Introduction

This guide is intended to help a developer interact with GRAM5. It includes sections on implementing clients in C and implementing a Local Resource Manager interface, as well as an overview of concepts and APIs used to interact with GRAM.
# Table of Contents

I. Introduction .................................................................................................................................................. 1  
   1. Before you begin .................................................................................................................................. 3  
   2. GRAM5 Concepts for Developers ........................................................................................................ 5  
II. GRAM Client Developer’s Guide ............................................................................................................... 9  
   3. Basic GRAM Client Scenarios ............................................................................................................. 11  
   4. Advanced GRAM Client Scenarios ....................................................................................................... 24  
III. GRAM Server Developer’s Guide ............................................................................................................ 34  
   5. LRM Adapter Tutorial .......................................................................................................................... 36  
IV. GRAM5 Developer’s Reference ................................................................................................................ 61  
   6. APIs ..................................................................................................................................................... 63  
   7. RSL Specification v1.1 .......................................................................................................................... 83  
   8. Debugging .......................................................................................................................................... 92  
   9. Troubleshooting .................................................................................................................................. 94  
  10. Semantics and syntax of protocols ...................................................................................................... 113  
  11. Related Documentation ...................................................................................................................... 121  
  12. Internal Components ......................................................................................................................... 122  
Glossary ....................................................................................................................................................... 123  
Index ........................................................................................................................................................... 125
List of Figures

10.1. GRAM State Transitions ................................................................. 120
List of Tables

2.1. GRAM Contact String Types ................................................................. 5
6.1. GRAM Client APIs ............................................................................. 63
9.1. GRAM5 Errors .................................................................................. 100
10.1. GRAM Job States ........................................................................... 119
List of Examples

2.1. Gatekeeper Service Contact Examples ................................................................. 7
5.1. fake.rvf ........................................................................................................ 39
7.1. Quoted Literal Examples .................................................................................. 85
7.2. GRAM5 Job Request Examples ..................................................................... 90
Part I. Introduction
# Table of Contents

1. Before you begin ........................................................................................................... 3  
   1. Feature summary ........................................................................................................ 3  
   2. Tested platforms ........................................................................................................ 3  
   3. Backward compatibility summary ............................................................................. 4  
   4. Technology dependencies ......................................................................................... 4  
   5. Security Considerations ............................................................................................ 4  
2. GRAM5 Concepts for Developers ................................................................................ 5  
   1. Blocking and Nonblocking Function Variants .......................................................... 5  
   2. Service Contact Strings ............................................................................................. 5  
   3. Job State Callbacks and Polling ............................................................................... 7  
   4. Credential Management ............................................................................................ 7  
   5. RSL ............................................................................................................................. 8
Chapter 1. Before you begin

1. Feature summary

New Features new since 5.2.1:

• Improved memory management and process management.
• Improved scalability and reliability

Other Standard Supported Features

• Remote job execution and management
• Uniform and flexible interface to local resource managers
• File staging before and after job execution
• File and directory clean up after job termination
• Service auditing for each submitted

Removed Features

• Condor SEG module is no longer included. Its functionality has been moved into the core of the job manager program.

2. Tested platforms

Tested platforms for GRAM5:

• Linux
  • CentOS 4 x86_64
  • CentOS 5, 6 x86_64, i386
  • Fedora 15, 16, 17 x86_64, i386
  • Red Hat Enterprise Linux 5, 6 x86_64, i386
  • Scientific Linux 5, 6 x86_64, i386
  • Debian 6, 7 (testing) x86_64, i386
  • Ubuntu 10.04LTS, 10.10, 11.04, 11.10, 12.04LTS x86_64, i386
• Mac OS X
  • Mac OS X 10.7 (Lion)
• Solaris
  • Solaris 11
3. Backward compatibility summary

Protocol changes in GRAM since GT4 series:

- The GRAM5 service uses a superset of the GRAM2 protocol for communication between the client and service. The extensions supported in GRAM5 are implemented in such a way that they are ignored by GRAM2 services or clients. These extensions provide improved error messages and version detection.

- GRAM5 does not support task co-allocation using DUROC and its related protocols. Jobs submitted using DUROC directives will fail.

- GRAM5 does not support file streaming. The standard output and standard error streams are sent after the job completes instead of during execution. As a special case, support for the Condor grid monitor program implements a small subset of the streaming capabilities of GRAM2 in GT 4.2.x.

4. Technology dependencies

GRAM depends on the following GT components:

- Globus Common
- GSI C
- GridFTP server

5. Security Considerations

5.1. Gatekeeper Security Considerations

GRAM5 runs different parts of itself under different privilege levels. The globus-gatekeeper runs as root, and uses its root privilege to access the host’s private key. It uses the grid map file to map Grid Certificates to local user ids and then uses the setuid() function to change to that user and execute the globus-job-manager program.

5.2. Job Manager Security Considerations

The globus-job-manager program runs as a local non-root account. It receives a delegated limited proxy certificate from the GRAM5 client which it uses to access Grid storage resources via GridFTP and to authenticate job signals (such as client cancel requests), and send job state callbacks to registered clients. This proxy is generally short-lived, and is automatically removed by the job manager when the job completes.

The globus-job-manager program uses a publicly-writable directory for job state files. This directory has the sticky bit set, so users may not remove other users files. Each file is named by a UUID, so it should be unique.

5.3. Fork SEG Module Security Considerations

The Fork Scheduler Event Generator module uses a globally writable file for job state change events. This is not recommended for production use.
Chapter 2. GRAM5 Concepts for Developers

1. Blocking and Nonblocking Function Variants

In the GRAM Client API, all functions that involve sending messages over the network have both blocking and non-blocking variants. These are useful in different programming situations.

The blocking variants, such as the `globus_gram_client_job_request` function require less application code, but will prevent subsequent instructions from executing until the request has been sent and the reply parsed. In a non-threaded environment, other callback functions registered with the Globus event driver may be invoked while the blocking function is running. In a threaded environment, other events may occur in other threads while the function is blocking, but the current thread will be blocked until the response is parsed.

The nonblocking variants, such as `globus_gram_client_register_job_request` require the application to include a callback function which will be called by the Globus event driver when the reply has been parsed. In a non-threaded environment, applications must poll the event driver using functions from the `globus_poll` or `globus_cond_wait` families of functions. In a threaded environment, the callback function may be invoked in another thread than the one calling the non-blocking function, even before the non-blocking function has returned. Application writers must be careful in using synchronization primitives such as `globus_mutex_t` and `globus_cond_t` when using non-blocking functions.

An application writer should use the non-blocking variants if the application will be submitting many jobs concurrently or requires custom network or security attributes. Using the non-blocking variants allows the Globus event driver to better schedule network I/O in these cases.

2. Service Contact Strings

GRAM uses three types of contact strings to describe how to contact different services. These service contacts are:

**Table 2.1. GRAM Contact String Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gatekeeper Service Contact</td>
<td>This string describes how to contact a gatekeeper service. It is used to submit jobs, send &quot;ping&quot; requests to determine if a service is properly deployed, and version requests to determine what version of the software is deployed. Full details of the syntax of this contact is located in the next section.</td>
</tr>
<tr>
<td>Callback Contact</td>
<td>This string is an HTTPS URL that is an endpoint for GRAM job state callbacks. An https message is posted to this address when the Job Manager detects a job state change.</td>
</tr>
<tr>
<td>Job Contact</td>
<td>This string is an HTTPS URL that is an endpoint for contacting an existing GRAM job. An https message is posted to this address to cancel, signal, or query a GRAM job.</td>
</tr>
</tbody>
</table>
2.1. Resource Names

In GRAM5, a Gatekeeper Service Contact contains the host, port, service name, and service identity required to contact a particular GRAM service. For convenience, default values are used when parts of the contact are omitted. An example of a full gatekeeper service contact is grid.example.org:2119/jobmanager:/C=US/O=Example/OU=Grid/CN=host/grid.example.org.

The various forms of the resource name using default values follow:

- HOST
- HOST:PORT
- HOST:PORT/SERVICE
- HOST/SERVICE
- HOST:/SERVICE
- HOST:PORT:SUBJECT
- HOST/SERVICE:SUBJECT
- HOST:/SERVICE:SUBJECT
- HOST:PORT/SERVICE:SUBJECT

Where the various values have the following meaning:

**HOST** Network name of the machine hosting the service.

**PORT** Network port number that the service is listening on. If not specified, the default of 2119 is used.

**SERVICE** Path of the service entry in $GLOBUS_LOCATION/etc/grid-services. If not specified, the default of jobmanager is used.

**SUBJECT** X.509 identity of the credential used by the service. If not specified, the default of host@HOST is used.
Example 2.1. Gatekeeper Service Contact Examples

The following strings all name the service grid.example.org:2119/jobmanager:/C=US/O=Example/OU=Grid/CN=host/grid.example.org using the formats with the various defaults described above.

- grid.example.org
- grid.example.org:2119
- grid.example.org:2119/jobmanager
- grid.example.org/jobmanager
- grid.example.org:/jobmanager
- grid.example.org:2119:/C=US/O=Example/OU=Grid/CN=host/grid.example.org
- grid.example.org/jobmanager:/C=US/O=Example/OU=Grid/CN=host/grid.example.org
- grid.example.org:/jobmanager:/C=US/O=Example/OU=Grid/CN=host/grid.example.org
- grid.example.org:2119/jobmanager:/C=US/O=Example/OU=Grid/CN=host/grid.example.org

3. Job State Callbacks and Polling

GRAM clients and learn about a job's state in two ways: by registering for job state callbacks and by polling for status. These two methods have different performance characteristics and costs.

In order to receive job state callbacks, a client application must create an HTTPS listener using the globus_gram_client_callback_allow or globus_gram_client_info_callback_allow functions. A non-threaded application must then periodically call a function from either the globus_cond_wait or globus_poll families in order to process the job state callbacks. Additionally, the network must be configured to allow the GRAM job manager to send messages to the port that the client is listening on. This may be difficult if there is a firewall between the client and service.

The GRAM service initiates the job state callbacks, and thus they are usually sent very shortly after the job state changes, so clients can be notified about the state changes quickly.

In order to poll for job states, a client can call either the blocking or nonblocking variant of the globus_gram_client_job_status or globus_gram_client_job_status_with_info functions. These functions require that the network be configured to allow the client to contact the network port that the GRAM service is listening on (the Job Contact).

The client initiates these polling operations, so they are only as accurate as the polling frequency of the client. If the client polls very often, it will receive job state changes more quickly, at the risk of increasing the computing and network cost of both the client and service.

4. Credential Management

The GRAM5 protocols all use GSSAPIv2 abstractions to provide authentication and authorization. By default, GRAM uses an SSL-based GSSAPI for its security.
The client delegates a credential to the gatekeeper service after authentication, and the GRAM job manager service uses this delegated credential as both a job-specific credential and for subsequent communication with GRAM clients.

If a client or clients submit multiple jobs to a gatekeeper service, they will eventually all be handled by a single job manager process. This process will use whichever delegated credential will remain valid the longest for accepting new connections and connecting to clients to send job state callbacks. When a client delegates a new credential to a job, this credential may also be used as the job manager's credential for future connections.

5. RSL

GRAM5 jobs are described using the RSL language. The GRAM client API submits jobs using the string representation of the RSL, rather than the RSL parse tree. Clients can, if they need to modify or construct RSL at runtime, use the functions in the RSL API to do so.
Part II. GRAM Client Developer's Guide
# Table of Contents

3. Basic GRAM Client Scenarios ........................................................................................................ 11
   1. "Ping" a Job Manager ........................................................................................................ 11
   2. Check a Job Manager Version ......................................................................................... 12
   3. Submitting a Job ................................................................................................................ 14
   4. Submitting a Job and Processing Job State Callbacks ....................................................... 16
   5. Polling Job Status ............................................................................................................. 19
   6. Canceling a Job ................................................................................................................ 21
   7. Refreshing Job Credential ............................................................................................... 22

4. Advanced GRAM Client Scenarios .......................................................................................... 24
   1. Non-blocking Job Submission ........................................................................................ 24
   2. Custom Security Attributes ........................................................................................... 27
   3. Modifying RSL ................................................................................................................ 30
Chapter 3. Basic GRAM Client Scenarios

This chapter contains a series of examples demonstrating how to use different features of the GRAM APIs to interact with the GRAM service. These examples can be compiled by using GNU make with the makefile from Makefile.examples.

1. "Ping" a Job Manager

This example shows how to use a gatekeeper "ping" request to determine if a service is running and if the client is authorized to contact it. It takes a gatekeeper service contact as its only command-line option. The source to this example\(^1\) can be downloaded.

```c
/*
   * These headers contain declarations for the globus_module functions
   * and GRAM Client API functions
   */
#include "globus_common.h"
#include "globus_gram_client.h"
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int rc;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT\n", argv[0]);
        rc = 1;
        goto out;
    }

    printf("Pinging GRAM resource: %s\n", argv[1]);

    /*
     * Always activate the GLOBUS_GRAM_CLIENT_MODULE prior to using any
     * functions from the GRAM Client API or behavior is undefined.
     */
    rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error activating %s because %s (Error %d)\n",
            GLOBUS_GRAM_CLIENT_MODULE->module_name,
            globus_gram_client_error_string(rc),
            rc);
        goto out;
    }

    gram_ping_example.c
```

\(^1\) gram_ping_example.c
Basic GRAM Client Scenarios

1. Ping the Service

* Ping the service passed as our first command-line option. If successful, * this function will return GLOBUS_SUCCESS, otherwise an integer * error code.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rc;
    globus_hashtable_t extensions = NULL;
    globus_gram_protocol_extension_t * extension_value;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT\n", argv[0]);
    } else {
        rc = globus_gram_client_ping(argv[1]);
        if (rc != GLOBUS_SUCCESS) {
            fprintf(stderr, "Unable to ping service at %s because %s (Error %d)\n", argv[1], globus_gram_client_error_string(rc), rc);
        } else {
            printf("Ping successful\n");
        }
    }
    rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
    return rc;
}
```

2. Check a Job Manager Version

This example shows how to use the "version" command to determine what software version a gatekeeper service is running. The source to this example can be downloaded.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int rc;
    globus_hashtable_t extensions = NULL;
    globus_gram_protocol_extension_t * extension_value;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT\n", argv[0]);
    } else {
        rc = globus_gram_client_ping(argv[1]);
        if (rc != GLOBUS_SUCCESS) {
            fprintf(stderr, "Unable to ping service at %s because %s (Error %d)\n", argv[1], globus_gram_client_error_string(rc), rc);
        } else {
            printf("Ping successful\n");
        }
    }
    rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
    return rc;
}
```

2. Check a Job Manager Version

This example shows how to use the "version" command to determine what software version a gatekeeper service is running. The source to this example can be downloaded.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int rc;
    globus_hashtable_t extensions = NULL;
    globus_gram_protocol_extension_t * extension_value;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT\n", argv[0]);
    } else {
        rc = globus_gram_client_ping(argv[1]);
        if (rc != GLOBUS_SUCCESS) {
            fprintf(stderr, "Unable to ping service at %s because %s (Error %d)\n", argv[1], globus_gram_client_error_string(rc), rc);
        } else {
            printf("Ping successful\n");
        }
    }
    rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
    return rc;
}
```
rc = 1;
goto out;
}
printf("Checking version of GRAM resource: %s\n", argv[1]);

/*
 * Always activate the GLOBUS_GRAM_CLIENT_MODULE prior to using any
 * functions from the GRAM Client API or behavior is undefined.
 */
rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Error activating %s because %s (Error %d)\n",
            GLOBUS_GRAM_CLIENT_MODULE->module_name,
            globus_gram_client_error_string(rc),
            rc);
    goto out;
}

/*
 * Contact the service passed as our first command-line option and perform
 * a version check. If successful,
 * this function will return GLOBUS_SUCCESS, otherwise an integer
 * error code. Old versions of the job manager will return
 * GLOBUS_GRAM_PROTOCOL_ERROR_HTTP_UNPACK_FAILED as they do not support
 * the version operation.
 */
rc = globus_gram_client_get_jobmanager_version(argv[1], &extensions);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Unable to get service version from %s because %s "
              "(Error %d)\n",
            argv[1], globus_gram_client_error_string(rc),
            rc);
}
else
{
    /* The version information is returned in the extensions hash table */
    extension_value = globus_hashtable_lookup(
                    &extensions,
                    "toolkit-version");

    if (extension_value == NULL)
    {
        printf("Unknown toolkit version\n");
    }
    else
    {
        printf("Toolkit Version: %s\n", extension_value->value);
    }

    extension_value = globus_hashtable_lookup(
                    &extensions,
                    "version");
if (extension_value == NULL)
{
    printf("Unknown package version\n");
}
else
{
    printf("Package Version: %s\n", extension_value->value);
}
/* Free the extensions hash and its values */
globus_gram_protocol_hash_destroy(&extensions);
/*
 * Deactivating the module allows it to free memory and close network
 * connections.
 */
rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
out:
    return rc;
}
/* End of gram_version_example.c */

3. Submitting a Job

This example shows how to submit a job to a GRAM service. The source to this example\(^3\) can be downloaded.

/*
 * These headers contain declarations for the globus_module functions
 * and GRAM Client API functions
 */
#include "globus_common.h"
#include "globus_gram_client.h"

#include <stdio.h>

int main(int argc, char *argv[]) {  
    int rc;
    char * job_contact = NULL;

    if (argc != 3)  
    {  
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT RSL\n", argv[0]);
        rc = 1;
        goto out;
    }

    printf("Submitting job to GRAM resource: %s\n", argv[1]);

    */

\(^3\) gram_submit_example.c
* Always activate the GLOBUSGRAM_CLIENT_MODULE prior to using any
* functions from the GRAM Client API or behavior is undefined.
*/
rc = globus_module_activate(GLOBUSGRAM_CLIENT_MODULE);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Error activating %s because %s (Error %d)\n",
        GLOBUSGRAM_CLIENT_MODULE->module_name,
        globus_gram_client_error_string(rc),
        rc);
    goto out;
}
/*
* Submit the job request to the service passed as our first command-line
* option. If successful, this function will return GLOBUS_SUCCESS,
* otherwise an integer error code.
*/
rc = globus_gram_client_job_request(
    argv[1], argv[2], 0, NULL, &job_contact);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Unable to submit job to %s because %s (Error %d)\n",
        argv[1], globus_gram_client_error_string(rc), rc);
    if (job_contact != NULL)
    {
        printf("Job Contact: %s\n", job_contact);
    }
}
else
{
    /* Display job contact string */
    printf("Job submit successful: %s\n", job_contact);
}
if (job_contact != NULL)
{
    free(job_contact);
}
/*
* Deactivating the module allows it to free memory and close network
* connections.
*/
rc = globus_module_deactivate(GLOBUSGRAM_CLIENT_MODULE);
out:
return rc;
/* End of gram_submit_example.c */
4. Submitting a Job and Processing Job State Callbacks

This example shows how to submit a job to a GRAM service and then wait until the job reaches the FAILED or DONE state. The source to this example\(^4\) can be downloaded.

```c
#include "globus_common.h"
#include "globus_gram_client.h"
#include <stdio.h>

struct monitor_t
{
    globus_mutex_t mutex;
    globus_cond_t cond;
    globus_gram_protocol_job_state_t state;
};

// * Job State Callback Function *
* This function is called when the job manager sends job states.
*/
static void example_callback(void * callback_arg, char * job_contact, int state,
                              int errorcode)
{
    struct monitor_t * monitor = callback_arg;

    globus_mutex_lock(&monitor->mutex);

    printf("Old Job State: %d\nNew Job State: %d\n", monitor->state, state);

    monitor->state = state;

    if (state == GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED ||
        state == GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE)
    {
        globus_cond_signal(&monitor->cond);
    }
    globus_mutex_unlock(&monitor->mutex);
}

int main(int argc, char *argv[])

4 gram_submit_and_wait_example.c
Basic GRAM Client Scenarios

{  
    int rc;
    char * callback_contact = NULL;
    char * job_contact = NULL;
    struct monitor_t monitor;

    if (argc != 3)  
    {  
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT RSL\n", argv[0]);
        rc = 1;

        goto out;
    }

    /*  
    * Always activate the GLOBUS_GRAM_CLIENT_MODULE prior to using any  
    * functions from the GRAM Client API or behavior is undefined.  
    */
    rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error activating %s because %s (Error %d)\n",
                GLOBUS_GRAM_CLIENT_MODULE->module_name,
                globus_gram_client_error_string(rc),
                rc);
        goto out;
    }

    rc = globus_mutex_init(&monitor.mutex, NULL);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error initializing mutex\n");
        goto deactivate;
    }

    rc = globus_cond_init(&monitor.cond, NULL);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error initializing condition variable\n");
        goto destroy_mutex;
    }

    monitor.state = GLOBUS_GRAM_PROTOCOL_JOB_STATE_UNSUBMITTED;

    globus_mutex_lock(&monitor.mutex);

    /*  
    * Allow GRAM state change callbacks  
    */
    rc = globus_gram_client_callback_allow(  
        example_callback, &monitor, &callback_contact);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error allowing callbacks because %s (Error %d)\n",
                globus_gram_client_error_string(rc), rc);
        goto out;
    }
}
goto destroy_cond;
}
/*
 * Submit the job request to the service passed as our first command-line
 * option.
 */
rc = globus_gram_client_job_request(
    argv[1], argv[2],
    GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED|
    GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE,
    callback_contact, &job_contact);

if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Unable to submit job to %s because %s (Error %d)\n",
            argv[1], globus_gram_client_error_string(rc), rc);
    /* Job submit failed. Short circuit the while loop below by setting
     * the job state to failed
     */
    monitor.state = GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED;
}
else
{
    /* Display job contact string */
    printf("Job submit successful: %s\n", job_contact);
}

/* Wait for job state callback to let us know the job has completed */
while (monitor.state != GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE &&
    monitor.state != GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED)
{
    globus_cond_wait(&monitor.cond, &monitor.mutex);
}
rc = globus_gram_client_callbackDisallow(callback_contact);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Error disabling callbacks because %s (Error %d)\n",
            globus_gram_client_error_string(rc), rc);
    globus_mutex_unlock(&monitor.mutex);
}
if (job_contact != NULL)
{
    free(job_contact);
}

destroy_cond:
    globus_cond_destroy(&monitor.cond);
destroy_mutex:
    globus_mutex_destroy(&monitor.mutex);
deactivate:
/*
 * Deactivating the module allows it to free memory and close network
 * connections.
5. Polling Job Status

This example shows how to submit a job to a GRAM service and then wait until the job reaches the FAILED or DONE state. The source to this example\(^5\) can be downloaded.

```c
#include "globus_common.h"
#include "globus_gram_client.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rc;
    int status = 0;
    int failure_code = 0;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s JOB-CONTACT\n", argv[0]);
        rc = 1;
        goto out;
    }

    /*
    * Always activate the GLOBUS_GRAM_CLIENT_MODULE prior to using any
    * functions from the GRAM Client API or behavior is undefined.
    */
    rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
    if (rc != GLOBUS_SUCCESS) {
        fprintf(stderr, "Error activating %s because %s (Error %d)\n", 
                GLOBUS_GRAM_CLIENT_MODULE->module_name, 
                globus_gram_client_error_string(rc), rc);
        goto out;
    }

    /*
    * Check the job status of the job named by the first argument to
    * this program.
    */

5 gram_poll_example.c
```
Basic GRAM Client Scenarios

*/
rc = globus_gram_client_job_status(argv[1], &status, &failure_code);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Unable to check job status because %s (Error %d)\n",
            globus_gram_client_error_string(rc), rc);
} else
{
    switch (status)
    {
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_UNSUBMITTED:
        printf("Unsubmitted\n");
        break;
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_STAGE_IN:
        printf("StageIn\n");
        break;
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_PENDING:
        printf("Pending\n");
        break;
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_ACTIVE:
        printf("Active\n");
        break;
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_SUSPENDED:
        printf("Suspended\n");
        break;
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_STAGE_OUT:
        printf("StageOut\n");
        break;
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE:
        printf("Done\n");
        break;
    case GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED:
        printf("Failed (%d)\n", failure_code);
        break;
    default:
        printf("Unknown job state\n");
        break;
    }
}

/* Deactivating the module allows it to free memory and close network */
/* connections. */
rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
out:
    return rc;
} /* End of gram_poll_example.c */
6. Canceling a Job

This example shows how to cancel a job being run by a GRAM service. The source to this example\(^6\) can be downloaded.

```c
#include "globus_common.h"
#include "globus_gram_client.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rc;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s JOB-CONTACT\n", argv[0]);
        rc = 1;
        goto out;
    }

    rc = globus_gram_client_job_cancel(argv[1]);
    if (rc != GLOBUS_SUCCESS) {
        fprintf(stderr, "Unable to cancel job because %s (Error %d)\n", 
            globus_gram_client_error_string(rc), rc);
    }
}
```

\(^6\) gram_cancel_example.c
Basic GRAM Client Scenarios

* Deactivating the module allows it to free memory and close network
  * connections.
  */
  rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
out:
  return rc;
} /* End of gram_cancel_example.c */

7. Refreshing Job Credential

This example shows how to refresh a GRAM job's credential after the job has been submitted by some other means. The source to this example⁷ can be downloaded.

