};
}
```

1.1.4. 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;
}
```

1.1.5. 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;
```

1.2. 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.

1.2.1. 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: 1.2 $)
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.

1.2.2. Registering as a Gatekeeper Service

Next, the setup script installs its perl module into the perl library directory and registers an entry in the Globus Gatekeeper's service directory. The program `_globus-job-manager-service`² (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;
```

1.2.3. 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](#)³ 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";
}
```

² <http://www-unix.globus.org/api/c-globus-2.4/perl/globus-job-manager-service.html>

³ http://www-unix.globus.org/api/c-globus-2.4/globus_gram_job_manager/html/globus_gram_job_manager_rsl_validation_file.html#globus_gram_job_manager_rsl_validation_file

```
    print VALIDATION_FILE join(" ", "Values:", @queues);  
}  
close VALIDATION_FILE;
```

1.2.4. 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();
```

1.3. 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`

1.3.1. Package Documentation

If there are any scheduler-specific options defined for this scheduler module, or if there are 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.

1.3.2. `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 substitutions done on.

```
AC_REVISION($Revision: 1.2 $)  
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
)
```

1.3.3. 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 finalized⁴.

```
<?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>
```

⁴ http://www-unix.globus.org/api/c-globus-2.4/globus_gram_job_manager/html/globus_gram_job_manager_interface_tutorial.html#updating_gpt_metadata

```
<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>
```

1.3.4. 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 into our source distribution. For the LSF package, we need to list the `lsf.in`, `find-lsf-tools`, and `code>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
```

1.3.5. 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 `find-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
```

1.4. 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
% ./gpt-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 -jmttype 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

1.4.1. When Things Go Wrong

If the test above fails, or more complicated job failures are occurring, then you'll 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](#)⁵ 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.

⁵ http://www.globus.org/toolkit/docs/2.4/faq_errors.html

Chapter 4. Usage scenarios

There is no content available at this time.

Chapter 5. Debugging

There is no content available at this time.

Chapter 6. Troubleshooting

There is no content available at this time.

Chapter 7. Related Documentation

Information about other C-APIs of the GT can be found [here](#)¹

¹ <http://www.globus.org/api/c-globus-4.2.1/>