/*
 * These headers contain declarations for the globus_module functions
 * and GRAM Client API functions
 */
#include "globus_common.h"
#include "globus_gram_client.h"

#include <stdio.h>

int
main(int argc, char *argv[])
{
  int rc;

  if (argc != 2)
  {
    fprintf(stderr, "Usage: %s JOB-CONTACT\n", argv[0]);
    rc = 1;
    goto out;
  }

  printf("Refreshing Credential for GRAM Job: %s\n", argv[1]);

  /*
   * Always activate the GLOBUS_GRAM_CLIENT_MODULE prior to using any
   * functions from the GRAM Client API or behavior is undefined.
   */
  rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
  if (rc != GLOBUS_SUCCESS)
  {
    fprintf(stderr, "Error activating %s because %s (Error %d)\n",
            GLOBUS_GRAM_CLIENT_MODULE->module_name,
            globus_gram_client_error_string(rc),
            rc);
    goto out;
  }

  /*

  ⁷ gram_refresh_example.c

  */
* Refresh the credential of the job running at the contact named
* by the first command-line argument to this program. We'll use the
* process's default credential by passing in GSS_C_NO_CREDENTIAL.
 */
rc = globus_gram_client_job_refresh_credentials(
    argv[1], GSS_C_NO_CREDENTIAL);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Unable to refresh credential for job %s because %s (Error %d)\n",
            argv[1], globus_gram_client_error_string(rc), rc);
}
else
{
    printf("Refresh successful\n")
    
}/* Deactivating the module allows it to free memory and close network
 * connections. */
rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
out:
    return rc;
}/* End of gram_refresh_example.c */
Chapter 4. Advanced GRAM Client Scenarios

1. Non-blocking Job Submission

This example shows how to submit a series of GRAM jobs using the non-blocking function
globus_gram_client_register_job_request and wait until all submissions have completed. This example throttles the number of concurrent job submissions to reduce the load on the service node. The source to this example\(^1\) can be downloaded.

```c
/*
 * These headers contain declarations for the globus_module functions
 * and GRAM Client API functions
 */
#include "globus_common.h"
#include "globus_gram_client.h"
#include <stdio.h>

struct monitor_t
{
    globus_mutex_t mutex;
    globus_cond_t cond;
    int submit_pending;
    int successful_submits;
};

#define CONCURRENT_SUBMITS 5

static void example_submit_callback(
    void * user_callback_arg,
    globus_gram_protocol_error_t operation_failure_code,
    const char * job_contact,
    globus_gram_protocol_job_state_t job_state,
    globus_gram_protocol_error_t job_failure_code)
{
    struct monitor_t * monitor = user_callback_arg;

    globus_mutex_lock(&monitor->mutex);
    monitor->submit_pending--;
    if (monitor->submit_pending < CONCURRENT_SUBMITS)
    {
        globus_cond_signal(&monitor->cond);
    }
    printf("Submitted job %s\n",
           job_contact ? job_contact : "UNKNOWN")
    if (operation_failure_code == GLOBUS_SUCCESS)
```

\(^1\) gram_nonblocking_submit_example.c
```c
{     
monitor->successful_submits++;
}  
else{  
printf("submit failed because %s (Error %d)\n",
    
globus_gram_client_error_string(operation_failure_code),
    
operation_failure_code);
}  
globus_mutex_unlock(&monitor->mutex);
}  

int  
main(int argc, char *argv[])
{
    
int rc;

int i;

struct monitor_t monitor;

if (argc < 3)
{
    fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT RSL-SPEC...
", 
    
argv[0]);
    
rc = 1;

    goto out;
}

printf("Submiting %d jobs to %s\n", argc-2, argv[1]);

/*
 * Always activate the GLOBUS_GRAM_CLIENT_MODULE prior to using any
 * functions from the GRAM Client API or behavior is undefined.
 */

rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Error activating %s because %s (Error %d)\n",
    
GLOBUS_GRAM_CLIENT_MODULE->module_name, 
    
globus_gram_client_error_string(rc), 
    
rc);

    goto out;
}

rc = globus_mutex_init(&monitor.mutex, NULL);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Error initializing mutex %d\n", rc);

    goto deactivate;
}

rc = globus_cond_init(&monitor.cond, NULL);
if (rc != GLOBUS_SUCCESS)
```
{    fprintf(stderr, "Error initializing condition variable \%d\n", rc);    goto destroy_mutex;  }

monitor.submit_pending = 0;

/* Submits jobs from argv[2] until end of the argv array. At most
 * CONCURRENT_SUBMITS will be pending at any given time.
 */
globus_mutex_lock(&monitor.mutex);
for (i = 2; i < argc; i++){
    /* This throttles the number of concurrent job submissions */
    while (monitor.submit_pending >= CONCURRENT_SUBMITS) {
        globus_cond_wait(&monitor.cond, &monitor.mutex);
    }
    /* When the job has been submitted, the example_submit_callback
     * will be called, either from another thread or from a
     * globus_cond_wait in a nonthreaded build
     */
    rc = globus_gram_client_register_job_request(
        argv[1], argv[i], 0, NULL, NULL, example_submit_callback,
        &monitor);
    if (rc != GLOBUS_SUCCESS) {
        fprintf(stderr, "Unable to submit job %s because %s (Error %d)\n", argv[i], globus_gram_client_error_string(rc), rc);
    } else {
        monitor.submit_pending++;
    }
}
/* Wait until the example_submit_callback function has been called for
 * each job submission
 */
while (monitor.submit_pending > 0) {
    globus_cond_wait(&monitor.cond, &monitor.mutex);
} 
globus_mutex_unlock(&monitor.mutex);

printf("Submitted %d jobs (%d successfully)\n", argc-2, monitor.successful_submits);

    globus_cond_destroy(&monitor.cond);
    globus_mutex_destroy(&monitor.mutex);
    deactivate:
/*
Deactivating the module allows it to free memory and close network connections.

```c
rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
```

```c
/* End of gram_nonblocking_submit_example.c */
```

## 2. Custom Security Attributes

This example shows how to submit a job and delegate a full credential to the job. The source to this example\(^2\) can be downloaded.

```c
/*
 * These headers contain declarations for the globus_module functions
 * and GRAM Client API functions
 */
#include "globus_common.h"
#include "globus_gram_client.h"

#include <stdio.h>

struct monitor_t
{
    globus_mutex_t mutex;
    globus_cond_t cond;
    globus_bool_t done;
};

static
void
example_submit_callback(
    void * user_callback_arg,
    globus_gram_protocol_error_t operation_failure_code,
    const char * job_contact,
    globus_gram_protocol_job_state_t job_state,
    globus_gram_protocol_error_t job_failure_code)
{
    struct monitor_t * monitor = user_callback_arg;

    globus_mutex_lock(&monitor->mutex);
    monitor->done = GLOBUS_TRUE;
    globus_cond_signal(&monitor->cond);
    if (operation_failure_code == GLOBUS_SUCCESS)
    {
        printf("Submitted job %s\n",
                job_contact ? job_contact : "UNKNOWN");
    }
    else
    {
        printf("submit failed because %s (Error %d)\n",
                job_contact ? job_contact : "UNKNOWN",
                job_failure_code);
    }
}
```

\(^2\)gram_attr_example.c
globus_gram_client_error_string(operation_failure_code),
operation_failure_code);
}
globus_mutex_unlock(&monitor->mutex);

int
main(int argc, char *argv[])
{
    int rc;
globus_gram_client_attr_t attr;
    struct monitor_t monitor;

    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT RSL-SPEC...
", argv[0]);
        rc = 1;
        goto out;
    }

    printf("Submiting job to %s with full proxy\n", argv[1]);

    /*
     * Always activate the GLOBUS_GRAM_CLIENT_MODULE prior to using any
     * functions from the GRAM Client API or behavior is undefined.
     */
    rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error activating %s because %s (Error %d)\n","%
GLOBUS_GRAM_CLIENT_MODULE->module_name, globus_gram_client_error_string(rc),
rc);
        goto out;
    }

    rc = globus_mutex_init(&monitor.mutex, NULL);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error initializing mutex %d\n", rc);
        goto deactivate;
    }

    rc = globus_cond_init(&monitor.cond, NULL);
    if (rc != GLOBUS_SUCCESS)
    {
        fprintf(stderr, "Error initializing condition variable %d\n", rc);
        goto destroy_mutex;
    }

    monitor.done = GLOBUS_FALSE;
/* Initialize attribute so that we can set the delegation attribute */
rc = globus_gram_client_attr_init(&attr);

/* Set the proxy attribute */
rc = globus_gram_client_attr_set_delegation_mode(
    attr,
    GLOBUS_IO_SECURE_DELEGATION_MODE_FULL_PROXY);

/* Submit the job rsl from argv[2] */
globus_mutex_lock(&monitor.mutex);
/* When the job has been submitted, the example_submit_callback * 
will be called, either from another thread or from a 
* globus_cond_wait in a nonthreaded build */
rc = globus_gram_client_register_job_request(
    argv[1], argv[2], 0, NULL, attr, example_submit_callback, 
    &monitor);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Unable to submit job %s because %s (Error %d)\n",
        argv[2], globus_gram_client_error_string(rc), rc);
}

/* Wait until the example_submit_callback function has been called for 
* the job submission */
while (!monitor.done)
{
    globus_cond_wait(&monitor.cond, &monitor.mutex);
}
globus_mutex_unlock(&monitor.mutex);

globus_cond_destroy(&monitor.cond);
destroy_mutex:
    globus_mutex_destroy(&monitor.mutex);
deactivate:
    /*
    * Deactivating the module allows it to free memory and close network 
    * connections.
    */
    rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
out:
    return rc;

/* End of gram_attr_example.c */
3. Modifying RSL

This example shows how to programmatically add environment variable definitions to an RSL prior to submitting a job. The source to this example can be downloaded.

```c
/*
 * These headers contain declarations for the globus_module,
 * the GRAM Client, RSL, and protocol functions
 */
#include "globus_common.h"
#include "globus_gram_client.h"
#include "globus_rsl.h"
#include "globus_gram_protocol.h"

#include <stdio.h>
#include <strings.h>

static int example_rsl_attribute_match(void * datum, void * arg)
{
    const char * relation_attribute = globus_rsl_relation_get_attribute(datum);
    const char * attribute = arg;

    /* RSL attributes are case-insensitive */
    return (relation_attribute &&
            strcasecmp(relation_attribute, attribute) == 0);
}

int main(int argc, char *argv[])
{
    int rc;
    globus_rsl_t *rsl, *environment_relation;
    globus_rsl_value_t *new_env_pair = NULL;
    globus_list_t *environment_relation_node;
    char * rsl_string;
    char * job_contact;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s RESOURCE-MANAGER-CONTACT RSL\n", argv[0]);
        rc = 1;

        goto out;
    }

    /*
     * Always activate the GLOBUSGRAM_CLIENT_MODULE prior to using any
     * functions from the GRAM Client API or behavior is undefined.
     */
```

3. gram_rsl_example.c
rc = globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Error activating %s because %s (Error %d)\n", 
            GLOBUS_GRAM_CLIENT_MODULE->module_name, 
            globus_gram_client_error_string(rc), 
            rc);
    goto out;
}

/* Parse the RSL string into a syntax tree */
rsl = globus_rsl_parse(argv[2]);
if (rsl == NULL)
{
    rc = 1;
    fprintf(stderr, "Error parsing RSL string\n");
    goto deactivate;
}

/* Create the new environment variable pair that we'll insert 
* into the RSL. We'll start by making an empty sequence 
*/
new_env_pair = globus_rsl_value_make_sequence(NULL);
if (new_env_pair == NULL)
{
    fprintf(stderr, "Error creating value sequence\n");
    rc = 1;
    goto free_rsl;
}

/* Then insert the name-value pair in reverse order */
rc = globus_list_insert( 
    globus_rsl_value_sequence_get_list_ref(new_env_pair),
    globus_rsl_value_make_literal( 
        strdup("itsvalue")));
if (rc != GLOBUS_SUCCESS)
{
    goto free_env_pair;
}

rc = globus_list_insert( 
    globus_rsl_value_sequence_get_list_ref(new_env_pair),
    globus_rsl_value_make_literal( 
        strdup("EXAMPLE_ENVIRONMENT_VARIABLE")));
if (rc != GLOBUS_SUCCESS)
{
    goto free_env_pair;
}

/* Now, check to see if the RSL already contains an environment 
* attribute. */
environment_relation_node = globus_list_search_pred( 
    globus_rsl_boolean_get_operand_list(rsl),
    example_rsl_attribute_match,
if (environment_relation_node == NULL)
{
    /* Not present yet, create a new relation and insert it into
       the RSL. */
    environment_relation = globus_rsl_make_relation(
        GLOBUS_RSL_EQ,
        strdup(GLOBUS_GRAM_PROTOCOL_ENVIRONMENT_PARAM),
        globus_rsl_value_make_sequence(NULL));
    rc = globus_list_insert(
        globus_rsl_boolean_get_operand_list_ref(rsl),
        environment_relation);
    if (rc != GLOBUS_SUCCESS)
    {
        globus_rsl_free_recursive(environment_relation);
        goto free_env_pair;
    }
}
else
{
    /* Pull the environment relation out of the node returned from the
       search function */
    environment_relation = globus_list_first(environment_relation_node);
}

/* Add the new environment binding to the value sequence associated with
   the environment relation */
rc = globus_list_insert(
    globus_rsl_value_sequence_get_list_ref(
        globus_rsl_relation_get_value_sequence(environment_relation)),
    new_env_pair);
if (rc != GLOBUS_SUCCESS)
{
    goto free_env_pair;
}
new_env_pair = NULL;

/* Convert the RSL parse tree to a string */
rsl_string = globus_rsl_unparse(rsl);

/* Submit the augmented RSL to the service passed as our first command-line
   option. If successful, this function will return GLOBUS_SUCCESS,
   otherwise an integer error code. */
rc = globus_gram_client_job_request(
    argv[1],
    rsl_string,
    0,
    NULL,
&job_contact);
if (rc != GLOBUS_SUCCESS)
{
    fprintf(stderr, "Unable to submit job to %s because %s (Error %d)\n", 
            argv[1], globus_gram_client_error_string(rc), rc);
}
else
{
    printf("Job submitted successfully: %s\n", job_contact);
}

free(rsl_string);

if (job_contact)
{
    free(job_contact);
}

free_env_pair:
    if (new_env_pair != NULL)
    {
        globus_rsl_value_free_recursive(new_env_pair);
    }

free_rsl:
    globus_rsl_free_recursive(rsl);

deactivate:
    /*
     * Deactivating the module allows it to free memory and close network
     * connections.
     */
    rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
out:
    return rc;
}

/* End of gram_rsl_example.c */
Part III. GRAM Server Developer's Guide
Table of Contents

5. LRM Adapter Tutorial ........................................................................................................ 36
   1. Introduction ....................................................................................................................... 36
   2. Parts of a GRAM5 LRM Adapter ....................................................................................... 36
   3. RSL Validation File ............................................................................................................. 36
   4. Configuration File ............................................................................................................. 40
   5. LRM Adapter Perl Module ............................................................................................... 40
   6. LRM SEG Module ............................................................................................................ 46
   7. Changes from Previous Versions ....................................................................................... 59
Chapter 5. LRM Adapter Tutorial

1. Introduction

GRAM5 provides a resource-independent abstraction to remote job management. The resource abstraction contains methods for job submission and cancelling, and a method for monitoring job state changes. This set of tutorials describes how to implement and bundle all packages needed for a complete LRM Adapter interface for GRAM5.

For purposes of this tutorial, we will create a fake LRM adapter that pretends to run jobs, but in fact just keeps track of jobs and expires them after the job's max_wall_time expires. We'll call this LRM the fake LRM adapter.

2. Parts of a GRAM5 LRM Adapter

A GRAM5 LRM Adapter consists of a few parts which work together to provide a full interface between the GRAM5 Job Manager and the Local Resource Manager. These parts include:

- **RSL Validation File**
  - An option file which defines any custom RSL attributes which the LRM Adapter implements, or sets any custom defaults for RSL attributes that the LRM processes. Defining new RSL attributes in this file allows the GRAM5 service to detect some sets of RSL errors without invoking the Perl LRM Adapter Module. For this example, the file will be called fake.rvf.

- **Perl LRM Adapter Module**
  - A Perl module which implements the execution interface to the LRM. This module translates the Resource Specification Language description of a job's requirements to a concrete way of starting the job on a particular LRM. For this example, this file will be called fake.pm.

- **Configuration File**
  - The GRAM5 service implements a simple configuration file parser which can be used to provide a way to add site customizations to LRM Adapters. These files are usually shared between the Perl LRM Adapter Module and the Scheduler Event Generator Module. For this example, this file will be called fake.conf.

- **Gatekeeper Service File**
  - The Gatekeeper is a privileged service which authenticates and authorizes clients and then starts a Job Manager process on their behalf. The Gatekeeper Service File contains the LRM-specific command-line options to the job manager. For this example, this file will be called jobmanager-fork.

- **Scheduler Event Generator Module**
  - A dynamic object which parses LRM state and generates job state change events in a generic format for GRAM5 to consume. For this example, the SEG module will be called libglobus_seg_fake.so.

3. RSL Validation File

Each LRM Adapter can have a custom RSL validation file (RVF) which indicates which RSL attributes are valid for that LRM, what their default values are, and when they can be used during a job lifecycle.

The RVF entries consist of a set of records containing attribute-value pairs, with a blank line separating records. Each attribute-value pair is separated by the colon character. The value may be quoted with the double-quote character, in which case, the value continues until a second quote character is found; otherwise, the value terminates at end of line.
3.1. RVF Attributes

The attribute names understood by the GRAM5 RVF parser are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>The name of an RSL attribute.</td>
</tr>
<tr>
<td>Description</td>
<td>A textual description of the attribute.</td>
</tr>
<tr>
<td>RequiredWhen</td>
<td>A sequence of WHEN-VALUES describing when this RSL attribute must be present.</td>
</tr>
<tr>
<td>DefaultWhen</td>
<td>A sequence of WHEN-VALUES describing when the default RSL value will be applied if it's not present in the RSL.</td>
</tr>
<tr>
<td>ValidWhen</td>
<td>A sequence of WHEN-VALUES describing when the RSL attribute may be present.</td>
</tr>
<tr>
<td>Default</td>
<td>A literal RSL value sequence containing the default value of the attribute, applied to the RSL when the attribute is not present, but the RSL use matches the DefaultWhen value.</td>
</tr>
<tr>
<td>Values</td>
<td>A sequence of strings enumerating the legal values for the RSL attribute.</td>
</tr>
<tr>
<td>Publish</td>
<td>When set to true, the RSL attribute will be added to the documentation for the LRM Adapter if the RVF is processed by the create_rsl_documentation.pl script. Otherwise, it will not be mentioned.</td>
</tr>
</tbody>
</table>

3.1.1. RVF When Values

The WHEN-VALUES used by the RVF parser are described in this list:

- **GLOBUS_GRAM_JOB_SUBMIT**: RSL Attribute used in a GRAM5 job request to submit a job to an LRM Adapter.
- **GLOBUS_GRAM_JOB_RESTART**: RSL Attribute used in a GRAM5 job request to restart a job which was stopped due to a two-phase commit timeout.
- **GLOBUS_GRAM_JOB_STDIO_UPDATE**: RSL Attribute used in a GRAM5 STDIO_UPDATE signal, which may be sent to a job during the two-phase end state.

3.1.2. Common RSL Attributes

The GRAM5 service by default implements a common set of RSL attributes for all jobs. Not all of these may be relevant to all LRM types, but are included in the common set so that the same concept will be processed by the same attribute for each LRM. LRM Adapters can disable particular RSL attributes if they want by adding the attribute to their RVF file with

```
Attribute: AttributeName
ValidWhen: ""
```

The common list of attributes is described in Section 4, “RSL Attribute Summary”.

3.2. Creating a RSL Validation File for the Fake LRM

Normally, the RVF for a new LRM Adapter will add any LRM-specific RSL attributes and perhaps change the DefaultValue for some. For the fake LRM, we'll be a bit more complicated and disable most of the GRAM common...
RSL attributes and reduce things to indicate the queue and execution time for the fake jobs. The fake.rvf will do the following:

- **Remove** executable, arguments, directory, environment, file_clean_up, file_stage_in, file_stage_out, file_stage_in_shared, gass_cache, gram_my_job, host_count, library_path, max_cpu_time, min_memory, project, queue, remote_io_url, scratch_dir, stdin, stdout, and stderr attributes.

- **Add a** max_queue_time attribute, which will be the maximum time a particular fake job will be in the PENDING state. This will have a default of 20 minutes.

- **Add a default value to the** max_wall_time attribute of 5 minutes.
Attribute: library_path
  ValidWhen: ""
  RequiredWhen: ""

Example 5.1. fake.rvf
Attribute: max_cpu_time
  ValidWhen: ""
  RequiredWhen: ""

Attribute: min_memory
  ValidWhen: ""
  RequiredWhen: ""

Attribute: project
  ValidWhen: ""
  RequiredWhen: ""

Attribute: queue
  ValidWhen: ""
  RequiredWhen: ""

Attribute: remote_io_url
  ValidWhen: ""
  RequiredWhen: ""

Attribute: scratch_dir
  ValidWhen: ""
  RequiredWhen: ""

Attribute: stdin
  ValidWhen: ""
  RequiredWhen: ""

Attribute: stdout
  ValidWhen: ""
  RequiredWhen: ""

Attribute: stderr
  ValidWhen: ""
  RequiredWhen: ""

# Add a new attribute max_queue_time
Attribute: max_queue_time
  ValidWhen: GLOBUS_GRAM_JOB_SUBMIT
  DefaultWhen: GLOBUS_GRAM_JOB_SUBMIT
  RequiredWhen: GLOBUS_GRAM_JOB_SUBMIT
  Description: "Maximum time a fake job will be in pending, in seconds. The default value is 1200 seconds (20 minutes)"
  Default: 1200

# Add a default value and requirement for max_wall_time
Attribute: max_wall_time
  DefaultWhen: GLOBUS_GRAM_JOB_SUBMIT
  RequiredWhen: GLOBUS_GRAM_JOB_SUBMIT
  Default: 300
  Description: "Maximum time a fake job will be in the ACTIVE state"

1 fake.rvf
4. Configuration File

For the fake LRM, there's not much to configure: a path to a file where the LRM should write its job files. For real LRMs, there are other things which might belong there: paths to LRM-specific executables such as qsub, tuning parameters to the LRM adapter script such as the number of available cores per execution node, or the host to contact when using a remote submit protocol between GRAM and the LRM. The configuration parameters used by the LRM adapters included in GRAM5 are described in Section 4, “LRM Adapter Configuration”.

The LRM adapter configuration files consist of attribute="value" pairs, which comment lines beginning with the # character. For the example fake LRM, the configuration file looks like this:

```
# log_path is the path to log file that the fake LRM generates. This file is
# updated each time a job is submitted or cancelled. The default if it is not
# present is ${localstatedir}/fake, which is typically /var/fake
log_path="/tmp"
```

4.1. Parsing the Configuration File

The Globus Toolkit contains API functions for parsing files in the format used by the LRM configuration files. In Perl, use the Globus::Core::Config class. In C/C++, use the globus_common_get_attribute_from_config_file() function.

4.1.1. Perl API

The Globus::Core::Config API is quite simple. The new() constructor parses the configuration file and returns an object containing the attribute=value pairs. The get_attribute() method returns the value of the named attribute. These functions are used in the fake LRM Perl Module.

4.1.2. C/C++ API

The globus_common_get_attribute_from_config_file() function will load the configuration file and return the value of the attribute passed to it. This function is used in the SEG module below. Note that this function returns a pointer to a copy of the string value in the location pointed to by the value parameter. The caller must free this value.

5. LRM Adapter Perl Module

The Perl-language LRM module provides the job submission and cancelling interface between GRAM5 and the underlying scheduler. Very little has been added to this part of the scheduler interface since Globus Toolkit 2—if you have a version for an older Globus Toolkit release, you can ignore most of this tutorial and jump to the Section 7, “Changes from Previous Versions” section of this tutorial. The module annotated below is available from fake.pm.

5.1. Perl LRM Adapter Module

The LRM Adapter interface is implemented as a Perl module which is a subclass of the Globus::GRAM::JobManager module. Its name must match the type string used when the job manager is started, but in all lower case: for the fake LRM, the module name is Globus::GRAM::JobManager::fake and it is stored in the file fake.pm. Though there are several methods in the Globus::GRAM::JobManager interface, the only ones which absolutely need to be implemented in a scheduler module are submit and cancel. The poll can be used if there is no SEG module for your LRM Adapter, but using polling can be resource intensive and slow. We'll present the methods in the module one by one, but the entire module can be downloaded from here: fake.pm.
We'll begin by looking at the start of the fake.pm source module. To begin the script, we import the GRAM support modules into the LRM adapter module's namespace, declare the module's package, and declare this module as a subclass of the Globus::GRAM::JobManager module. All LRM adapter packages will need to do this, substituting the name of the LRM type being implemented where we see fake below.

```perl
use Globus::GRAM::Error;
use Globus::GRAM::JobState;
use Globus::GRAM::JobManager;
use Globus::Core::Paths;
use Globus::Core::Config;
use File::Path;
use strict;
use warnings;

package Globus::GRAM::JobManager::fake;

our @ISA = ("Globus::GRAM::JobManager");

Next, we declare any system-specific values which will be read from the configuration file. In the fake case, we will declare a module-global directory for job information and for SEG log entries. In real LRM Adapters, there are often variables which are loaded from the configuration file for such things as the list of available queues, paths to executables such as mpiexec, and any other site-specific configuration.

```perl
our($job_dir, $fake_seg_dir);

BEGIN
{
    my $config = new Globus::Core::Config(
        '${sysconfdir}/globus/globus-fake.conf');

    $job_dir = $fake_seg_dir = "";

    if ($config)
    {
        $job_dir = $config->get_attribute("log_path") || "";
    }
    if ($job_dir eq '')
    {
        $job_dir = Globus::Core::Paths::eval_path('${localstatedir}/fake');
    }
}

5.1.1. Writing a Constructor

For LRM Adapter 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 default Globus::GRAM::JobManager::new constructor will do the job.

If you do need to overloaded this method, be sure to call the parent module's constructor to allow it to do its initialization. In this example, we create an object which includes a sequence number to ensure that the job ids returned from the LRM script is unique.

```perl
sub new
{
    my $proto = shift;
```
my $class = ref($proto) || $proto;
my $self = $class->SUPER::new(0_);
$self->{sequence} = 0;

return $self;
}

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

- **job_id**: The string returned as the value of JOB_ID in the eturn 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 and might be useful in creating temporary files or LRM-specific processing.

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

### 5.1.2. Submitting Jobs

All LRM adapter modules must implement the submit method. This method is called when the job manager wishes to submit the job to the LRM. The information in the original job request RSL string is available to the LRM adapter interface through the JobDescription data member of its hash.

For most LRM Adapters, this is the longest method to be implemented, as it must decide what to do with the job description, and convert RSL elements to something which the LRM can understand.

For our fake adapter, we will validate that the two RSL attributes we process are integers, and if so generate a new unique LRM ID and return it and the state Globus::GRAM::JobState::PENDING. Note the call to respond with GT3_FAILURE_MESSAGE. This allows the GRAM5 client application to see the context-sensitive error message along with the general failure code from GRAM5.

sub submit {
    my $self = shift;
    my $description = $self->{JobDescription};
    my $now = time();
    my $jobid;
    my $fh;
    my $pending_time;
    my $active_time;
    my $done_time;
    my $failed_time = 0;

    if ($description->max_wall_time() != int($description->max_wall_time()))
    {
        return Globus::GRAM::Error::INVALID_MAX_WALL_TIME;
    }
    elsif ($description->max_queue_time() != int($description->max_queue_time()))
    {
        $self->respond({GT3_FAILURE_MESSAGE => "Invalid max_queue_time");
        return Globus::GRAM::Error::INVALID_ATTR;
    }
}

my $class = ref($proto) || $proto;
my $self = $class->SUPER::new(0_);
$self->{sequence} = 0;

return $self;
}
$self->{sequence}++;  
$pending_time = $now;  
$active_time = $pending_time + int($description->max_queue_time);  
$done_time = $active_time + int($description->max_wall_time);  

$jobid = sprintf("%.63s", "$\$\$\$.self->{sequence}.".$now");  

if (!open($fh, ">>$job_dir/fakejob.log"))  
{  
    $self->respond({GT3_FAILURE_MESSAGE => "Unable to write job file"});  
    return Globus::GRAM::Error::INVALID_SCRIPT_STATUS;  
}  

print $fh "$jobid;$pending_time;$active_time;$done_time;$failed_time\n";  
close($fh);  

return { JOB_STATE => Globus::GRAM::JobState::PENDING,  
          JOB_ID => $jobid };  
}

That finishes the submit method. Most of the functionality for the scheduler interface is now written.

### 5.1.3. Polling Job State

GRAM5 requires some way to determine the state of a job. In most systems, writing a Scheduler Event Generator module will provide the best performance and lowest resource overhead. However, when developing an LRM adapter, it is helpful to implement the polling interface so that the submission and cancel mechanism can be tested independent of having the SEG module completed. For the fake LRM Adapter, we'll write a simple poll method which will compare the current time with the time when the job was originally submitted.

sub poll  
{
    my $self = shift;  
    my $description = $self->{JobDescription};  
    my $state;  
    my $pid;  
    my $now;  
    my $fh;  
    my $pending_time = 0;  
    my $active_time;  
    my $done_time;  
    my $failed_time;  
    my $seqno;  
    my $jobid = $description->jobid();  

    if(!defined $jobid)  
    {  
        $self->log("poll: job id undefined!");  
        return { JOB_STATE => Globus::GRAM::JobState::FAILED };  
    }  

    open($fh, "<$job_dir/fakejob.log");  

    # Multiple matches might occur if the job is cancelled, so we keep looping
# until EOF
while (<$fh>)
{
    chomp;

    my @fields = split(/;/);

    if ($fields[0] ne $jobid)
    {
        next;
    }

    $pending_time = $fields[1];
    $active_time = $fields[2];
    $done_time = $fields[3];
    $failed_time = $fields[4];
}

close($fh);

$now = time();

if ($pending_time == 0)
{
    # not found
    $state = Globus::GRAM::JobState::FAILED;
}
elsif (int($failed_time) != 0)
{
    $state = Globus::GRAM::JobState::FAILED;
}
elsif ($now < int($active_time))
{
    $state = Globus::GRAM::JobState::PENDING;
    return
}
elsif ($now < int($done_time))
{
    $state = Globus::GRAM::JobState::ACTIVE;
}
else
{
    $state = Globus::GRAM::JobState::DONE;
}

return { JOB_STATE => $state };
FAILED state. Otherwise we can return an empty hash reference, and let either the Scheduler Event Generator or a subsequent call to poll determine when the state change occurs.

For the fake LRM adapter, we will update the job file with a cancellation time and return the Globus::GRAM::JobState::FAILED state change.

```perl
sub cancel {
    my $self = shift;
    my $description = $self->{JobDescription};
    my $pgid;
    my $jobid = $description->{jobid};
    my $fh;
    my $pending_time = 0;
    my $active_time;
    my $done_time;
    my $failed_time;
    my $now = time();

    if(!defined $jobid) {
        $self->log("cancel: no jobid defined!");
        return { JOB_STATE => Globus::GRAM::JobState::FAILED };}
    }

    open($fh, "<$job_dir/fakejob.log");

    # Multiple matches might occur if the job is cancelled, so we keep looping
    # until EOF
    while (<$fh>) {
        chomp;

        my @fields = split(/;/);

        if ($fields[0] ne $jobid) {
            next;
        }

        $pending_time = $fields[1];
        $active_time = $fields[2];
        $done_time = $fields[3];
        $failed_time = $fields[4];
    }
    close($fh);

    $self->log("cancel job ", $jobid);
    if ($now < int($done_time) && int($failed_time) == 0) {
        $failed_time = $now;
        $done_time = 0;
        if (!open($fh, ">>$job_dir/fakejob.log")) {
            ...
```
$self->respond({GT3_FAILURE_MESSAGE => "Unable to write job file"});
return Globus::GRAM::Error::INVALID_SCRIPT_STATUS;
}
print $fh "$jobid;$pending_time;$active_time;$done_time;$failed_time\n";
close($fh);
}

  return { JOB_STATE => Globus::GRAM::JobState::FAILED };
}

5.1.5. 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;

6. LRM SEG Module

6.1. Intro

The Scheduler Event Generator (SEG) module provides an efficient job monitoring interface between GRAM5 and
the underlying local resource manager. In most cases, the SEG module parses a log generated by the local resource
manager which contains information about job state changes and then uses the SEG API\(^1\) to signal job state changes
as they occur.

A SEG module is implemented as a shared library which is loaded as a globus extension. This means that the only
entry point to the library is a globus_module_descriptor, which defines activation and deactivation functions for the
library. For this tutorial, we will build up the SEG module piecewise, but the entire fake SEG module source\(^3\) can be
downloaded as well.

6.2. Outline

The outline for our SEG module is:

Include Headers
Module Specific Data
Module Specific Prototypes
Extension Module Descriptor
Module Activation
Module Deactivation
Process Events
Utility Functions

From this outline, we'll explain the various sections of the source file below.

6.3. LRM Module Dependencies

The LRM module uses the globus_common API from Globus for its linked list, mutual exclusion, timed events, and
module dependency tracking. It also uses the Scheduler Event Generator APIs, which provide functions for defining
and emitting LRM events.

\(^1\) http://www.globus.org/api/c-globus-5.2.2/globus_scheduler_event_generator/html/group__seg__api.html

\(^3\) seg_fake_module.c
Include Headers. For our implementation, we'll need to include the headers for the Globus modules we'll be using. In this case we'll be using `globus_common.h`, `globus_scheduler_event_generator.h` (which includes the API for emitting SEG events), and `globus_scheduler_event_generator_app.h` (which includes the SEG event type definitions).

```c
#include "globus_common.h"
#include "globus_scheduler_event_generator.h"
#include "globus_scheduler_event_generator_app.h"
```

6.4. Module Specific Data

For the fake LRM, we need to keep some global state to keep track of what we've parsed from our LRM's log file, and what events are we should be sending in the future. To do this, we define two data structures, a `fake_job_info_t` which defines the set of event timestamps associated with a job, and `fake_state_t` which contains the state of the fake SEG parser.

Fake Job Info

LRM Parser State

Fake Job Info. For the `fake_job_info_t` structure, we will want to keep track of the LRM id (an up to 64-character long string), and the timestamps for the pending, active, failed, and done events for the job. We use the timestamp value of 0 to indicate an event which will not happen or has already been processed.

```c
typedef struct
{
    char   jobid[64];
    time_t pending;
    time_t active;
    time_t failed;
    time_t done;
} fake_job_info_t;
```

In addition, we will keep a null initializer for the `fake_job_info_t` structure so that we can simply initialize dynamically allocated data.

```c
/* A statically-initialized empty job info which is used to initialize
 * dynamically allocated fake_job_info_t structs
 */
static fake_job_info_t fake_job_info_initializer;
```

LRM Parser State. For the LRM parser state, we will keep track of the start time for which we will emit events, the path to the fake job log, a file pointer open to that log, and a list of `fake_job_info_t` structs for each job we have data for. We also use a mutex/condition variable combination to block deactivation until all callback functions have completed. The data in this struct is initialized in the module's activation function below.

```c
/**
 * State of the FAKE log file parser.
 */
static struct
{
    /** Timestamp of when to start generating events from */
    time_t start_timestamp;
```
/** Log file path */
char *                              log_path;
/** Log file pointer */
FILE *                              log;
/** List of job info containing future info we might need to
* turn into job state changes */
globus_list_t *                     jobs;
/**
* shutdown mutex
*/
globus_mutex_t                      mutex;
/**
* shutdown condition
*/
globus_cond_t                       cond;
/**
* shutdown flag
*/
globus_bool_t                       shutdown_called;
/**
* callback count
*/
int                                 callback_count;
} globus_l_fake_state;

6.5. Module Specific Prototypes

For our SEG, we define a small number of static functions to process the fake job log. These include our activation
and deactivation functions, and our event callback which is called periodically to process the fake job log. We also
have a couple of utility functions to look up entries in the job list and a predicate used to sort a list of SEG events by
timestamp and jobid.

static
int
globus_l_fake_module_activate(void);

static
int
globus_l_fake_module_deactivate(void);

static
void
globus_l_fake_read_callback(void *user_arg);

static
int
globus_l_fake_find_by_job_id(void * datum, void * arg);

static
int
globus_l_fake_compare_events(void * low_datum, void * high_datum, void * relation_args);
6.6. Extension Module Descriptor

The SEG dynamically loads our code using the Globus Extension API. To implement the interface it needs, we must define an extension descriptor so that it can find the entry point to our library. This module descriptor contains pointers to the activation and deactivation functions we prototyped above. It can contain other pointers but they aren't needed for our module implementation so we leave them as NULL.

```c
GlobusExtensionDefineModule(globus_seg_fake) =
{
   "globus_seg_fake",
   globus_l_fake_module_activate,
   globus_l_fake_module_deactivate,
   NULL,
   NULL,
   NULL
};
```

6.7. Module Activation

The entry point to our LRM-specific module is the activation function. This function is invoked by the `globus-scheduler-event-generator` program when it starts and dynamically loads the LRM-specific module. It is not passed any parameters, and is expected to return `GLOBUS_SUCCESS` if it is able to activate itself. Typically the activation function will do the following:

```c
static
int
globus_l_fake_module_activate(void)
{
Declare Variables
Activate Dependencies
Prepare Shutdown Handler
Read Configuration
Register Event
Cleanup on Failure
   return result;
} /* globus_l_fake_module_activate() */
```

Declare Variables. For our activation function, we'll use variables to store the path to the configuration file as well as return values from functions we call.

```c
char *                              config_path = NULL;
char *                              log_dir;
int                                 rc;
globus_result_t                     result = GLOBUS_SUCCESS;
```

Activate Dependencies. The headers we've just included contain the module descriptors which we will activate in our LRM-specific activation function, so that we are able to use the APIs in those modules. Our module is only ever activated by the SEG module, so we shouldn't activate it. In the activation function for our module, we'll include this fragment

```c
rc = globus_module_activate(GLOBUS_COMMON_MODULE);
if (rc != GLOBUS_SUCCESS)
```
Prepare Shutdown Handler. To handling deactivation safely, we'll create a mutex and condition variable to handle the case of when a shutdown is called while our event handler is running. In that case, we'll set the shutdown_called variable to GLOBUS_TRUE and then wait until the callback has terminated. Here we just set the variables to their non-shutdown values.

```c
rc = globus_mutex_init(&globus_l_fake_state.mutex, NULL);
if (rc != GLOBUS_SUCCESS)
{
    result = GLOBUS_FAILURE;
    goto mutex_init_failed;
}

rc = globus_cond_init(&globus_l_fake_state.cond, NULL);
if (rc != GLOBUS_SUCCESS)
{
    result = GLOBUS_FAILURE;
    goto cond_init_failed;
}

globus_l_fake_state.shutdown_called = GLOBUS_FALSE;
globus_l_fake_state.callback_count = 0;
```

6.7.1. LRM SEG Module Configuration

Read Configuration. There are two main pieces of configuration information we'll need to process SEG events: the earliest timestamp we care about (which we get from the SEG module) and the path to our fake job log file (which we get from our configuration file as in the perl module).

So first, to get the timestamp, we'll use the `globus_scheduler_event_generator_get_timestamp()` function.

```c
result = globus_scheduler_event_generator_get_timestamp(
    &globus_l_fake_state.start_timestamp);
if (result != GLOBUS_SUCCESS)
{
    goto get_timestamp_failed;
}
```

Then, to get the configuration file data, we first construct the path to the configuration file and then pull out the configuration attribute log_path, falling back to the default (`${localstatedir}/fake`) if it is not found.

```c
result = globus_eval_path(
    "${sysconfdir}/globus/globus-fake.conf",
    &config_path);
if (result != GLOBUS_SUCCESS || config_path == NULL)
{
    goto get_config_path_failed;
}
```
result = globus_common_get_attribute_from_config_file(
    "",
    config_path,
    "log_path",
    &log_dir);

/* This default must match fake.pm's default for things to work */
if (result != GLOBUS_SUCCESS)
{
    result = globus_eval_path("${localstatedir}/fake", &log_dir);
}

if (result != GLOBUS_SUCCESS)
{
    goto get_log_dir_failed;
}

globus_l_fake_state.log_path =
    globus_common_create_string("%s/fakejob.log", log_dir);
if (globus_l_fake_state.log_path == NULL)
{
    result = GLOBUS_FAILURE;
    goto get_log_path_failed;
}

6.7.2. Register Event

The next main action the activation function does is to register an event to happen later to process the events in the
LRM log. For this, we use the globus_callback_register_oneshot() function to register an event han-
dler to execute as soon as possible within the globus-scheduduler-event-generator program. The callback function
in this case is the globus_l_fake_read_callback() function defined later.

result = globus_callback_register_oneshot(
    NULL,
    NULL,
    globus_l_fake_read_callback,
    &globus_l_fake_state);
if (result != GLOBUS_SUCCESS)
{
    goto register_oneshot_failed;
}
globus_l_fake_state.callback_count++;

6.7.3. Cleanup on Failure

Here we handle the errors that might have occurred above and free temporarily used memory. In case of a failure,
result is set to something other than GLOBUS_SUCCESS.

register_oneshot_failed:
get_log_path_failed:
    if (result != GLOBUS_SUCCESS)
    {
        free(globus_l_fake_state.log_path);
    }
free(log_dir);
get_log_dir_failed:
    free(config_path);
get_config_path_failed:
get_timestamp_failed:
    if (result != GLOBUS_SUCCESS)
    {
        malloc_state_failed:
            globus_cond_destroy(&globus_l_fake_state.cond);
cond_init_failed:
            globus_mutex_destroy(&globus_l_fake_state.mutex);
mutex_init_failed:
                globus_module_deactivate(GLOBUS_COMMON_MODULE);
    }
activation_failure:

6.8. Module Deactivation

For our deactivation function, we will wait use the shutdown handling variables in the state structure to wait until all
outstanding callback have terminated and then free memory associated with the state.

static
int
globus_l_fake_module_deactivate(void)
{
    // Shutdown Handling
    Cleanup State
} /* globus_l_fake_module_deactivate() */

Shutdown Handling. To handle shutdown safely, we must wait until pending callbacks have terminated. To do
this, we set the shutdown_called field in the state structure and wait until the callback_count field is 0.
Inside the callback function, if we see that the shutdown_called field is GLOBUS_TRUE then it will not rereg-
ister itself and will signal when it terminates.

globus_mutex_lock(&globus_l_fake_state.mutex);
globus_l_fake_state.shutdown_called = GLOBUS_TRUE;
while (globus_l_fake_state.callback_count > 0)
    {
        globus_cond_wait(&globus_l_fake_state.cond, &globus_l_fake_state.mutex);
    }
globus_mutex_unlock(&globus_l_fake_state.mutex);

Cleanup State. Finally, we’ll free data we allocated in the activation function.

globus_mutex_destroy(&globus_l_fake_state.mutex);
globus_cond_destroy(&globus_l_fake_state.cond);
if (globus_l_fake_state.log_path)
    {
        free(globus_l_fake_state.log_path);
    }
if (globus_l_fake_state.log)
    {
        fclose(globus_l_fake_state.log);
    }
while (!globus_list_empty(globus_l_fake_state.jobs))
{
    fake_job_info_t *info;

    info = globus_list_remove(
        &globus_l_fake_state.jobs,
        globus_l_fake_state.jobs);
    free(info);
}

globus_module_deactivate(GLOBUS_COMMON_MODULE);

return GLOBUS_SUCCESS;

6.9. Process Events

The main activity of our LRM module is to generate SEG events so that a job manager will be able to efficient manage its jobs. In this code, we will parse our log file periodically, and fire off any events which are to have occurred for the jobs in the fake job log. The structure of the processing function is this

static
void
globus_l_fake_read_callback(void * arg)
{
    Declare Variables
Check for Shutdown
Open Log
Read Log
Create Events
Emit Events
Reregister Callback
Error Handling
} /* globus_l_fake_read_callback() */

Declare Variables.

char                                jobid[64];
globus_list_t                       *l, *events;
fake_job_info_t                     *info;
globus_reltime_t                    delay = {0};
time_t                              now;
unsigned long                       pending_time, active_time, done_time, failed_time;
globus_scheduler_event_t            *e;
time_t                              last_timestamp;
globus_result_t                     result = GLOBUS_SUCCESS;

Check for Shutdown. To check for shutdown, we’ll first lock the mutex associated with the state structure and check if the shutdown_called field is set to true. If so, we’ll jump to our error handling code.

globus_mutex_lock(&globus_l_fake_state.mutex);
if (globus_l_fake_state.shutdown_called)
{
result = GLOBUS_FAILURE;
goto error;
}

**Open Log.** In general, we'll keep a file open to parse the log, but the first time around, or before any events have been written, the log file might not exist. So we'll check to see if we have a NULL file pointer, and if so, try to open the file. Once opened, we'll use line buffering while we process the file.

if (globus_l_fake_state.log == NULL)
{
    globus_l_fake_state.log = fopen(globus_l_fake_state.log_path, "r");
    if (globus_l_fake_state.log != NULL)
    {
        /* Enable line buffering */
        setvbuf(globus_l_fake_state.log, NULL, _IOLBF, 0);
    }
} if (globus_l_fake_state.log == NULL)
{
    result = GLOBUS_FAILURE;
    GlobusTimeReltimeSet(delay, 30, 0);
    goto reregister;
}

**Read Log.** Now we will read all of the log entries from our current position until the end of file. If we've already parsed an entry for a particular job, we will zero out its timestamps and replace with the new timestamps to handle cancel events in the fake job log.

/* previous read might have hit EOF, so clear it before trying to read */
clearerr(globus_l_fake_state.log);

/* Read any new job info from the log */
while (fscanf(globus_l_fake_state.log, "%63[^;];%ld;%ld;%ld;%ld\n", 
            jobid, 
            &pending_time, 
            &active_time, 
            &done_time, 
            &failed_time) == 5)
{
    l = globus_list_search_pred(globus_l_fake_state.jobs, globus_l_fake_find_by_job_id, jobid);
    if (l != NULL)
    {
        info = globus_list_first(l);
        /* If there's a second entry for the same job, it was cancelled, so */
        /* clear done/failed timestamps and copy them below */
        info->done = info->failed = 0;
    }
    else
    {
        /* First time we've seen this job, set jobid and insert*/
        info = malloc(sizeof(fake_job_info_t));
        *info = fake_job_info_initializer;
        strcpy(info->jobid, jobid);
globus_list_insert(&globus_l_fake_state.jobs, info);

} /* set timestamps */
info->pending = pending_time;
info->active = active_time;
info->done = done_time;
info->failed = failed_time;

Create Events. Now, we'll walk our list of jobs and create SEG events for each state transition which has occurred since our last timestamp and the current time. These events will be out of order in our events list, because they are created in order of job IDs in the jobs list, and not in timestamp list. We'll deal with this later. Note that we set the timestamp values in the job info to 0 after we create an event. This keeps us from generating an event multiple times.

/* Create set of events that we'll emit this time through: jobs which will changed state since our last event update */
now = time(NULL);

events = NULL;
for (l = globus_l_fake_state.jobs; l != NULL; l = globus_list_rest(l))
{
    info = globus_list_first(l);
    if (info->pending >= globus_l_fake_state.start_timestamp &&
        info->pending < now)
    {
        e = malloc(sizeof(globus_scheduler_event_t));
        e->event_type = GLOBUS_SCHEDULER_EVENT_PENDING;
        e->job_id = info->jobid;
        e->timestamp = info->pending;
        e->exit_code = 0;
        e->failure_code = 0;
        e->raw_event = NULL;
        info->pending = 0;
        globus_list_insert(&events, e);
    }
    if (info->active >= globus_l_fake_state.start_timestamp &&
        info->active < now)
    {
        e = malloc(sizeof(globus_scheduler_event_t));
        e->event_type = GLOBUS_SCHEDULER_EVENT_ACTIVE;
        e->job_id = info->jobid;
        e->timestamp = info->active;
        e->exit_code = 0;
        e->failure_code = 0;
        e->raw_event = NULL;
        info->active = 0;
        globus_list_insert(&events, e);
if (info->done != 0 &&
    info->done >= globus_l_fake_state.start_timestamp &&
    info->done < now)
{
    e = malloc(sizeof(globus_scheduler_event_t));
    e->event_type = GLOBUS_SCHEDULER_EVENT_DONE;
    e->job_id = info->jobid;
    e->timestamp = info->done;
    e->exit_code = 0;
    e->failure_code = 0;
    e->raw_event = NULL;

    info->done = 0;

    globus_list_insert(&events, e);
}
if (info->failed != 0 &&
    info->failed >= globus_l_fake_state.start_timestamp &&
    info->failed < now)
{
    e = malloc(sizeof(globus_scheduler_event_t));
    e->event_type = GLOBUS_SCHEDULER_EVENT_FAILED;
    e->job_id = info->jobid;
    e->timestamp = info->failed;
    e->exit_code = 0;
    e->failure_code = GLOBUS_GRAM_PROTOCOL_ERROR_USER_CANCELLED;
    e->raw_event = NULL;

    info->failed = 0;

    globus_list_insert(&events, e);
}
}

Emit Events. Now we have a set of events, we will sort them by timestamp and then use the SEG API to emit
them. After we've emitted an event, we have to free it. If the event is a terminal one (DONE or FAILED) we'll re-
move the job from the list of jobs in the state structure.

/* Sort the events so that they're in timestamp order */
events = globus_list_sort_destructive(events, globus_l_fake_compare_events, NULL);

/* Emit events in proper order */
while (! globus_list_empty(events))
{
    e = globus_list_remove(&events, events);
    last_timestamp = e->timestamp;

    switch (e->event_type)
    {
    case GLOBUS_SCHEDULER_EVENT_PENDING:
        globus_scheduler_event_pending(e->timestamp, e->job_id);
        break;
    case GLOBUS_SCHEDULER_EVENT_ACTIVE:
**LRM Adapter Tutorial**

```c
globus_scheduler_event_active(e->timestamp, e->job_id); break;

case GLOBUS_SCHEDULER_EVENT_FAILED:
    globus_scheduler_event_failed(e->timestamp, e->job_id, e->failure_code); break;

case GLOBUS_SCHEDULER_EVENT_DONE:
    globus_scheduler_event_done(e->timestamp, e->job_id, e->exit_code); break;
}

/* If this is a terminal event, we can remove the job from the list */
if (e->event_type == GLOBUS_SCHEDULER_EVENT_FAILED || e->event_type == GLOBUS_SCHEDULER_EVENT_DONE)
{
    l = globus_list_search_pred(globus_l_fake_state.jobs, globus_l_fake_find_by_job_id, e->job_id);
    info = globus_list_remove(&globus_l_fake_state.jobs, l);
    free(info);
}
free(e);
}
globus_l_fake_state.start_timestamp = last_timestamp;

**Reregister Callback.** We'll register a new callback instance now (provided we haven't had an error occur) so that we can continue to process jobs later.

GlobusTimeReltimeSet(delay, 1, 0);
reregister:
result = globus_callback_register_oneshot(
    NULL,
    &delay,
    globus_l_fake_read_callback,
    &globus_l_fake_state);
if (result != GLOBUS_SUCCESS)
{
    goto error;
}
globus_mutex_unlock(&globus_l_fake_state.mutex);
return;

**Error Handling.** If an error occurred registering the event or the shutdown handler is invoked, we'll exit this function without registering a new event. In the case the shutdown handling is in place, we'll signal that as well

error:
if (globus_l_fake_state.shutdown_called)
{
    globus_l_fake_state.callback_count--;

    if (globus_l_fake_state.callback_count == 0)
    {
        globus_cond_signal(&globus_l_fake_state.cond);
    }
}
else
{
    fprintf(stderr,
```
"FATAL: Unable to register callback. FAKE SEG exiting\n");
    exit(EXIT_FAILURE);
}
globus_mutex_unlock(&globus_l_fake_state.mutex);

return;

6.10. Utility Functions

We have two utility functions to implement for this module to manage our lists of pending events and jobs.

Find By Job ID

Sort Events

Find By Job ID. The globus_l_fake_find_by_job_id() function is used to search the jobs field of
the state structure for a fake_job_info_t containing info about a particular job. This predicate returns a non-zero value if the datum passed to the function has the same job ID as the arg parameter.

```c
static int
globus_l_fake_find_by_job_id(void * datum, void * arg)
{
    fake_job_info_t * info = datum;

    return (strcmp(info->jobid, arg) == 0);
} /* globus_l_fake_find_by_job_id() */
```

Sort Events. The globus_l_fake_compare_events() function is used as a predicate to compare the
timestamps and job ids of a pair of events. If the log_datum points to an event which happens earlier in the job lifecycle than the high_datum, this function returns GLOBUS_TRUE; otherwise it returns GLOBUS_FALSE.

```c
static int
globus_l_fake_compare_events(void * low_datum, void * high_datum, void * relation_args)
{
    globus_scheduler_event_t *low_event = low_datum, *high_event = high_datum;

    if (low_event->timestamp < high_event->timestamp)
    {
        return GLOBUS_TRUE;
    }
    else if (low_event->timestamp == high_event->timestamp)
    {
        if (low_event->event_type == GLOBUS_SCHEDULER_EVENT_PENDING)
        {
            return GLOBUS_TRUE;
        }
        else if (low_event->event_type == GLOBUS_SCHEDULEREVENT_ACTIVE &&
                 high_event->event_type != GLOBUS_SCHEDULER_EVENT_PENDING)
        {
            return GLOBUS_TRUE;
        }
    }
    else if (high_event->event_type != GLOBUS_SCHEDULER_EVENT_PENDING &&
```
7. Changes from Previous Versions

7.1. Changes in GT 5.2

GRAM5 is now designed to work as a native Debian or RPM package, with default configuration being done at configuration time, so the setup script description has been removed.

7.2. Changes in GT 5.0

GRAM5 is based again on the C code base used for GRAM2 (also known as Pre-WS GRAM). The SEG module interface from GRAM4 is retained and optionally used by GRAM5. The GRAM job manager will avoid reloading the GRAM LRM Adapter script for each operation, so all variables not intended to be global state in the Perl LRM Adapter module must be lexically scoped, or state will leak between jobs and cause potentially cause problems.

7.3. Changes in GT 4.0

7.3.1. Module Methods

The GT-4.0 ws-GRAM service only calls a subset of the Perl methods which were used by the pre-ws GRAM services. Most importantly for script implementors, the polling method is no longer used. Instead, the scheduler-event-generator monitors jobs to signal the service when job change changes occur. Staging is now done via the Reliable File Transfer service, so the file_stage_in and file_stage_out methods are no longer called.Schedulers typically did not implement the staging methods, so this shouldn't affect most scheduler modules.

That being said, scheduler implementers which would like to have their scheduler both with pre-ws GRAM and WS-GRAM should definitely implement the poll() method described in the pre-WS version of this tutorial.

7.3.2. GASS Cache

The GT-4.0 ws-GRAM service does not use the GASS cache for storing temporary files or for staging files.

7.4. Changes in GT 3.2

In GT 3.2, additional error message context info was added. Scripts can optionally add one of these fields to the return hash from an operation to provide extra error information to the client:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT3_FAILURE_MESSAGE</td>
<td>Error message from underlying script processing indicating what caused a job request to fail</td>
</tr>
<tr>
<td>GT3_FAILURE_TYPE</td>
<td>One of filestagein, filestageout, filestageinshared, executable, or stdin indicating what job request element caused a staging fault.</td>
</tr>
<tr>
<td>GT3_FAILURE_SOURCE</td>
<td>Source URL or file for a failed staging operation</td>
</tr>
</tbody>
</table>
GT3_FAILURE_DESTINATION  Destination URL or file for a failed staging operation
Part IV. GRAM5 Developer's Reference
Table of Contents

6. APIs .................................................................................................................. 63
   1. Programming Model Overview ...................................................................... 63
7. RSL Specification v1.1 ..................................................................................... 83
   1. RSL Syntax Overview .................................................................................. 83
   2. RSL Tokenization Overview ....................................................................... 84
   3. RSL Substitution Semantics ....................................................................... 85
   4. RSL Attribute Summary ............................................................................. 85
   5. Simple RSL Examples .................................................................................. 89
   6. RSL grammar and tokenization rules .......................................................... 90
8. Debugging .......................................................................................................... 92
   1. Basic Debugging Methods ........................................................................... 92
   2. Advanced Debugging Methods ................................................................... 93
9. Troubleshooting .................................................................................................. 94
   1. GRAM Client Troubleshooting ................................................................... 94
   2. Admin Troubleshooting ............................................................................. 97
   3. Errors ........................................................................................................... 99
10. Semantics and syntax of protocols .................................................................. 113
    1. GRAM5 Protocol ....................................................................................... 113
Chapter 6. APIs

1. Programming Model Overview

1.1. C API Documentation Links

Table 6.1. GRAM Client APIs

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAM Protocol¹</td>
<td>Low-level functions for processing GRAM protocol messages. Symbolic constants for RSL attributes, signals, and job states.</td>
</tr>
<tr>
<td>GRAM Client¹</td>
<td>Functions for submitting job requests, sending signals, and listening for job state updates.</td>
</tr>
<tr>
<td>RSL¹</td>
<td>Functions for parsing and manipulating job specifications in the RSL language.</td>
</tr>
<tr>
<td>Scheduler Event Generator¹</td>
<td>Functions for generating and parsing LRM-independent job state change events.</td>
</tr>
</tbody>
</table>

1.2. GRAM5 Perl API Reference

¹ http://www.globus.org/api/c-globus-5.2.2/globus_gram_protocol/html/
¹ http://www.globus.org/api/c-globus-5.2.2/globus_gram_client/html/
¹ http://www.globus.org/api/c-globus-5.2.2/globus_rsl/html/
¹ http://www.globus.org/api/c-globus-5.2.2/globus_scheduler_event_generator/html/
Name
Globus::GRAM::Error — GRAM Protocol Error Constants

DESCRIPTION

The Globus::GRAM::Error module defines symbolic names for the Error constants in the GRAM Protocol.

The Globus::GRAM::Error module methods return an object consisting of an integer error code, and (optionally) a string explaining the error.

Methods

$error = new Globus::GRAM::Error($number, $string);

Create a new error object with the given error number and string description. This is called by the error-specific factory methods described below.

$error->string()
Return the error string associated with a Globus::GRAM::Error object.

$error->value()
Return the integer error code associated with a Globus::GRAM::Error object.

$error = Globus::GRAM::Error::PARAMETER_NOT_SUPPORTED();

Create a new PARAMETER_NOT_SUPPORTED GRAM error.

$error = Globus::GRAM::Error::INVALID_REQUEST();

Create a new INVALID_REQUEST GRAM error.

$error = Globus::GRAM::Error::NO_RESOURCES();

Create a new NO_RESOURCES GRAM error.

$error = Globus::GRAM::Error::BAD_DIRECTORY();

Create a new BAD_DIRECTORY GRAM error.

$error = Globus::GRAM::Error::EXECUTABLE_NOT_FOUND();

Create a new EXECUTABLE_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::INSUFFICIENT_FUNDS();

Create a new INSUFFICIENT_FUNDS GRAM error.

$error = Globus::GRAM::Error::AUTHORIZATION();

Create a new AUTHORIZATION GRAM error.

$error = Globus::GRAM::Error::USER_CANCELLED();

Create a new USER_CANCELLED GRAM error.

$error = Globus::GRAM::Error::SYSTEM_CANCELLED();

Create a new SYSTEM_CANCELLED GRAM error.

$error = Globus::GRAM::Error::PROTOCOL_FAILED();

Create a new PROTOCOL_FAILED GRAM error.

$error = Globus::GRAM::Error::STDIN_NOT_FOUND();

Create a new STDIN_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::CONNECTION_FAILED();

Create a new CONNECTION_FAILED GRAM error.
$error = 
Globus::GRAM::Error::INVALID_MAXTIME()

Create a new INVALID_MAXTIME GRAM error.

$error = 
Globus::GRAM::Error::INVALID_COUNT()

Create a new INVALID_COUNT GRAM error.

$error = 
Globus::GRAM::Error::NULL_SPECIFICATION_TREE()

Create a new NULL_SPECIFICATION_TREE GRAM error.

$error = 
Globus::GRAM::Error::JM_FAILED_ALLOW_ATTACH()

Create a new JM_FAILED_ALLOW_ATTACH GRAM error.

$error = 
Globus::GRAM::Error::JOB_EXECUTION_FAILED()

Create a new JOB_EXECUTION_FAILED GRAM error.

$error = 
Globus::GRAM::Error::INVALID_PARADYN()

Create a new INVALID_PARADYN GRAM error.

$error = 
Globus::GRAM::Error::INVALID_JOBTYPE()

Create a new INVALID_JOBTYPE GRAM error.

$error = 
Globus::GRAM::Error::INVALID_GRAM_MYJOB()

Create a new INVALID_GRAM_MYJOB GRAM error.

$error = 
Globus::GRAM::Error::BAD_SCRIPT_ARG_FILE()

Create a new BAD_SCRIPT_ARG_FILE GRAM error.

$error = 
Globus::GRAM::Error::ARG_FILE_CREATION_FAILED()

Create a new ARG_FILE_CREATION_FAILED GRAM error.

$error = 
Globus::GRAM::Error::INVALID_JOBSTATE()

Create a new INVALID_JOBSTATE GRAM error.

$error = 
Globus::GRAM::Error::INVALID_SCRIPT_REPLY()

Create a new INVALID_SCRIPT_REPLY GRAM error.

$error = 
Globus::GRAM::Error::INVALID_SCRIPT_STATUS()

Create a new INVALID_SCRIPT_STATUS GRAM error.

$error = 
Globus::GRAM::Error::JOBTYPE_NOT_SUPPORTED()

Create a new JOBTYPE_NOT_SUPPORTED GRAM error.

$error = 
Globus::GRAM::Error::UNIMPLEMENTED()

Create a new UNIMPLEMENTED GRAM error.

$error = 
Globus::GRAM::Error::TEMP_SCRIPT_FILE_FAILED()

Create a new TEMP_SCRIPT_FILE_FAILED GRAM error.

$error = 
Globus::GRAM::Error::USER_PROXY_NOT_FOUND()

Create a new USER_PROXY_NOT_FOUND GRAM error.

$error = 
Globus::GRAM::Error::OPENING_USER_PROXY()

Create a new OPENING_USER_PROXY GRAM error.

$error = 
Globus::GRAM::Error::JOB_CANCEL_FAILED()

Create a new JOB_CANCEL_FAILED GRAM error.
$error = Globus::GRAM::Error::MALLOC_FAILED()

Create a new MALLOC_FAILED GRAM error.

$error = Globus::GRAM::Error::DUCT_INIT_FAILED()

Create a new DUCT_INIT_FAILED GRAM error.

$error = Globus::GRAM::Error::DUCT_LSP_FAILED()

Create a new DUCT_LSP_FAILED GRAM error.

$error = Globus::GRAM::Error::INVALID_HOST_COUNT()

Create a new INVALID_HOST_COUNT GRAM error.

$error = Globus::GRAM::Error::UNSUPPORTED_PARAMETER()

Create a new UNSUPPORTED_PARAMETER GRAM error.

$error = Globus::GRAM::Error::INVALID_QUEUE()

Create a new INVALID_QUEUE GRAM error.

$error = Globus::GRAM::Error::INVALID_PROJECT()

Create a new INVALID_PROJECT GRAM error.

$error = Globus::GRAM::Error::RSL_EVALUATION_FAILED()

Create a new RSL_EVALUATION_FAILED GRAM error.

$error = Globus::GRAM::Error::BAD_RSL_ENVIRONMENT()

Create a new BAD_RSL_ENVIRONMENT GRAM error.

$error = Globus::GRAM::Error::DRYRUN()

Create a new DRYRUN GRAM error.

$error = Globus::GRAM::Error::ZERO_LENGTH_RSL()

Create a new ZERO_LENGTH_RSL GRAM error.

$error = Globus::GRAM::Error::STAGING_EXECUTABLE()

Create a new STAGING_EXECUTABLE GRAM error.

$error = Globus::GRAM::Error::STAGING_STDIN()

Create a new STAGING_STDIN GRAM error.

$error = Globus::GRAM::Error::INVALID_JOB_MANAGER_TYPE()

Create a new INVALID_JOB_MANAGER_TYPE GRAM error.

$error = Globus::GRAM::Error::BAD_ARGUMENTS()

Create a new BAD_ARGUMENTS GRAM error.

$error = Globus::GRAM::Error::GATEKEEPER_MISCONFIGURED()

Create a new GATEKEEPER_MISCONFIGURED GRAM error.

$error = Globus::GRAM::Error::BAD_RSL()

Create a new BAD_RSL GRAM error.

$error = Globus::GRAM::Error::VERSION_MISMATCH()

Create a new VERSION_MISMATCH GRAM error.

$error = Globus::GRAM::Error::RSL_ARGUMENTS()

Create a new RSL_ARGUMENTS GRAM error.
$error = Globus::GRAM::Error::RSL_COUNT()  
Create a new RSL_COUNT GRAM error.

$error = Globus::GRAM::Error::RSL_DIRECTORY()  
Create a new RSL_DIRECTORY GRAM error.

$error = Globus::GRAM::Error::RSL_DRYRUN()  
Create a new RSL_DRYRUN GRAM error.

$error = Globus::GRAM::Error::RSL_ENVIRONMENT()  
Create a new RSL_ENVIRONMENT GRAM error.

$error = Globus::GRAM::Error::RSL_EXECUTABLE()  
Create a new RSL_EXECUTABLE GRAM error.

$error = Globus::GRAM::Error::RSL_HOST_COUNT()  
Create a new RSL_HOST_COUNT GRAM error.

$error = Globus::GRAM::Error::RSL_JOBTYPE()  
Create a new RSL_JOBTYPE GRAM error.

$error = Globus::GRAM::Error::RSL_MAXTIME()  
Create a new RSL_MAXTIME GRAM error.

$error = Globus::GRAM::Error::RSL_MYJOB()  
Create a new RSL_MYJOB GRAM error.

$error = Globus::GRAM::Error::RSL_PARADYN()  
Create a new RSL_PARADYN GRAM error.

$error = Globus::GRAM::Error::RSL_PROJECT()  
Create a new RSL_PROJECT GRAM error.

$error = Globus::GRAM::Error::RSL_QUEUE()  
Create a new RSL_QUEUE GRAM error.

$error = Globus::GRAM::Error::RSL_STDERR()  
Create a new RSL_STDERR GRAM error.

$error = Globus::GRAM::Error::RSL_STDIN()  
Create a new RSL_STDIN GRAM error.

$error = Globus::GRAM::Error::RSL.Stdout()  
Create a new RSL_STDOUT GRAM error.

$error = Globus::GRAM::Error::OPENING_JOBMANAGER_SCRIPT()  
Create a new OPENING_JOBMANAGER_SCRIPT GRAM error.

$error = Globus::GRAM::Error::CREATING_PIPE()  
Create a new CREATING_PIPE GRAM error.

$error = Globus::GRAM::Error::FCNTL_FAILED()  
Create a new FCNTL_FAILED GRAM error.

$error = Globus::GRAM::Error::STDOUT_FILENAME_FAILED()  
Create a new STDOUT_FILENAME_FAILED GRAM error.
$error = Globus::GRAM::Error::STDERR_FILENAME_FAILED() Create a new STDERR_FILENAME_FAILED GRAM error.

$error = Globus::GRAM::Error::FORKING_EXECUTABLE() Create a new FORKING_EXECUTABLE GRAM error.

$error = Globus::GRAM::Error::EXECUTABLE_PERMISSIONS() Create a new EXECUTABLE_PERMISSIONS GRAM error.

$error = Globus::GRAM::Error::OPENING_STDOUT() Create a new OPENING_STDOUT GRAM error.

$error = Globus::GRAM::Error::OPENING_STDERR() Create a new OPENING_STDERR GRAM error.

$error = Globus::GRAM::Error::OPENING_CACHE_USER_PROXY() Create a new OPENING_CACHE_USER_PROXY GRAM error.

$error = Globus::GRAM::Error::OPENING_CACHE() Create a new OPENING_CACHE GRAM error.

$error = Globus::GRAM::Error::INSERTING_CLIENT_CONTACT() Create a new INSERTING_CLIENT_CONTACT GRAM error.

$error = Globus::GRAM::Error::CLIENT_CONTACT_NOT_FOUND() Create a new CLIENT_CONTACT_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::CONTACTING_JOB_MANAGER() Create a new CONTACTING_JOB_MANAGER GRAM error.

$error = Globus::GRAM::Error::INVALID_JOB_CONTACT() Create a new INVALID_JOB_CONTACT GRAM error.

$error = Globus::GRAM::Error::UNDEFINED_EXE() Create a new UNDEFINED_EXE GRAM error.

$error = Globus::GRAM::Error::CONDOR_ARCH() Create a new CONDOR_ARCH GRAM error.

$error = Globus::GRAM::Error::CONDOR_OS() Create a new CONDOR_OS GRAM error.

$error = Globus::GRAM::Error::RSL_MIN_MEMORY() Create a new RSL_MIN_MEMORY GRAM error.

$error = Globus::GRAM::Error::RSL_MAX_MEMORY() Create a new RSL_MAX_MEMORY GRAM error.

$error = Globus::GRAM::Error::INVALID_MIN_MEMORY() Create a new INVALID_MIN_MEMORY GRAM error.

$error = Globus::GRAM::Error::INVALID_MAX_MEMORY() Create a new INVALID_MAX_MEMORY GRAM error.

$error = Globus::GRAM::Error::HTTP_FRAME_FAILED() Create a new HTTP_FRAME_FAILED GRAM error.
$error = Globus::GRAM::Error::HTTP_UNFRAME_FAILED()
Create a new HTTP_UNFRAME_FAILED GRAM error.

$error = Globus::GRAM::Error::HTTP_PACK_FAILED()
Create a new HTTP_PACK_FAILED GRAM error.

$error = Globus::GRAM::Error::HTTP_UNPACK_FAILED()
Create a new HTTP_UNPACK_FAILED GRAM error.

$error = Globus::GRAM::Error::INVALID_JOB_QUERY()
Create a new INVALID_JOB_QUERY GRAM error.

$error = Globus::GRAM::Error::SERVICE_NOT_FOUND()
Create a new SERVICE_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::JOB_QUERY_DENIAL()
Create a new JOB_QUERY_DENIAL GRAM error.

$error = Globus::GRAM::Error::CALLBACK_NOT_FOUND()
Create a new CALLBACK_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::BAD_GATEKEEPER_CONTACT()
Create a new BAD_GATEKEEPER_CONTACT GRAM error.

$error = Globus::GRAM::Error::POE_NOT_FOUND()
Create a new POE_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::MPIRUN_NOT_FOUND()
Create a new MPIRUN_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::RSL_START_TIME()
Create a new RSL_START_TIME GRAM error.

$error = Globus::GRAM::Error::RSL_RESERVATION_HANDLE()
Create a new RSL_RESERVATION_HANDLE GRAM error.

$error = Globus::GRAM::Error::RSL_MAX_WALL_TIME()
Create a new RSL_MAX_WALL_TIME GRAM error.

$error = Globus::GRAM::Error::INVALID_MAX_WALL_TIME()
Create a new INVALID_MAX_WALL_TIME GRAM error.

$error = Globus::GRAM::Error::RSL_MAX_CPU_TIME()
Create a new RSL_MAX_CPU_TIME GRAM error.

$error = Globus::GRAM::Error::INVALID_MAX_CPU_TIME()
Create a new INVALID_MAX_CPU_TIME GRAM error.

$error = Globus::GRAM::Error::SIGNALING_JOB()
Create a new SIGNALING_JOB GRAM error.
$error = Globus::GRAM::Error::UNKNOWN_SIGNAL_TYPE()
Create a new UNKNOWN_SIGNAL_TYPE GRAM error.

$error = Globus::GRAM::Error::GETTING_JOBID()
Create a new GETTING_JOBID GRAM error.

$error = Globus::GRAM::Error::WAITING_FOR_COMMIT()
Create a new WAITING_FOR_COMMIT GRAM error.

$error = Globus::GRAM::Error::COMMIT_TIMED_OUT()
Create a new COMMIT_TIMED_OUT GRAM error.

$error = Globus::GRAM::Error::RSL_SAVE_STATE()
Create a new RSL_SAVE_STATE GRAM error.

$error = Globus::GRAM::Error::RSL_RESTART()
Create a new RSL_RESTART GRAM error.

$error = Globus::GRAM::Error::RSL_TWO_PHASE_COMMIT()
Create a new RSL_TWO_PHASE_COMMIT GRAM error.

$error = Globus::GRAM::Error::INVALID_TWO_PHASE_COMMIT()
Create a new INVALID_TWO_PHASE_COMMIT GRAM error.

$error = Globus::GRAM::Error::RSL_STDOUT_POSITION()
Create a new RSL_STDOUT_POSITION GRAM error.

$error = Globus::GRAM::Error::INVALID_STDOUT_POSITION()
Create a new INVALID_STDOUT_POSITION GRAM error.

$error = Globus::GRAM::Error::RSL_STDERR_POSITION()
Create a new RSL_STDERR_POSITION GRAM error.

$error = Globus::GRAM::Error::INVALID_STDERR_POSITION()
Create a new INVALID_STDERR_POSITION GRAM error.

$error = Globus::GRAM::Error::RESTART_FAILED()
Create a new RESTART_FAILED GRAM error.

$error = Globus::GRAM::Error::NO_STATE_FILE()
Create a new NO_STATE_FILE GRAM error.

$error = Globus::GRAM::Error::READING_STATE_FILE()
Create a new READING_STATE_FILE GRAM error.

$error = Globus::GRAM::Error::WRITING_STATE_FILE()
Create a new WRITING_STATE_FILE GRAM error.

$error = Globus::GRAM::Error::OLD_JM_ALIVE()
Create a new OLD_JM_ALIVE GRAM error.

$error = Globus::GRAM::Error::TTL_EXPIRED()
Create a new TTL_EXPIRED GRAM error.

$error = Globus::GRAM::Error::SUBMIT_UNKNOWN()
Create a new SUBMIT_UNKNOWN GRAM error.
$error = Create a new RSL_REMOTE_IO_URL GRAM error.
Globus::GRAM::Error::RSL_REMOTE_IO_URL()

$error = Create a new WRITING_REMOTE_IO_URL GRAM error.
Globus::GRAM::Error::WRITING_REMOTE_IO_URL()

$error = Create a new STDIO_SIZE GRAM error.
Globus::GRAM::Error::STDIO_SIZE()

$error = Create a new JM_STOPPED GRAM error.
Globus::GRAM::Error::JM_STOPPED()

$error = Create a new USER_PROXY_EXPIRED GRAM error.
Globus::GRAM::Error::USER_PROXY_EXPIRED()

$error = Create a new JOB_UNSUBMITTED GRAM error.
Globus::GRAM::Error::JOB_UNSUBMITTED()

$error = Create a new INVALID_COMMIT GRAM error.
Globus::GRAM::Error::INVALID_COMMIT()

$error = Create a new RSL_SCHEDULER_SPECIFIC GRAM error.
Globus::GRAM::Error::RSL_SCHEDULER_SPECIFIC()

$error = Create a new STAGE_IN_FAILED GRAM error.
Globus::GRAM::Error::STAGE_IN_FAILED()

$error = Create a new INVALID_SCRATCH GRAM error.
Globus::GRAM::Error::INVALID_SCRATCH()

$error = Create a new RSL_CACHE GRAM error.
Globus::GRAM::Error::RSL_CACHE()

$error = Create a new INVALID_SUBMIT_ATTRIBUTE GRAM error.
Globus::GRAM::Error::INVALID_SUBMIT_ATTRIBUTE()

$error = Create a new INVALID_STDIO_UPDATE_ATTRIBUTE GRAM error.
Globus::GRAM::Error::INVALID_STDIO_UPDATE_ATTRIBUTE()

$error = Create a new INVALID_RESTART_ATTRIBUTE GRAM error.
Globus::GRAM::Error::INVALID_RESTART_ATTRIBUTE()

$error = Create a new RSL_FILE_STAGE_IN GRAM error.
Globus::GRAM::Error::RSL_FILE_STAGE_IN()

$error = Create a new RSL_FILE_STAGE_IN_SHARED GRAM error.
Globus::GRAM::Error::RSL_FILE_STAGE_IN_SHARED()

$error = Create a new RSL_FILE_STAGE_OUT GRAM error.
Globus::GRAM::Error::RSL_FILE_STAGE_OUT()

$error = Create a new RSL_GASS_CACHE GRAM error.
Globus::GRAM::Error::RSL_GASS_CACHE()

$error = Create a new RSL_FILE_CLEANUP GRAM error.
Globus::GRAM::Error::RSL_FILE_CLEANUP()
$error = Globus::GRAM::Error::RSL_SCRATCH()
Create a new RSL_SCRATCH GRAM error.

$error = Globus::GRAM::Error::INVALID_SCHEDULER_SPECIFIC()
Create a new INVALID_SCHEDULER_SPECIFIC GRAM error.

$error = Globus::GRAM::Error::UNDEFINED_ATTRIBUTE()
Create a new UNDEFINED_ATTRIBUTE GRAM error.

$error = Globus::GRAM::Error::INVALID_CACHE()
Create a new INVALID_CACHE GRAM error.

$error = Globus::GRAM::Error::INVALID_SAVE_STATE()
Create a new INVALID_SAVE_STATE GRAM error.

$error = Globus::GRAM::Error::OPENING_VALIDATION_FILE()
Create a new OPENING_VALIDATION_FILE GRAM error.

$error = Globus::GRAM::Error::READING_VALIDATION_FILE()
Create a new READING_VALIDATION_FILE GRAM error.

$error = Globus::GRAM::Error::RSL_PROXY_TIMEOUT()
Create a new RSL_PROXY_TIMEOUT GRAM error.

$error = Globus::GRAM::Error::INVALID_PROXY_TIMEOUT()
Create a new INVALID_PROXY_TIMEOUT GRAM error.

$error = Globus::GRAM::Error::STAGE_OUT_FAILED()
Create a new STAGE_OUT_FAILED GRAM error.

$error = Globus::GRAM::Error::JOB_CONTACT_NOT_FOUND()
Create a new JOB_CONTACT_NOT_FOUND GRAM error.

$error = Globus::GRAM::Error::DELEGATION_FAILED()
Create a new DELEGATION_FAILED GRAM error.

$error = Globus::GRAM::Error::LOCKING_STATE_LOCK_FILE()
Create a new LOCKING_STATE_LOCK_FILE GRAM error.

$error = Globus::GRAM::Error::INVALID_ATTR()
Create a new INVALID_ATTR GRAM error.

$error = Globus::GRAM::Error::NULL_PARAMETER()
Create a new NULL_PARAMETER GRAM error.

$error = Globus::GRAM::Error::STILL_STREAMING()
Create a new STILL_STREAMING GRAM error.

$error = Globus::GRAM::Error::AUTHORIZATION_DENIED()
Create a new AUTHORIZATION_DENIED GRAM error.

$error = Globus::GRAM::Error::AUTHORIZATION_SYSTEM_FAILURE()
Create a new AUTHORIZATION_SYSTEM_FAILURE GRAM error.

$error = Globus::GRAM::Error::AUTHORIZATION_DENIED_JOB_ID()
Create a new AUTHORIZATION_DENIED_JOB_ID GRAM error.
$error = Globus::GRAM::Error::AUTHORIZATION_DENIED_EXECUTABLE;
Create a new AUTHORIZATION_DENIED_EXECUTABLE GRAM error.

$error = Globus::GRAM::Error::RSL_USER_NAME;
Create a new RSL_USER_NAME GRAM error.

$error = Globus::GRAM::Error::INVALID_USER_NAME;
Create a new INVALID_USER_NAME GRAM error.

$error = Globus::GRAM::Error::LAST;
Create a new LAST GRAM error.
Name
Globus::GRAM::JobDescription — GRAM Job Description

Synopsis

use Globus::GRAM::JobDescription;

$hash = { executable => [ '/bin/echo' ], arguments => [ 'hello' ] };  
$description = new Globus::GRAM::JobDescription($filename);
$description = new Globus::GRAM::JobDescription($hash);
executable = $description->executable();
$description->add($new_attribute, $new_value);
$description->save();
$description->save($filename);
$description->print_recursive($file_handle);

DESCRIPTION

This object contains the parameters of a job request in a simple object wrapper. The object may be queried to determine the value of any RSL parameter, may be updated with new parameters, and may be saved in the filesystem for later use.

Methods

new
A JobDescription is constructed from a file consisting of a Perl hash of parameter => array mappings. Every value in the Job Description is stored internally as an array, even single literals, similar to the way an RSL tree is parsed in C. An example of such a file is

$description =
{
    executable => [ '/bin/echo' ],
    arguments => [ 'hello', 'world' ],
    environment => [
        ['GLOBUS_GRAM_JOB_CONTACT', 'https://globus.org:1234/2345/4332'],
    ]
};

which corresponds to the rsl fragment

$ (executable = /bin/echo)  
  (arguments = hello world)  
  (environment =  
    (GLOBUS_GRAM_JOB_CONTACT 'https://globus.org:1234/2345/4332'))

When the library_path RSL attribute is specified, this object modifies the environment RSL attribute value to append its value to any system specific variables.
$description->add('name', $value); Add a parameter to a job description. The parameter will be normalized internally so that the access methods described below will work with this new parameter. As an example,

$description->add('new_attribute', $new_value)

will create a new attribute in the JobDescription, which can be accessed by calling the $description->new_attribute() method.

$value $description->get('name'); Get a parameter from a job description. As an example,

$description->get('attribute')

will return the appropriate attribute in the JobDescription by name.

$description->save($filename); Save the JobDescription, including any added parameters, to the file named by $filename if present, or replacing the file used in constructing the object.

$description->print_recursive($file_handle) Write the value of the job description object to the file handle specified in the argument list.

$description->parameter(); For any parameter defined in the JobDescription can be accessed by calling the method named by the parameter. The method names are automatically created when the JobDescription is created, and may be invoked with arbitrary SillyCaps or underscores. That is, the parameter gram_myjob may be accessed by the GramMyJob, grammyjob, or gram_my_job method names (and others).

If the attributes does not in this object, then undef will be returned.

In a list context, this returns the list of values associated with an attribute.

In a scalar context, if the attribute's value consist of a single literal, then that literal will be returned, otherwise undef will be returned.

For example, from a JobDescription called $d constructed from a description file containing

```
{  executable => [ '/bin/echo' ],  arguments => [ 'hello', 'world' ]}
```

The following will hold:

```
$executable = $d->executable()    # '/bin/echo'
$arguments = $d->arguments()      # undef
@executable = $d->executable()    # ('/bin/echo')
@arguments = $d->arguments()      # ('hello', 'world')
$not_present = $d->not_present()  # undef
@not_present = $d->not_present()  # ()
```

To test for existence of a value:

```
@not_present = $d->not_present()
print "Not defined\n" if(!defined($not_present[0]));
```
Name
Globus::GRAM::JobManager — Base class for all Job Manager scripts

Synopsis

$manager = new Globus::GRAM::JobManager($job_description);

$manager->log("Starting new operation");
$manager->nfssync($fileobj,$createflag);
$manager->respond($hashref);
$hashref = $manager->submit();
$hashref = $manager->poll();
$hashref = $manager->cancel();
$hashref = $manager->signal();
$hashref = $manager->make_scratchdir();
$hashref = $manager->remove_scratchdir();
$hashref = $manager->rewrite_urls();
$hashref = $manager->stage_in();
$hashref = $manager->stage_out();
$hashref = $manager->cache_cleanup();
$hashref = $manager->remote_io_file_create();
$hashref = $manager->proxy_relocate();
$hashref = $manager->proxy_update();
$scalar = $manager->pipe_out_cmd(@arglist);
($stderr, $rc) = $manager->pipe_err_cmd(@arglist);
$status = $manager->fork_and_exec_cmd(@arglist);
$manager->append_path($hash, $variable, $path);
$scalar = $manager->setup_softenv();

DESCRIPTION

The Globus::GRAM::JobManager module implements the base behavior for a Job Manager script interface. Scheduler-specific job manager scripts must inherit from this module in order to be used by the job manager.

Methods

$manager = Globus::GRAM::JobManager->new($JobDescription) Each Globus::GRAM::JobManager object is created by calling the constructor with a single argument, a Globus::GRAM::JobDescription object containing the information about the job request which the script will be modifying. Modules which subclass Globus::GRAM::JobManager MUST call the super-class's constructor, as in this code fragment:

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

bless $self, $class;

$manager->log($string) Log a message to the job manager log file. The message is preceded by a timestamp.

$manager->nfssync($object,$create) Send an NFS update by touching the file (or directory) in question. If the $create is true, a file will be created. If it is false, the $object will not be created.
$manager->respond($message) Send a response to the job manager program. The response may either be a hash reference consisting of a hash of (variable, value) pairs, which will be returned to the job manager, or an already formatted string. This only needs to be directly called by a job manager implementation when the script wants to send a partial response while processing one of the scheduler interface methods (for example, to indicate that a file has been staged).

The valid keys for a response are defined in the RESPONSES section.

$manager->submit() Submit a job request to the scheduler. The default implementation returns with the Globus::GRAM::Error::UNIMPLEMENTED error. Scheduler specific subclasses should reimplement this method to submit the job to the scheduler.

A scheduler which implements this method should return a hash reference containing a scheduler-specific job identifier as the value of the hash's JOB_ID key, and optionally, the a GRAM job state as the value of the hash's JOB_STATE key if the job submission was successful; otherwise a Globus::GRAM::Error value should be returned. The job state values are defined in the Globus::GRAM::JobState module. The job parameters (as found in the job rsl) are defined in Globus::GRAM::Jobdescription object in $self->{JobDescription}.

For example:

```perl
return {JOB_STATE => Globus::GRAM::JobState::PENDING,
        JOB_ID => $job_id};
```

$manager->poll() Poll a job's status. The default implementation returns with the Globus::GRAM::Error::UNIMPLEMENTED error. Scheduler specific subclasses should reimplement this method to poll the scheduler.

A scheduler which implements this method should return a hash reference containing the JOB_STATE value. The job's ID can be accessed by calling the $self->{JobDescription}->{jobid()} method.

$manager->cancel() Cancel a job. The default implementation returns with the Globus::GRAM::Error::UNIMPLEMENTED error. Scheduler specific subclasses should reimplement this method to remove the job from the scheduler.

A scheduler which implements this method should return a hash reference containing the JOB_STATE value. The job's ID can be accessed by calling the $self->{JobDescription}->{jobid()} method.

$manager->signal() Signal a job. The default implementation returns with the Globus::GRAM::Error::UNIMPLEMENTED error. Scheduler specific subclasses should reimplement this method to remove the job from the scheduler. The JobManager module can determine the job's ID, the signal number, and the (optional) signal arguments from the Job Description by calling its job_id(), signal(), and signal_arg() methods, respectively.

Depending on the signal, it may be appropriate for the JobManager object to return a hash reference containing a JOB_STATE update.

$manager->make_scratchdir() Create a scratch directory for a job. The scratch directory location is based on the JobDescription's scratch_dir_base() and scratch_dir() methods.
If the scratch_dir() value is a relative path, then a directory will be created as a subdirectory of scratch_dir_base()/scratch_dir(), otherwise, it will be created as a subdirectory of scratch_dir(). This method will return a hash reference containing mapping SCRATCH_DIR to the absolute path of newly created scratch directory if successful.

$manager->remove_scratchdir() Delete a job's scratch directory. All files and subdirectories of the JobDescription's scratch_directory() will be deleted.

$manager->file_cleanup() Delete some job-related files. All files listed in the JobDescription's file_cleanup() array will be deleted.

$manager->rewrite_urls() Looks up URLs listed in the JobDescription's stdin() and executable(), and replaces them with paths to locally cached copies.

$manager->stage_in() Stage input files need for the job from remote storage. The files to be staged are defined by the array of [URL, path] pairs in the job description's file_stage_in() and file_stage_in_shared() methods. The Globus::GRAM::JobManager module provides an implementation of this functionality using the globus-url-copy and globus-gass-cache programs. Files which are staged in are not automatically removed when the job terminates.

This function returns intermediate responses using the Globus::GRAM::JobManager::response() method to let the job manager know when each individual file has been staged.

$manager->stage_out() Stage output files generated by this job to remote storage. The files to be staged are defined by the array of [URL, destination] pairs in the job description's file_stage_out() method. The Globus::GRAM::JobManager module provides an implementation of this functionality using the globus-url-copy program. Files which are staged out are not removed by this method.

$manager->cache_cleanup() Clean up cache references in the GASS which match this job's cache tag.

$manager->remote_io_file_create() Create the remote I/O file in the job dir which will contain the remote_io_url RSL attribute's value.

$manager->proxy_relocate() Relocate the delegated proxy for job execution. Job Managers need to override the default if they intend to relocate the proxy into some common file system other than the cache. The job manager program does not depend on the new location of the proxy. Job Manager modules must not remove the default proxy.

$hashref = $manager->proxy_update();

$manager->append_path($ref, $var, $path) Append $path to the value of $ref->{$var}, dealing with the case where $ref->{$var} is not yet defined.

$manager->pipe_out_cmd(@arg) Create a new process to run the first argument application with the remaining arguments (which may be empty). No shell metacharacter will be evaluated, avoiding a shell invocation. Stderr is redirected to /dev/null and stdout is being captured by the parent process, which is also the result returned. In list mode, all lines are returned, in scalar mode, only the first line is being re-
turned. The line termination character is already cut off. Use this function as more efficient backticks, if you do not need shell metacharacter evaluation.

Caution: This function deviates in two manners from regular backticks. Firstly, it chomps the line terminator from the output. Secondly, it returns only the first line in scalar context instead of a multiline concatenated string. As with regular backticks, the result may be undefined in scalar context, if no result exists.

A child error code with an exit code of 127 indicates that the application could not be run. The scalar result returned by this function is usually undef'ed in this case.

($stderr, $rc) = $manager->pipe_err_cmd(@arg)

Create a new process to run the first argument application with the remaining arguments (which may be empty). No shell metacharacter will be evaluated, avoiding a shell invocation.

This method returns a list of two items, the standard error of the program, and the exit code of the program. If the error code is 127, then the application could not be run. Standard output is discarded.

$manager->fork_and_exec_cmd(@arg)

Fork off a child to run the first argument in the list. Remaining arguments will be passed, but shell interpolation is avoided. Signals SIGINT and SIGQUIT are ignored in the child process. Stdout is appended to /dev/null, and stderr is dup2 from stdout. The parent waits for the child to finish, and returns the value for the CHILD_ERROR variable as result. Use this function as more efficient system() call, if you can do not need shell metacharacter evaluation.

Note that the inability to execute the program will result in a status code of 127.

$manager->job_dir()

Return the temporary directory to store job-related files, which have no need for file caching.

$manager->setup_softenv()

Either add a line to the specified command script file handle to load the user's default SoftEnv configuration, or create a custom SoftEnv script and add commands to the specified command script file handle to load it.

**RESPONSES**

When returning from a job interface method, or when sending an intermediate response via the response() method, the following hash keys are valid:

* **JOB_STATE**
  An integer job state value. These are enumerated in the Globus::GRAM::JobState module.

* **ERROR**
  An integer error code. These are enumerated in the Globus::GRAM::Error module.

* **JOB_ID**
  A string containing a job identifier, which can be used to poll, cancel, or signal a job in progress. This response should only be returned by the submit method.

* **SCRATCH_DIR**
  A string containing the path to a newly-created scratch directory. This response should only be returned by the make_scratchdir method.

* **STAGED_IN**
  A string containing the (URL, path) pair for a file which has now been staged in. This response should only be returned by the stage_in method.
* STAGED_IN_SHARED  A string containing the (URL, path) pair for a file which has now been staged in and symlinked from the cache. This response should only be returned by the `stage_in_shared` method.

* STAGED_OUT  A string containing the (path, URL) pair for a file which has now been staged out by the script. This response should only be returned by the `stage_out` method.
**Name**
Globus::GRAM::JobSignal — GRAM Protocol JobSignal Constants

**DESCRIPTION**
The Globus::GRAM::JobSignal module defines symbolic names for the JobSignal constants in the GRAM Protocol.

**Methods**

```perl
$value = Globus::GRAM::CANCEL()  # Return the value of the CANCEL constant.
$value = Globus::GRAM::SUSPEND()  # Return the value of the SUSPEND constant.
$value = Globus::GRAM::RESUME()  # Return the value of the RESUME constant.
$value = Globus::GRAM::PRIORITY()  # Return the value of the PRIORITY constant.
$value = Globus::GRAM::COMMIT_REQUEST()  # Return the value of the COMMIT_REQUEST constant.
$value = Globus::GRAM::COMMIT_EXTEND()  # Return the value of the COMMIT_EXTEND constant.
$value = Globus::GRAM::STDIO_UPDATE()  # Return the value of the STDIO_UPDATE constant.
$value = Globus::GRAM::STDIO_SIZE()  # Return the value of the STDIO_SIZE constant.
$value = Globus::GRAM::STOP_MANAGER()  # Return the value of the STOP_MANAGER constant.
$value = Globus::GRAM::COMMIT_END()  # Return the value of the COMMIT_END constant.
```
Name
Globus::GRAM::JobState — GRAM Protocol JobState Constants

DESCRIPTION
The Globus::GRAM::JobState module defines symbolic names for the JobState constants in the GRAM Protocol.

Methods

$value = $value = $value = $value = $value = $value = $value = $value = Globus::GRAM::PENDING() Globus::GRAM::ACTIVE() Globus::GRAM::FAILED() Globus::GRAM::SUSPENDED() Globus::GRAM::UNSUBMITTED() Globus::GRAM::STAGE_IN() Globus::GRAM::STAGE_OUT() Globus::GRAM::ALL() Return the value of the PENDING constant. Return the value of the ACTIVE constant. Return the value of the FAILED constant. Return the value of the SUSPENDED constant. Return the value of the UNSUBMITTED constant. Return the value of the STAGE_IN constant. Return the value of the STAGE_OUT constant. Return the value of the ALL constant.
Chapter 7. RSL Specification v1.1

This is a document to specify the existing RSL v1.0 implementation and interfaces, as they are provided in the GT 5.2.2 release. This document serves as a reference, and more introductory text.

The Globus Resource Specification Language (RSL) provides a common interchange language to describe resources. The various components of the Globus Resource Management architecture manipulate RSL strings to perform their management functions in cooperation with the other components in the system. The RSL provides the skeletal syntax used to compose complicated resource descriptions, and the various resource management components introduce specific ATTRIBUTE,VALUE> pairings into this common structure. Each attribute in a resource description serves as a parameter to control the behavior of one or more components in the resource management system.

1. RSL Syntax Overview

The core syntax of the RSL syntax is the relation. Relations associate an attribute name with a value, e.g., the relation executable=a.out provides the name of an executable in a resource request. There are two generative syntactic structures in the RSL that are used to build more complicated resource descriptions out of the basic relations: compound requests and value sequences. In addition, the RSL syntax includes a facility to both introduce and dereference string substitution variables.

The simplest form of compound request, utilized by all resource management components, is the conjunct-request. The conjunct-request expresses a conjunction of simple relations or compound requests (like a boolean AND). The most common conjunct-request in Globus RSL strings is the combination of multiple relations such as executable name, node count, executable arguments, and output files for a basic GRAM job request. Similarly, the core RSL syntax includes a disjunct-request form to represent disjunctive relations (like a boolean OR). Currently, however, no resource management component utilizes the disjunct-request form.

The last form of compound request is the multi-request. The multi-request expresses multiple parallel resources that make up a resource description. The multi-request form differs from the conjunction and disjunction in two ways: multi-requests introduce new variable scope, meaning variables defined in one clause of a multi-request are not visible to the other clauses, and multi-requests introduce a non-reducible hierarchy to the resource description. Whereas relations within a conjunct-request can be thought of as constraints on the resource being described, the subclauses of a multi-request are best thought of as individual resource descriptions that together constitute an abstract resource collection; the same attributes may be constrained in different ways in each subclause without causing a logical contradiction. An example of a contradiction would be to constrain the executable attribute to be two conflicting values within a conjunction. Currently, however, no resource management component utilizes the disjunct-request form.

The simplest form of value in the RSL syntax is the string literal. When explicitly quoted, literals can contain any character, and many common literals that don't contain special characters can appear without quotes. Values can also be variable references, in which case the variable reference is in essence replaced with the string value defined for that variable. RSL descriptions can also express string-concatenation of values, especially useful to construct long strings out of several variable references. String concatenation is supported with both an explicit concatenation operator and implicit concatenation for many idiomatic constructions involving variable references and literals.

In addition to the simple value forms given above, the RSL syntax includes the value sequence to express ordered sets of values. The value sequence syntax is used primarily for defining variables and for providing the argument list for a program.
2. RSL Tokenization Overview

Each RSL string consists of a sequence of RSL tokens, whitespace, and comments. The RSL tokens are either special syntax or regular unquoted literals, where special syntax contains one or more of the following listed special characters and unquoted literals are made of sequences of characters excluding the special characters.

The complete set of special characters that cannot appear as part of an unquoted literal is:

- + (plus)
- & (ampersand)
- | (pipe)
- ( (left paren)
- ) (right paren)
- = (equal)
- < (left angle)
- > (right angle)
- ! (exclamation)
- " (double quote)
- ' (apostrophe)
- ^ (carat)
- # (pound)
- $ (dollar)

These characters can only be used for the special syntactic forms described in the section and in the section or as within quoted literals.

Quoted literals are introduced with the " (double quote) or ' (single quote/apostrophe) and consist of all the characters up to (but not including) the next solo double or single quote, respectively. To escape a quote character within a quoted literal, the appearance of the quote character twice in a row is converted to a single instance of the character and the literal continues until the next solo quote character. For any quoted literal, there is only one possible escape sequence, eg within a literal delimited by the single quote character only the single quote character uses the escape notation and the double quote character can appear without escape.

Quoted literals can also be introduced with an alternate user delimiter notation. User delimited literals are introduced with the ^ (carat) character followed immediately by a user-provided delimiter; the literal consists of all the characters after the user’s delimiter up to (but not including) the next solo instance of the delimiter. The delimiter itself may be escaped within the literal by providing two instances in a row, just as the regular quote delimiters are escaped in regular quoted literals.

RSL string comments use a notation similar to comments in the C programming language. Comments are introduced by the prefix (* and Comments continue to the first terminating suffix *) and cannot be nested. Comments are stripped from the RSL string during processing and are syntactically equivalent to whitespace.
Example 7.1. Quoted Literal Examples

Assign the value Hello. Welcome to "The Grid" to the attribute arguments, using double-quote as the delimiter and the escaping sequence.

arguments = "Hello. Welcome to ""The Grid""

Assign the value Hello. Welcome to "The Grid" to the attribute arguments using the single-quote delimiter.

arguments = 'Hello. Welcome to "The Grid'

Assign the value Hello. Welcome to "The Grid" to the attribute arguments using a user-defined quoting character !.

arguments = ^!Hello. Welcome to "The Grid"!

3. RSL Substitution Semantics

RSL strings can introduce and reference string variables. String substitution variables are defined in a special relation using the rsl_substitution attribute, and the definitions affect variable references made in the same conjunct-request (or disjunct-request), as well as references made within any multi-request nested inside one of the clauses of the conjunction (or disjunction). Each multi-request introduces a new variable scope for each subrequest, and variable definitions do not escape the closest enclosing scope.

Within any given scope, variable definitions are processed left-to-right in the resource description. Outermost scopes are processed before inner scopes, and the definitions in inner scopes augment the inherited definitions with new and/or updated variable definitions.

Variable definitions and variable references are processed in a single pass, with each definition updating the environment prior to processing the next definition. The value provided in a variable definition may include a reference to a previously-defined variable. References to variables that are not yet provided with definitions in the standard RSL variable processing order are replaced with an empty literal string.

4. RSL Attribute Summary

The RSL syntax is extensible because it defines structure without too many keywords. Each Globus resource management component introduces additional attributes to the set recognized by RSL-aware components, so it is difficult to provide a complete listing of attributes which might appear in a resource description. Resource management components are designed to utilize attributes they recognize and pass unrecognized relations through unchanged. This allows powerful compositions of different resource management functions.

The following listing summarizes the attribute names utilized by existing resource management components in the standard Globus release. Please see the individual component documentation for discussion of the attribute semantics.
### Name
**rsl** — GRAM5 RSL Attributes

### Description

**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. [Default: 1]

**directory**
Specifies the path of the directory the jobmanager will use as the default directory for the requested job. [Default: $(HOME)]

**dry_run**
If dryrun = yes then the jobmanager will not submit the job for execution and will return success. [Default: no]

**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.

**expiration**
Time (in seconds) after a a job fails to receive a two-phase commit end signal before it is cleaned up. [Default: 14400]

**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. [Default: collective]

**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.) [Default: multiple]

**library_path**
Specifies a list of paths to be appended to the system-specific library path environment variables. [Default: $(GLOBUS_LOCATION)/lib]

**loglevel**
Override the default log level for this job. The value of this attribute consists of a combination of the strings FATAL, ERROR, WARN, INFO, DEBUG, TRACE joined by the | character
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logpattern</td>
<td>Override the default log path pattern for this job. The value of this attribute is a string (potentially containing RSL substitutions) that is evaluated to the path to write the log to. If the resulting string contains the string $(DATE) (or any other RSL substitution), it will be reevaluated at log time.</td>
</tr>
<tr>
<td>max_cpu_time</td>
<td>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.</td>
</tr>
<tr>
<td>max_memory</td>
<td>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.</td>
</tr>
<tr>
<td>max_time</td>
<td>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.</td>
</tr>
<tr>
<td>max_wall_time</td>
<td>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.</td>
</tr>
<tr>
<td>min_memory</td>
<td>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.</td>
</tr>
<tr>
<td>project</td>
<td>Target the job to be allocated to a project account as defined by the scheduler at the defined (remote) resource.</td>
</tr>
<tr>
<td>proxy_timeout</td>
<td>Obsolete and ignored. Now a job-manager-wide setting.</td>
</tr>
<tr>
<td>queue</td>
<td>Target the job to a queue (class) name as defined by the scheduler at the defined (remote) resource.</td>
</tr>
<tr>
<td>remote_io_url</td>
<td>Writes the given value (a URL base string) to a file, and adds the path to that file to the environment through 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.</td>
</tr>
<tr>
<td>restart</td>
<td>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.</td>
</tr>
<tr>
<td>rsl_substitution</td>
<td>Specifies a list of values which can be substituted into other rsl attributes' values through the $(SUBSTITUTION) mechanism.</td>
</tr>
</tbody>
</table>
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.

savejobdescription

Save a copy of the job description to $HOME [Default: no]

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. There are two accepted forms of this value. It can consist of a single destination: stderr = URL, or a sequence of destinations: stderr = (DESTINATION) (DESTINATION). In the latter case, the DESTINATION may itself be a URL or a sequence of an x-gass-cache URL followed by a cache tag. [Default: /dev/null]

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. [Default: /dev/null]

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. There are two accepted forms of this value. It can consist of a single destination: stdout = URL, or a sequence of destinations: stdout = (DESTINATION) (DESTINATION). In the latter case, the DESTINATION may itself be a URL or a sequence of an x-gass-cache URL followed by a cache tag. [Default: /dev/null]

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.

5. Simple RSL Examples

The following are some simple example RSL strings to illustrate idiomatic usage with existing tools and to make concrete some of the more interesting cases of tokenization, concatenation, and variable semantics. These are meant to illustrate the use of the RSL notation without much regard for the specific details of a particular resource management component.

Typical GRAM5 resource descriptions contain at least a few relations in a conjunction:
Example 7.2. GRAM5 Job Request Examples

This example shows a conjunct request containing values that are unquoted literals and ordered sequences of a mix of quoted and unquoted literals.

(* this is a comment *)

& (executable = a.out (* <-- that is an unquoted literal *))
  (directory = /home/nobody)
  (arguments = arg1 "arg 2")
  (count = 1)

This example demonstrates RSL substitutions, which can be used to make sure a string is used consistently multiple times in a resource description:

& (rsl_substitution = (TOPDIR "/home/nobody")
  (DATADIR $(TOPDIR)/"data")
  (EXECDIR $(TOPDIR)/bin)
  (executable = $(EXECDIR)/a.out
    (* ^-- implicit concatenation *))
  (directory = $(TOPDIR))
  (arguments = $(DATADIR)/file1
    (* ^-- implicit concatenation *)
    $(DATADIR) # /file2
    (* ^-- explicit concatenation *)
    "$(FOO)"
    (* <-- a quoted literal *))
  (environment = (DATADIR $(DATADIR)))
  (count = 1)

Performing all variable substitution and removing comments yields an equivalent RSL string:

& (rsl_substitution = (TOPDIR "/home/nobody")
  (DATADIR "/home/nobody/data")
  (EXECDIR "/home/nobody/bin")
  (executable = "/home/nobody/bin/a.out"
    (directory = "/home/nobody"
      (arguments = "/home/nobody/data/file1"
        "/home/nobody/data/file2"*
        "$ (FOO) "
        (environment = (DATADIR "/home/nobody/data"))
        (count = 1)

Note in the above variable-substitution example, the variable substitution definitions are not automatically made a part of the job's environment. And explicit environment attribute must be used to add environment variables for the job. Also note that the third value in the arguments clause is not a variable reference but only quoted literal that happens to contain one of the special characters.

6. RSL grammar and tokenization rules

The following is a modified BNF grammar for the Resource Specification Language. Lexical rules are provided for the implicit concatenation sequences in the form of conventional regular expressions; for the implicit-concat non-terminal rules, whitespace is not allowed between juxtaposed non-terminals. Grammar comments are provided in square brackets in a column to the right of the productions, eg [comment] to help relate productions in the grammar to the terminology used in the above discussion.
Regular expressions are provided for the terminal class `string-literal` and for RSL comments. These regular expressions make use of a common inverted character-class notation, as popularized by the various lex tools. Comments are syntactically equivalent to whitespace and can only appear where the comment prefix cannot be mistaken for the trailing part of a multi-character unquoted literal.

**RSL Grammar**

1. `specification ::= relation
   | '+' spec-list
   | '&' spec-list
   | '|' spec-list`  /* relation */, /* multi-request */, /* conjunct-request */, /* disjunct-request */
2. `spec-list ::= (' specification ') spec-list
   | (' specification ')'`  /* Substitution variable definition */
3. `relation ::= 'rsl_substitution'= binding-sequence
   | attribute op value-sequence`  /* Attribute binding relation */
4. `binding-sequence ::= binding binding-sequence
   | binding`  /* String */, /* Concatenation */
5. `binding ::= (' string-literal simple-value ')'`  /* Variable Reference */
6. `attribute ::= string-literal`  /* attribute */
7. `op ::= '=' | '!=' | '>' | '>=' | '<' | '<='`  /* Substitution variable definition */
8. `value-sequence ::= value value-sequence
   | value`  /* String */, /* Concatenation */
9. `value ::= (' value-sequence ') simple-value
   | simple-value`  /* String */, /* Concatenation */
10. `simple-value ::= string-literal
    | implicit-concat
    | variable-reference`  /* String */, /* Concatenation */
11. `variable-reference ::= $(' string-literal ')'
12. `implicit-concat ::= (unquoted-literal)? (implicit-concat-core)+`  /* Implicit concatenation */
13. `implicit-concat-core ::= variable-reference
    | (variable-reference) (unquoted-literal)`  /* Implicit concatenation */
14. `string-literal ::= quoted-literal
    | unquoted-literal`  /* Single-quote delimiter with escaping */, /* Double-quote delimiter with escaping */, /* User defined delimiter c with escaping */
15. `quoted-literal ::= " (("(")) | ("\") | ("\")\" | ("\")*" | (\c)((\c)(cc))\c`  /* Non-special characters */
16. `unquoted-literal ::= ((\^\v\n+&|()=<>!"'^#$\))+
   | comment ::= '*( ((\*))((\*))\#*)'`  /* Comment */
Chapter 8. Debugging

Log output from GRAM5 is a useful tool for debugging issues. GRAM5 can log to either local files or syslog. See the Admin Guide for information about how to configure logging.

In most cases, logging at the INFO level will produce enough information to show progress of most operations. Adding DEBUG will also allow log information from the GRAM LRM scripts.

1. Basic Debugging Methods

The first thing to determine when debugging unexpected failures is to determine whether the gatekeeper service is running, reachable from the client, and properly configured.

First, determine that the gatekeeper is running by using a tool such as telnet to connect to the TCP/IP port that the gatekeeper is listening on. From the GRAM service node, using a default configuration, use a command like:

```bash
% telnet localhost 2119
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'
```

An error message like the following indicates that the gatekeeper service is not starting:

```
telnet: connect to address 127.0.0.1: Connection refused
telnet: Unable to connect to remote host
```

If the telnet command exits immediately, then the gatekeeper service is being started but not running. Check the gatekeeper log (by default $GLOBUS_LOCATION/var/globus-gatekeeper.log) to see if there is an error message. A common error is having a missing library path environment variable in the gatekeeper's environment or having a malformed configuration file. See the globus-gatekeeper for information on the configuration options.

The next recommended diagnostic is to run the same telnet command from the machine which is acting as the GRAM client if it is distinct from the GRAM service node. Be sure to replace localhost with the actual host name of the GRAM service. Again, check for log entries in the case of immediate exit or refused connection. If the connection does not work, then there may be some network connectivity or firewall issues preventing access.

Next use a tool like globusrun to diagnose whether the client is authorized to contact the gatekeeper service. This is done by using the -a command-line option. For example:

```bash
% globusrun -a -r grid.example.org
GRAM Authentication test successful
```

If you do not get the success message above, then check the gatekeeper log to see if there is a diagnostic message. A common problem is that the identity of the client is not in the grid mapfile used by the gatekeeper.

The next test is to use the -dryrun option to globusrun to verify that the job manager service is properly configured. To do so, try the following:

```bash
% globusrun -dryrun -r grid.example.org "& (executable=/bin/sh)"
globus_gram_client_callback_allow successful
Dryrun successful
```

If you do not get the success message above, first check the error number in the GRAM5 Error codes table to determine how to proceed. If the result is unclear, check the job manager log (default $HOME/gram_DATE.log) to see if there are any further details of the error.
The final test is to submit a test job to the GRAM5 service and wait for it to terminate, such as this example shows:

```
% globus-job-run grid.example.org /bin/sh -c 'echo "hello, grid"'
hello, grid
```

If the process appears to hang, it might be that the job manager is unable to send state callbacks to the client. Check that there are no firewalls or network issues that would prevent the job manager process from connecting from the GRAM service node to the client node.

## 2. Advanced Debugging Methods

The methods described in this section are intended for debugging problems in the GRAM code, not in the user environment.

### 2.1. Debugging the Job Manager

To debug the GRAM5 job manager, run the command located in $GLOBUS_LOCATION/etc/grid-services/jobmanager-LRM (ignoring the first 3 fields). For example:

```
% $GLOBUS_LOCATION/libexec/globus-job-manager -conf $GLOBUS_LOCATION/etc/globus-job-manager.conf -type fork
```

When the job manager is started in this way, it will log messages to standard error and will terminate 60 seconds after its last job has completed. This only works if there are no job managers running for this particular user. The job manager can be started in a debugger such as gdb or valgrind using a similar command-line.
Chapter 9. Troubleshooting

For a list of error codes generated by GRAM5, see Section 3, “Errors”.

1. GRAM Client Troubleshooting

1.1. Credential Problems

GRAM requires a client certificate and private key in order to authenticate with the GRAM service. If these are not available, the GRAM client will fail. In typical use, a user will create a temporary proxy certificate either derived from their identity certificate issued by some certificate authority, or from a service such as myproxy. If a GRAM client command returns any error containing the string `GSS Major Status`, you’ve hit a credential problem. Look at the Troubleshooting Section of the GSI manual for details about how to diagnose and correct these errors. The `grid-cert-diagnostics` tool with the `-p` command-line option is especially helpful for diagnosing some of these types of problems.

1.2. Connection Problems

There are a few things which can go wrong when trying to contact a GRAM service. These have slightly different error types which can help diagnose which problem is occurring.

1.2.1. Invalid Resource Name

If the hostname or TCP port you are using for a GRAM resource name is not correct, then the GRAM client will be unable to access the service. Errors of this type will look like this:

```
% globus-job-run grid.example.org/jobmanager-fork /bin/hostname
GRAM Job submission failed because the connection to the server failed (check host and port) (error code 12)
```

When this occurs, check with the resource administrator for correct resource naming so that you can contact the service.

1.2.2. Mutual Authentication Failure

GRAM performs mutual authentication, that is, both the client and service provide certificates indicating who they are. The service uses the client’s identity to map the user to a local unix account. The client uses the server’s identity to verify that the service is running with a host credential. The failure of the client to trust the server’s certificate will generate an error message that looks like this: `globus_gsi_gssapi: Authorization denied: The expected name for the remote host (host@alias.example.org) does not match the authenticated name of the remote host (host@grid.example.org). This happens when the name in the host certificate does not match the information obtained from DNS and is often a DNS configuration problem.`

This mismatch can happen for a number of reasons: a site administrator has multiple hosts sharing a certificate, a host has multiple DNS aliases, and the client is not aware of which name the server is using for its certificate, or a host’s name has changed since the certificate was issued. The remedy for the client, after confirming with the GRAM administrator that the name after "authenticated name of the remote host" is the correct certificate name is to use a form of the GRAM resource name which includes this name. For example, explicitly adding a name to the abbreviated GRAM contact so that instead of `alias.example.org`, you would use `alias.example.org::host@grid.example.org`. 
1.2.3. Certificate Trust Issues

Because of the mutual authentication, both GRAM users and services can hit problems if they do not trust their peer's certificate or the Certificate Authority which issued it. If the client doesn't trust the server's certificate, it is easier to diagnose, because the GRAM service doesn't send much information back to the client if it doesn't trust it. However, working with the system administrator to get information from the GRAM logs will usually fix these problems fairly easily.

If the service's certificate is not trusted, the client will receive a message like this:

```
% globus-job-run grid.example.org /bin/hostname
GRAM Job submission failed because an authentication operation failed
OpenSSL Error: s3_clnt.c:915: in library: SSL routines, function SSL3_GET_SERVER_CERTIFICATE
globus_gsi_callback_module: Could not verify credential
globus_gsi_callback_module: Can't get the local trusted CA certificate: Untrusted self-signed certificate
```

This error indicates that certificate chain from the service certificate to the client contained a self-signed certificate (usually an indication that it's a CA certificate), which the client doesn't trust, and includes the hash of the certificate name (bbfccedf in this case). If you hit this particular type of error, you should send the information to the GRAM administrator and determine which CA should be trusted and what its signing policy is, to determine if you want to add it to your local set of trust roots.

**Note**

Different versions of OpenSSL produce different hashes for the same certificate names. If you upgrade a system (or transfer CA certificates between systems) to a different version of OpenSSL, you may hit this problem even if you think you have the CA certificate in your trusted certificate directory. If so, run the `globus-update-certificate-dir` program to update your hashes.

There are other reasons why a certificate might not be trusted (it's in a revoked list, it has expired or was issued in the future, etc). For more details look at the troubleshooting information in the GSI user's guide.

If for some reason the service does not trust your certificate, you'll get a rather cryptic message from GRAM that looks like this:

```
% globus-job-run grid.example.org /bin/hostname
GRAM Job submission failed because an authentication operation failed
globus_gsi_gssapi: Unable to verify remote side's credentials
globus_gsi_gssapi: Unable to verify remote side's credentials: Couldn't verify the remote
OpenSSL Error: s3_pkt.c:1086: in library: SSL routines, function SSL3_READ_BYTES: sslv3 alert bad certificate SSL alert number 42
```

To remedy this, consult the GRAM administrator to get information from the `/var/log/globus-gatekeeper.log` file to determine the reason why the gatekeeper didn't like your certificate. Again it could be CA trust issues, clock skew, or a revoked certificate. The error in the gatekeeper log would typically look like the client-side trust issue above.

1.2.4. Authentication with the Remote Server Failed

Once the GRAM service has authenticated the client, it maps the client's identity to a local user account using a grid-mapfile or other mapping service. If this fails, the client will receive a message that looks like this:

```
% globus-job-run grid.example.org /bin/hostname
GRAM Job submission failed because authentication with the remote server failed (error code ?)
```

95
To remedy this, consult the system administrator of the GRAM resource to be added to the authorized user's list. Be sure to send your credential subject name to make it easier for them. To get that information, run the command `grid-cert-info`.  

### 1.2.5. Unable to Find the Requested Service

Recall that a GRAM resource name includes a component called the service name. The default if not specified is `jobmanager`, but some sites may not use that name, or have a different LRM name than you expect. If you specify an incorrect service name, or the default is not present, you'll get an error that looks like this:  

```bash  
% grid-job-run grid.example.org /bin/hostname  
GRAM Job submission failed because the gatekeeper failed to find the requested service (error code 93)  
```

If you get this error, you'll need to determine which services are available on that GRAM resource, either by asking the admin or by looking at the entries in `/etc/grid-services`.

### 1.2.6. Failed to Run the Job Manager

The GRAM service is split between a privileged process called the `globus-gatekeeper` and a non-privileged process called the `globus-job-manager` which runs as a user process. If the `globus-gatekeeper` is unable to locate the `globus-job-manager` process, then this misconfiguration will show up like this:

```bash  
% grid-job-run grid.example.org /bin/hostname  
GRAM Job submission failed because the gatekeeper failed to run the job manager (error code 47)  
```

This is an installation mistake, and the administrator of the GRAM resource must fix this.

### 1.3. Jobs are Hanging

One problem GRAM users sometimes encounter is that it looks like jobs submitted to GRAM are not making any progress, even though the local resource manager thinks they've run. There are a couple of reasons why this might occur: GRAM is not getting the information it needs from the local resource manager or the GRAM client is not getting the information it needs. We'll cover diagnosing and handling the latter case in this document, as the other is an system administrator issue.  

The way `globus-job-run` and `globusrun` determine that jobs have completed is via GRAM job state callbacks. These are messages sent by the GRAM service to the client node indicating that something significant has happened in the lifecycle of the job. If for some reason the GRAM service can not get those messages to the client, the client will not be able to detect job state changes.

In order to determine if this is the case, submit a job using `globus-job-submit`, and then use the `globus-job-status` command to see if the job state changes. If it does not, then consult the GRAM administrator---there might be some problem with the installation. If it does, then for some reason the callbacks are not happening. This might be firewall issues or host naming issues.

The GRAM client sends a "callback contact" to the GRAM service when it submits a job, in order that it can receive notifications. This contact is a reference to a https server embedded in the GRAM client which only handles GRAM state callbacks. As with all web servers, it has a URL which defines how to contact it, which in this case consists of the client host name and the service port number. If the host name that is used is not resolvable (such as a for a laptop with a dynamic address), then the GRAM service will not be able to contact it. If that's the case, you can set the `GLOBUS_HOSTNAME` environment variable to the IP address that your client can be reached at, and then submit your jobs. This will cause GRAM to publish that address instead of what it thinks the client's host name is.

Another way that the GRAM service would be unable to send job state updates to a client would be if there's a firewall between the service and the client. If that's the case, you might need to set the `GLOBUS_TCP_PORT_RANGE`
environment variable to a comma-separated list of numbers which represent a range of minimum and maximum TCP port numbers to listen on. You might have to contact your site administrator to determine what TCP ports are allowed. If there are none, you can still use `globus-job-submit` and `globus-job-status` to track your job's state changes, or use another tool like those mentioned in the section about client tools.

### 1.4. Logs and Debugging

The GRAM service has a log file which contains information about the job as it is processed. These logs are located by default in `/var/log/globus/gram_USERNAME.log`. There are some different logging levels available, as described in the GRAM Administrator's Guide. These can be controlled on a per-job basis by adding the `loglevel` RSL attribute to your job description. The default is to log only `FATAL` and `ERROR` messages, but other levels can sometimes help understand what is going on.

### 1.5. Diagnosing LRM Errors

Sometimes, bugs creep into the LRM adapter scripts. When that occurs, the GRAM job will usually fail with an error like this:

```
GRAM Job failed because the job manager detected an invalid script status (error code 25)
```

If this occurs, you may have to work with a GRAM administrator to help debug this problem. One helpful thing you can do when reporting it is to save the GRAM internal script data so that it can be used outside of the GRAM service to see what the low-level error looks like. To do this, add the RSL fragment `(savejobdescription = yes)` to your job request. This will cause GRAM to leave a file called something like `$HOME/gram_[0-9]*.pl` in your home directory. You can use this with the internal tool `/usr/share/globus/globus-job-manager-script.pl` to try to submit the job to the LRM without using the GRAM service. The command line `/usr/share/globus/globus-job-manager-script.pl -m LRM -c submit -f GRAM-PL-FILE` will attempt to submit the job to the LRM. It will show all the information the LRM script sends to the GRAM service, which might include some perl-language error or badly formatted output from the script (which must only output lines which begin with `GRAM_SCRIPT_`.

In some extreme cases, the `savejobdescription` option will not generate a file. If that's the case, pass `/dev/null` as the argument to the `-f` command-line option. The problem is likely a perl syntax error which will be reached before the job description is loaded.

### 1.6. Email Support

If all else fails, please send information about your problem to `<gram-user@globus.org>`. You'll have to subscribe to a list before you can send an e-mail to it. See [here](http://dev.globus.org/wiki/Mailing_Lists) for general e-mail lists and information on how to subscribe to a list and [here](http://dev.globus.org/wiki/GRAM#Mailing_Lists) for GRAM specific lists. Depending on the problem, you may be requested to file a bug report to the globus project's [Issue Tracker](http://jira.globus.org).

## 2. Admin Troubleshooting

### 2.1. Security

GRAM requires a host certificate and private key in order for the `globus-gatekeeper` service to run. These are typically located in `/etc/grid-security/hostcert.pem` and `/etc/grid-security/hostkey.pem`, but

---

1. [http://dev.globus.org/wiki/Mailing_Lists](http://dev.globus.org/wiki/Mailing_Lists)
2. [http://dev.globus.org/wiki/GRAM#Mailing_Lists](http://dev.globus.org/wiki/GRAM#Mailing_Lists)
3. [http://jira.globus.org](http://jira.globus.org)
the path is configurable in the gatekeeper configuration file. The key must be protected by file permissions allowing
only the root user to read it.

GRAM also (by default) uses a grid-mapfile to authorize Grid users as local users. This file is typically located
in /etc/grid-security/grid-mapfile, but is configurable in the gatekeeper configuration file.

Problems in either of these configurations will show up in the gatekeeper log described below. See the GSI docu-
mentation for more detailed information about obtaining and installing host certificates and maintaining a grid-
mapfile.

2.2. Verify that Services are Running

GRAM relies on the globus-gatekeeper program and (in some cases) the globus-scheduler-event-generator pro-
grams to process jobs. If the former is not running, jobs requests will fail with a "connection refused" error. If the
latter is not running, GRAM jobs will appear to "hang" in the PENDING state.

The globus-gatekeeper is typically started via an init script installed in /etc/init.d/globus-gatekeep-
er. The command /etc/init.d/globus-gatekeeper status will indicate whether the service is running. See Section 2,
“Starting and Stopping GRAM5 services” for more information about starting and stopping the globus-gatekeeper
program.

If the globus-gatekeeper service fails to start, the output of the command globus-gatekeeper -test will output infor-
mation describing some types of configuration problems.

The globus-scheduler-event-generator is typically started via an init script installed in /etc/init.d/globus-
scheduler-event-generator. It is only needed when the LRM-specific "setup-seg" package is installed.
The command /etc/init.d/globus-scheduler-event-generator status will indicate whether the service is running.
See Section 2, “Starting and Stopping GRAM5 services” for more information about starting and stopping the
globus-scheduler-event-generator program.

2.3. Verify that LRM packages are installed

The globus-gatekeeper program starts the globus-job-manager service with different command-line parameters
depending on the LRM being used. Use the command globus-gatekeeper-admin -l to list which LRMs the gate-
keeper is configured to use.

The globus-job-manager-script.pl is the interface between the GRAM job manager process and the LRM adapter.
The command /usr/share/globus/globus-job-manager-script.pl -h will print the list of available adapters.

% /usr/share/globus/globus/globus-job-manager-script.pl -h
Usage: /usr/share/globus/globus/globus-job-manager-script.pl -m MANAGER -f FILE -c COMMAND
Installed managers: condor fork

The globus-scheduler-event-generator also uses an LRM-specific module to generate scheduler events for GRAM
to reduce the amount of resources GRAM uses on the machine where it runs. To determine which LRMs are in-
stalled and configured, use the command globus-scheduler-event-generator-admin -l.

% globus-scheduler-event-generator-admin -l
fork [DISABLED]

If any of these do not show the LRM you are trying to use, install the relevant packages related to that LRM and
restart the GRAM services. See the GRAM Administrator's Guide for more information about starting and stopping
the GRAM services.
2.4. Verify that the LRM packages are configured

All GRAM5 LRM adapters have a configuration file for site customizations, such as queue names, paths to executables needed to interface with the LRM, etc. Check that the values in these files are correct. These files are described in Section 4, “LRM Adapter Configuration”.

2.5. Check the Gatekeeper Log

The /var/log/globus-gatekeeper.log file contains information about service requests from clients, and will be useful when diagnosing service startup failures, authentication failures, and authorization failures.

2.5.1. Authorization failures

GRAM uses GSI to authenticate client job requests. If there is a problem with the GSI configuration for your host, or a client is trying to connect with a certificate signed by a CA your host does not trust, the job request will fail. This will show up in the log as a “GSS authentication failure”. See the GSI Administrator’s Guide for information about diagnosing authentication failures.

2.5.2. Gridmap failures

After authentication is complete, GRAM maps the Grid identity to a local user prior to starting the globus-job-manager process. If this fails, an error will show up in the log as "globus_gss_assist_gridmap() failed authorization". See the GSI Administrator’s Guide for information about managing gridmap files.

2.6. Job Manager Logs

A per-user job manager log is typically located in /var/log/globus/gram_$USERNAME.log. This log contains information from the job manager as it attempts to execute GRAM jobs via a local resource manager. The logs can be fairly verbose. Sometimes looking for log entries near those containing the string level=ERROR will show more information about what caused a particular failure.

Once you’ve found an error in the log, it is generally useful to find log entries related to the job which hit that error. There are two job IDs associated with each job, one a GRAM-specific ID, and one an LRM-specific ID. To determine the GRAM ID associated with a job, look for the attribute gramid in the log message. Finding that, looking for all other log messages which contain that gramid value will give a better picture of what the job manager is doing. To determine the LRM-specific ID, look for a message at TRACE level with the matching GRAM ID found above with the response value matching GRAM_SCRIPT_JOB_ID:LRM-ID. You can then find follow the state of the LRM-ID as well as the GRAM ID in the log, and correlate the LRM-ID information with local resource manager logs and administrative tools.

2.7. Email Support

If all else fails, please send information about your problem to <gram-user@globus.org>. You’ll have to subscribe to a list before you can send an e-mail to it. See here\(^4\) for general e-mail lists and information on how to subscribe to a list and here\(^5\) for GRAM-specific lists. Depending on the problem, you may be requested to file a bug report to the Globus project’s Issue Tracker\(^6\).

3. Errors

\(^4\) http://dev.globus.org/wiki/Mailing_Lists
\(^5\) http://dev.globus.org/wiki/GRAM#Mailing_Lists
\(^6\) http://jira.globus.org
### Table 9.1. GRAM5 Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Reason</th>
<th>Possible Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one of the RSL parameters is not supported</td>
<td>Check RSL documentation</td>
</tr>
<tr>
<td>2</td>
<td>the RSL length is greater than the maximum allowed</td>
<td>Use RSL substitutions to reduce length of RSL strings</td>
</tr>
<tr>
<td>3</td>
<td>an I/O operation failed</td>
<td>Enable trace logging and report to <a href="mailto:gram-dev@globus.org">gram-dev@globus.org</a></td>
</tr>
<tr>
<td>4</td>
<td>jobmanager unable to set default to the directory requested</td>
<td>Check that RSL directory attribute refers to a directory that exists on the target system.</td>
</tr>
<tr>
<td>5</td>
<td>the executable does not exist</td>
<td>Check that the RSL executable attribute refers to an executable that exists on the target system.</td>
</tr>
<tr>
<td>6</td>
<td>INSUFFICIENT_FUNDS</td>
<td>Unimplemented feature.</td>
</tr>
<tr>
<td>7</td>
<td>authentication with the remote server failed</td>
<td>Check that the contact string contains the proper X.509 DN.</td>
</tr>
<tr>
<td>8</td>
<td>the user cancelled the job</td>
<td>Don't cancel jobs you want to complete.</td>
</tr>
<tr>
<td>9</td>
<td>the system cancelled the job</td>
<td>Check RSL requirements such as maximum time and memory are valid for the job.</td>
</tr>
<tr>
<td>10</td>
<td>data transfer to the server failed</td>
<td>Check gatekeeper and/or job manager logs to see why the process failed.</td>
</tr>
<tr>
<td>11</td>
<td>the stdin file does not exist</td>
<td>Check that the RSL stdin attribute refers to a file that exists on the target system or has a valid ftp, gsiftp, http, or https URL.</td>
</tr>
<tr>
<td>12</td>
<td>the connection to the server failed (check host and port)</td>
<td>Check that the service is running on the expected TCP/IP port. Check that no firewall prevents contacting that TCP/IP port. Check $GLOBUS_LOCATION/var/globus-gatekeeper.log for runtime configuration errors.</td>
</tr>
<tr>
<td>13</td>
<td>the provided RSL 'maxtime' value is not an integer</td>
<td>Check that the RSL maxtime value evaluates to an integer.</td>
</tr>
<tr>
<td>14</td>
<td>the provided RSL 'count' value is not an integer</td>
<td>Check that the RSL count value evaluates to an integer.</td>
</tr>
<tr>
<td>15</td>
<td>the job manager received an invalid RSL</td>
<td>Check that the RSL string can be parsed by using <code>globusrun -p RSL</code>.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>--------------------</td>
</tr>
<tr>
<td>16</td>
<td>the job manager failed in allowing others to make contact</td>
<td>Check job manager log.</td>
</tr>
<tr>
<td>17</td>
<td>the job failed when the job manager attempted to run it</td>
<td>Verify that the LRM is configured properly.</td>
</tr>
<tr>
<td>18</td>
<td>an invalid paradyn was specified</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>19</td>
<td>the provided RSL 'jobtype' value is invalid</td>
<td>The RSL jobtype attribute is not indicated as supported by the LRM. Valid jobtype values are single, multiple, mpi, and condor.</td>
</tr>
<tr>
<td>20</td>
<td>the provided RSL 'myjob' value is invalid</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>21</td>
<td>the job manager failed to locate an internal script argument file</td>
<td>Check that $GLOBUS_LOCATION/libexec/globus-job-manager-script.pl exists and is executable. Check that the LRM-specific perl module is located in $GLOBUS_LOCATION/lib/perl/Globus/GRAM/JobManager directory and is valid. The command perl -I $GLOBUS_LOCATION/lib/perl $GLOBUS_LOCATION/lib/perl/Globus/GRAM/JobManager/LRM.pm can be used to check if there are any syntax errors in the script.</td>
</tr>
<tr>
<td>22</td>
<td>the job manager failed to create an internal script argument file</td>
<td>Check that your home directory is writable and not full.</td>
</tr>
<tr>
<td>23</td>
<td>the job manager detected an invalid job state</td>
<td>Check job manager logs.</td>
</tr>
<tr>
<td>24</td>
<td>the job manager detected an invalid script response</td>
<td>Check job manager logs. This is likely a bug in the LRM script.</td>
</tr>
<tr>
<td>25</td>
<td>the job manager detected an invalid script status</td>
<td>Check job manager logs. This is likely a bug in the LRM script.</td>
</tr>
<tr>
<td>26</td>
<td>the provided RSL 'jobtype' value is not supported by this job manager</td>
<td>Check that the RSL jobtype attribute is implemented by the LRM script. Note that some job types require configuration</td>
</tr>
<tr>
<td>27</td>
<td>unused ERROR_UNIMPLEMENTED</td>
<td>LRM does not support some feature included in the job request.</td>
</tr>
<tr>
<td>28</td>
<td>the job manager failed to create an internal script submission file</td>
<td>Check that the user's home file system is not full. Check job manager log</td>
</tr>
<tr>
<td>29</td>
<td>the job manager cannot find the user proxy</td>
<td>Check that client is delegating a proxy when authenticating with</td>
</tr>
</tbody>
</table>
## Troubleshooting

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Reason</th>
<th>Possible Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>the gatekeeper. Check that the user's home filesystem and the /tmp file system are not full.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>the job manager failed to open the user proxy</td>
<td>Check that the user's home filesystem and the /tmp file system are not full.</td>
</tr>
<tr>
<td>31</td>
<td>the job manager failed to cancel the job as requested</td>
<td>Check that the user's home filesystem and the /tmp file system are not full.</td>
</tr>
<tr>
<td>32</td>
<td>system memory allocation failed</td>
<td>Check job manager log for details.</td>
</tr>
<tr>
<td>33</td>
<td>the interprocess job communication initialization failed</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>34</td>
<td>the interprocess job communication setup failed</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>35</td>
<td>the provided RSL 'host count' value is invalid</td>
<td>Check that the RSL host_count attribute evaluates to an integer.</td>
</tr>
<tr>
<td>36</td>
<td>one of the provided RSL parameters is unsupported</td>
<td>Check job manager log for details about invalid parameter.</td>
</tr>
<tr>
<td>37</td>
<td>the provided RSL 'queue' parameter is invalid</td>
<td>Check that the RSL queue attribute evaluates to a string that corresponds to an LRM-specific queue name.</td>
</tr>
<tr>
<td>38</td>
<td>the provided RSL 'project' parameter is invalid</td>
<td>Check that the RSL project attribute evaluates to a string that corresponds to an LRM-specific project name.</td>
</tr>
<tr>
<td>39</td>
<td>the provided RSL string includes variables that could not be identified</td>
<td>Check that all RSL substitutions are defined before being used in the job description.</td>
</tr>
<tr>
<td>40</td>
<td>the provided RSL 'environment' parameter is invalid</td>
<td>Check that the RSL environment attribute contains a sequence of VARIABLE VALUE pairs.</td>
</tr>
<tr>
<td>41</td>
<td>the provided RSL 'dryrun' parameter is invalid</td>
<td>Remove the RSL dryrun attribute from the job description.</td>
</tr>
<tr>
<td>42</td>
<td>the provided RSL is invalid (an empty string)</td>
<td>Include a non-empty RSL string in your job submission request.</td>
</tr>
<tr>
<td>43</td>
<td>the job manager failed to stage the executable</td>
<td>Check that the file service hosting the executable is reachable from the GRAM5 service node. Check that the executable exists on the file service node. Check that there is sufficient disk space in the user's home directory on the service node to store the executable.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>--------------------</td>
</tr>
<tr>
<td>44</td>
<td>the job manager failed to stage the stdin file</td>
<td>Check that the file service hosting the standard input file is reachable from the GRAM5 service node. Check that the standard input file exists on the file service node. Check that there is sufficient disk space in the user's home directory on the service node to store the standard input file.</td>
</tr>
<tr>
<td>45</td>
<td>the requested job manager type is invalid</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>46</td>
<td>the provided RSL 'arguments' parameter is invalid</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>47</td>
<td>the gatekeeper failed to run the job manager</td>
<td>Check the gatekeeper or job manager logs for more information.</td>
</tr>
<tr>
<td>48</td>
<td>the provided RSL could not be properly parsed</td>
<td>Check that the RSL string can be parsed by using <code>globusrun -p RSL</code>.</td>
</tr>
<tr>
<td>49</td>
<td>there is a version mismatch between GRAM components</td>
<td>Ask system administrator to upgrade GRAM service to GRAM2 or GRAM5</td>
</tr>
<tr>
<td>50</td>
<td>the provided RSL 'arguments' parameter is invalid</td>
<td>Check that the RSL arguments attribute evaluates to a sequence of strings.</td>
</tr>
<tr>
<td>51</td>
<td>the provided RSL 'count' parameter is invalid</td>
<td>Check that the RSL count attribute evaluates to a positive integer value.</td>
</tr>
<tr>
<td>52</td>
<td>the provided RSL 'directory' parameter is invalid</td>
<td>Check that the RSL directory attribute evaluates to a string.</td>
</tr>
<tr>
<td>53</td>
<td>the provided RSL 'dryrun' parameter is invalid</td>
<td>Check that the RSL dryrun attribute evaluates to either yes or no.</td>
</tr>
<tr>
<td>54</td>
<td>the provided RSL 'environment' parameter is invalid</td>
<td>Check that the RSL environment attribute evaluates to a sequence of <code>VARIABLE, VALUE</code> pairs.</td>
</tr>
<tr>
<td>55</td>
<td>the provided RSL 'executable' parameter is invalid</td>
<td>Check that the RSL executable attribute evaluates to a string value.</td>
</tr>
<tr>
<td>56</td>
<td>the provided RSL 'host_count' parameter is invalid</td>
<td>Check that the RSL host_count attribute evaluates to a positive integer value.</td>
</tr>
<tr>
<td>57</td>
<td>the provided RSL 'jobtype' parameter is invalid</td>
<td>Check that the RSL jobtype attribute evaluates to one of <code>single, multiple, mpi, or condor</code>.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>--------------------</td>
</tr>
<tr>
<td>58</td>
<td>the provided RSL 'maxtime' parameter is invalid</td>
<td>Check that the RSL maxtime attribute evaluates to a positive integer value.</td>
</tr>
<tr>
<td>59</td>
<td>the provided RSL 'myjob' parameter is invalid</td>
<td>OBSOLETE IN GRAM5.</td>
</tr>
<tr>
<td>60</td>
<td>the provided RSL 'paradyn' parameter is invalid</td>
<td>OBSOLETE IN GRAM2.</td>
</tr>
<tr>
<td>61</td>
<td>the provided RSL 'project' parameter is invalid</td>
<td>Check that the RSL project attribute evaluates to a string value.</td>
</tr>
<tr>
<td>62</td>
<td>the provided RSL 'queue' parameter is invalid</td>
<td>Check that the RSL queue attribute evaluates to a string value.</td>
</tr>
<tr>
<td>63</td>
<td>the provided RSL 'stderr' parameter is invalid</td>
<td>Check that the RSL stderr attribute evaluates to a string value or a sequence of DESTINATION URLs with optional CACHE_TAG string parameters.</td>
</tr>
<tr>
<td>64</td>
<td>the provided RSL 'stdin' parameter is invalid</td>
<td>Check that the RSL stdin attribute evaluates to a string value.</td>
</tr>
<tr>
<td>65</td>
<td>the provided RSL 'stdout' parameter is invalid</td>
<td>Check that the RSL stdout attribute evaluates to a string value or a sequence of DESTINATION URLs with optional CACHE_TAG string parameters.</td>
</tr>
<tr>
<td>66</td>
<td>the job manager failed to locate an internal script</td>
<td>Check job manager log for more details.</td>
</tr>
<tr>
<td>67</td>
<td>the job manager failed on the system call pipe()</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>68</td>
<td>the job manager failed on the system call fcntl()</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>69</td>
<td>the job manager failed to create the temporary stdout filename</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>70</td>
<td>the job manager failed to create the temporary stderr filename</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>71</td>
<td>the job manager failed on the system call fork()</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>72</td>
<td>the executable file permissions do not allow execution</td>
<td>Check that the RSL executable attribute refers to an executable program or script.</td>
</tr>
<tr>
<td>73</td>
<td>the job manager failed to open stdout</td>
<td>Check that the RSL stdout attribute refers to one or more valid destination files or URLs.</td>
</tr>
<tr>
<td>74</td>
<td>the job manager failed to open stderr</td>
<td>Check that the RSL stderr attribute refers to one or more valid destination files or URLs.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>75</td>
<td>the cache file could not be opened in order to relocate the user proxy</td>
<td>Check that the user’s home directory is writable and not full on the GRAM5 service node.</td>
</tr>
<tr>
<td>76</td>
<td>cannot access cache files in <code>/globus/gass_cache</code>, check permissions, quota, and disk space</td>
<td>Check that the user’s home directory is writable and not full on the GRAM5 service node.</td>
</tr>
<tr>
<td>77</td>
<td>the job manager failed to insert the contact in the client contact list</td>
<td>Check job manager log</td>
</tr>
<tr>
<td>78</td>
<td>the contact was not found in the job manager’s client contact list</td>
<td>Don’t attempt to unregister callback contacts that are not registered.</td>
</tr>
<tr>
<td>79</td>
<td>connecting to the job manager failed. Possible reasons: job terminated, invalid job contact, network problems, ...</td>
<td>Check that the job manager process is running. Check that the job manager credential has not expired. Check that the job manager contact refers to the correct TCP/IP host and port. Check that the job manager contact is not blocked by a firewall.</td>
</tr>
<tr>
<td>80</td>
<td>the syntax of the job contact is invalid</td>
<td>Check the syntax of job contact string.</td>
</tr>
<tr>
<td>81</td>
<td>the executable parameter in the RSL is undefined</td>
<td>Include the RSL <code>executable</code> in all job requests.</td>
</tr>
<tr>
<td>82</td>
<td>the job manager service is misconfigured. <code>condor arch</code> undefined</td>
<td>Add the <code>-condor-arch</code> to the command-line or configuration file for a job manager configured to use the <code>condor</code> LRM.</td>
</tr>
<tr>
<td>83</td>
<td>the job manager service is misconfigured. <code>condor os</code> undefined</td>
<td>Add the <code>-condor-os</code> to the command-line or configuration file for a job manager configured to use the <code>condor</code> LRM.</td>
</tr>
<tr>
<td>84</td>
<td>the provided RSL ‘min_memory’ parameter is invalid</td>
<td>Check that the RSL <code>min_memory</code> attribute evaluates to a positive integer value.</td>
</tr>
<tr>
<td>85</td>
<td>the provided RSL ‘max_memory’ parameter is invalid</td>
<td>Check that the RSL <code>max_memory</code> attribute evaluates to a positive integer value.</td>
</tr>
<tr>
<td>86</td>
<td>the RSL ‘min_memory’ value is not zero or greater</td>
<td>Check that the RSL <code>min_memory</code> attribute evaluates to a positive integer value.</td>
</tr>
<tr>
<td>87</td>
<td>the RSL ‘max_memory’ value is not zero or greater</td>
<td>Check that the RSL <code>max_memory</code> attribute evaluates to a positive integer value.</td>
</tr>
<tr>
<td>88</td>
<td>the creation of a HTTP message failed</td>
<td>Check job manager log.</td>
</tr>
<tr>
<td>89</td>
<td>parsing incoming HTTP message failed</td>
<td>Check job manager log.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>90</td>
<td>the packing of information into a HTTP message failed</td>
<td>Check job manager log.</td>
</tr>
<tr>
<td>91</td>
<td>an incoming HTTP message did not contain the expected information</td>
<td>Check job manager log.</td>
</tr>
<tr>
<td>92</td>
<td>the job manager does not support the service that the client requested</td>
<td>Check that the client is talking to the correct service</td>
</tr>
<tr>
<td>93</td>
<td>the gatekeeper failed to find the requested service</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>94</td>
<td>the jobmanager does not accept any new requests (shutting down)</td>
<td>Execute queries before the job has been cleaned up.</td>
</tr>
<tr>
<td>95</td>
<td>the client failed to close the listener associated with the callback URL</td>
<td>Call <code>globus_gram_client_callback_disallow()</code> with a valid the callback contact.</td>
</tr>
<tr>
<td>96</td>
<td>the gatekeeper contact cannot be parsed</td>
<td>Check the syntax of the gatekeeper contact string you are attempting to contact.</td>
</tr>
<tr>
<td>97</td>
<td>the job manager could not find the 'poe' command</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>98</td>
<td>the job manager could not find the 'mpirun' command</td>
<td>Configure the LRM script with <code>mpirun</code> in your path.</td>
</tr>
<tr>
<td>99</td>
<td>the provided RSL 'start_time' parameter is invalid</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>100</td>
<td>the provided RSL 'reservation_handle' parameter is invalid</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>101</td>
<td>the provided RSL 'max_wall_time' parameter is invalid</td>
<td>Check that the RSL <code>max_wall_time</code> attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>102</td>
<td>the RSL 'max_wall_time' value is not zero or greater</td>
<td>Check that the RSL <code>max_wall_time</code> attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>103</td>
<td>the provided RSL 'max_cpu_time' parameter is invalid</td>
<td>Check that the RSL <code>max_cpu_time</code> attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>104</td>
<td>the RSL 'max_cpu_time' value is not zero or greater</td>
<td>Check that the RSL <code>max_cpu_time</code> attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>105</td>
<td>the job manager is misconfigured, a scheduler script is missing</td>
<td>Check that the administrator has configured the LRM by running its setup script.</td>
</tr>
<tr>
<td>106</td>
<td>the job manager is misconfigured, a scheduler script has invalid permis-sions</td>
<td>Check that the administrator has installed the <code>GLLOBUS_LOCATION/libexec/globus-job-ma-</code></td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>-------------------</td>
</tr>
<tr>
<td>107</td>
<td>er-script.pl script. Check that the file system containing that script allows file execution.</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>the job manager did not recognize/support the signal type</td>
<td>Check that your signal operation is using the correct signal constant.</td>
</tr>
<tr>
<td>109</td>
<td>the job manager failed to get the job id from the local scheduler</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>110</td>
<td>the job manager is waiting for a commit signal</td>
<td>Send a two-phase commit signal to the job manager to acknowledge receiving the job contact from the job manager.</td>
</tr>
<tr>
<td>111</td>
<td>the job manager timed out while waiting for a commit signal</td>
<td>Send a two-phase commit signal to the job manager to acknowledge receiving the job contact from the job manager. Increase the two-phase commit time out for your job. Check that the job manager contact TCP/IP port is reachable from your client.</td>
</tr>
<tr>
<td>112</td>
<td>the provided RSL 'save_state' parameter is invalid</td>
<td>Check that the RSL save_state attribute is set to yes or no.</td>
</tr>
<tr>
<td>113</td>
<td>the provided RSL 'restart' parameter is invalid</td>
<td>Check that the RSL restart attribute evaluates to a string containing a job contact string.</td>
</tr>
<tr>
<td>114</td>
<td>the provided RSL 'two_phase' parameter is invalid</td>
<td>Check that the RSL two_phase attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>115</td>
<td>the RSL 'two_phase' value is not zero or greater</td>
<td>Check that the RSL two_phase attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>116</td>
<td>the provided RSL 'stdout_position' parameter is invalid</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>117</td>
<td>the RSL 'stdout_position' value is not zero or greater</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>118</td>
<td>the provided RSL 'stderr_position' parameter is invalid</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>119</td>
<td>the RSL 'stderr_position' value is not zero or greater</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>120</td>
<td>the job manager restart attempt failed</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
</tbody>
</table>
## Error Code

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Reason</th>
<th>Possible Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>the job state file doesn't exist</td>
<td>Check that the job contact you are trying to restart matches one that the job manager returned to you.</td>
</tr>
<tr>
<td>122</td>
<td>could not read the job state file</td>
<td>Check that the state file directory is not full.</td>
</tr>
<tr>
<td>123</td>
<td>could not write the job state file</td>
<td>Check that the state file directory is not full.</td>
</tr>
<tr>
<td>124</td>
<td>old job manager is still alive</td>
<td>Contact the returned job manager contact to manage the job you are trying to restart.</td>
</tr>
<tr>
<td>125</td>
<td>job manager state file TTL expired</td>
<td>OBSOLETE in GRAM2</td>
</tr>
<tr>
<td>126</td>
<td>it is unknown if the job was submitted</td>
<td>Check job manager log.</td>
</tr>
<tr>
<td>127</td>
<td>the provided RSL <code>remote_io_url</code> parameter is invalid</td>
<td>Check that the RSL <code>remote_io_url</code> attribute evaluates to a string value.</td>
</tr>
<tr>
<td>128</td>
<td>could not write the remote io url file</td>
<td>Check that the user's home file system on the job manager service node is writable and not full.</td>
</tr>
<tr>
<td>129</td>
<td>the standard output/error size is different</td>
<td>Send a stdio update signal to redirect the job manager output to a new URL</td>
</tr>
<tr>
<td>130</td>
<td>the job manager was sent a stop signal (job is still running)</td>
<td>Submit a restart request to monitor the job.</td>
</tr>
<tr>
<td>131</td>
<td>the user proxy expired (job is still running)</td>
<td>Generate a new proxy and then submit a restart request to monitor the job.</td>
</tr>
<tr>
<td>132</td>
<td>the job was not submitted by original job manager</td>
<td>OBSOLETE IN GRAM2</td>
</tr>
<tr>
<td>133</td>
<td>the job manager is not waiting for that commit signal</td>
<td>Do not send a commit signal to a job that is not waiting for a commit signal.</td>
</tr>
<tr>
<td>134</td>
<td>the provided RSL scheduler specific parameter is invalid</td>
<td>Check the LRM-specific documentation to determine what values are legal for the RSL extensions implemented by the LRM.</td>
</tr>
<tr>
<td>135</td>
<td>the job manager could not stage in a file</td>
<td>Check that the file service hosting the file to stage is reachable from the GRAM5 service node. Check that the file to stage exists on the file service node. Check that there is sufficient disk space in the user's home directory on the service node to store the file to stage.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>-------------------</td>
</tr>
<tr>
<td>136</td>
<td>the scratch directory could not be created</td>
<td>Check that the directory named by the RSL <code>scratch_dir</code> attribute exists and is writable. Check that the directory named by the RSL <code>scratch_dir</code> attribute is not full.</td>
</tr>
<tr>
<td>137</td>
<td>the provided 'gass_cache' parameter is invalid</td>
<td>Check that the RSL <code>gass_cache</code> attribute evaluates to a string.</td>
</tr>
<tr>
<td>138</td>
<td>the RSL contains attributes which are not valid for job submission</td>
<td>Do not use restart- or signal-only RSL attributes when submitting a job.</td>
</tr>
<tr>
<td>139</td>
<td>the RSL contains attributes which are not valid for stdio update</td>
<td>Do not use submit- or restart-only RSL attributes when sending a stdio update signal to a job.</td>
</tr>
<tr>
<td>140</td>
<td>the RSL contains attributes which are not valid for job restart</td>
<td>Do not use submit- or signal-only RSL attributes when restarting a job.</td>
</tr>
<tr>
<td>141</td>
<td>the provided RSL 'file_stage_in' parameter is invalid</td>
<td>Check that the RSL <code>file_stage_in</code> attribute evaluates to a sequence of <code>SOURCE DESTINATION</code> pairs.</td>
</tr>
<tr>
<td>142</td>
<td>the provided RSL 'file_stage_in_shared' parameter is invalid</td>
<td>Check that the RSL <code>file_stage_in_shared</code> attribute evaluates to a sequence of <code>SOURCE DESTINATION</code> pairs.</td>
</tr>
<tr>
<td>143</td>
<td>the provided RSL 'file_stage_out' parameter is invalid</td>
<td>Check that the RSL <code>file_stage_out</code> attribute evaluates to a sequence of <code>SOURCE DESTINATION</code> pairs.</td>
</tr>
<tr>
<td>144</td>
<td>the provided RSL 'gass_cache' parameter is invalid</td>
<td>Check that the RSL <code>gass_cache</code> attribute evaluates to a string.</td>
</tr>
<tr>
<td>145</td>
<td>the provided RSL 'file_cleanup' parameter is invalid</td>
<td>Check that the RSL <code>file_cleanup</code> attribute evaluates to a sequence of strings.</td>
</tr>
<tr>
<td>146</td>
<td>the provided RSL 'scratch_dir' parameter is invalid</td>
<td>Check that the RSL <code>scratch_dir</code> attribute evaluates to a string.</td>
</tr>
<tr>
<td>147</td>
<td>the provided scheduler-specific RSL parameter is invalid</td>
<td>Check the LRM-specific documentation to determine what values are legal for the RSL extensions implemented by the LRM.</td>
</tr>
<tr>
<td>148</td>
<td>a required RSL attribute was not defined in the RSL spec</td>
<td>Check that the RSL <code>executable</code> attribute is present in your job request RSL. Check that</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>--------------------</td>
</tr>
<tr>
<td>149</td>
<td>the gass_cache attribute points to an invalid cache directory</td>
<td>Check that the RSL gass_cache attributes evaluates to a directory that exists or can be created. Check that the user's home file system is writable and not full.</td>
</tr>
<tr>
<td>150</td>
<td>the provided RSL 'save_state' parameter has an invalid value</td>
<td>Check that the RSL save_state attribute has a value of yes or no.</td>
</tr>
<tr>
<td>151</td>
<td>the job manager could not open the RSL attribute validation file</td>
<td>Check that the $GLOBUS_LOCATION/share/globus_gram_job_manager/globus-gram-job-manager.rvf is present and readable on the job manager service node. Check that the $GLOBUS_LOCATION/share/globus_gram_job_manager/LRM.rvf is readable on the job manager service node if present.</td>
</tr>
<tr>
<td>152</td>
<td>the job manager could not read the RSL attribute validation file</td>
<td>Check that the $GLOBUS_LOCATION/share/globus_gram_job_manager/globus-gram-job-manager.rvf is valid. Check that the $GLOBUS_LOCATION/share/globus_gram_job_manager/LRM.rvf is valid if present.</td>
</tr>
<tr>
<td>153</td>
<td>the provided RSL 'proxy_timeout' is invalid</td>
<td>Check that RSL proxy_timeout attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>154</td>
<td>the RSL 'proxy_timeout' value is not greater than zero</td>
<td>Check that RSL proxy_timeout attribute evaluates to a positive integer.</td>
</tr>
<tr>
<td>155</td>
<td>the job manager could not stage out a file</td>
<td>Check that the source file being staged exists on the job manager service node. Check that the directory of the destination file being staged exists on the file service node. Check that the directory of the destination file being staged is writable by the user. Check that the destination file service is reachable by the job manager service node.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Reason</td>
<td>Possible Solutions</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>156</td>
<td>the job contact string does not match any which the job manager is handling</td>
<td>Check that the job contact string matches one returned from a job request.</td>
</tr>
<tr>
<td>157</td>
<td>proxy delegation failed</td>
<td>Check that the job manager service node trusts the signer of your credential. Check that you trust the signer of the job manager service node's credential.</td>
</tr>
<tr>
<td>158</td>
<td>the job manager could not lock the state lock file</td>
<td>Check that the file system holding the job state directory supports POSIX advisory locking. Check that the job state directory is writable by the user on the service node. Check that the job state directory is not full.</td>
</tr>
<tr>
<td>159</td>
<td>an invalid globus_io_clientattr_t was used.</td>
<td>Check that you have initialized the globus_io_clientattr_t attribute prior to using it with the GRAM client API.</td>
</tr>
<tr>
<td>160</td>
<td>an null parameter was passed to the gram library</td>
<td>Check that you are passing legal values to all GRAM API calls.</td>
</tr>
<tr>
<td>161</td>
<td>the job manager is still streaming output</td>
<td>OBSOLETE IN GRAM5</td>
</tr>
<tr>
<td>162</td>
<td>the authorization system denied the request</td>
<td>Check with your GRAM system administrator to allow a particular certificate to be authorized.</td>
</tr>
<tr>
<td>163</td>
<td>the authorization system reported a failure</td>
<td>Check with your system administrator to verify that the authorization system is configured properly.</td>
</tr>
<tr>
<td>164</td>
<td>the authorization system denied the request - invalid job id</td>
<td>Check with your system administrator to verify that the authorization system is configured properly. Use a credential which is authorized to interact with a particular GRAM job.</td>
</tr>
<tr>
<td>165</td>
<td>the authorization system denied the request - not authorized to run the specified executable</td>
<td>Check with your system administrator to verify that the authorization system is configured properly. Use a credential which is authorized to interact with a particular GRAM job.</td>
</tr>
<tr>
<td>166</td>
<td>the provided RSL 'user_name' parameter is invalid.</td>
<td>Check that the RSL user_name attribute evaluates to a string.</td>
</tr>
<tr>
<td>167</td>
<td>the job is not running in the account named by the 'user_name' parameter.</td>
<td>Ask with the GRAM system administrator to add an authorization entry to allow your credential to</td>
</tr>
</tbody>
</table>
## Troubleshooting

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Reason</th>
<th>Possible Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>run jobs as the specified user account.</td>
</tr>
</tbody>
</table>
Chapter 10. Semantics and syntax of protocols

1. GRAM5 Protocol

The GRAM Protocol is used to handle communication between the Gatekeeper, Job Manager, and GRAM Clients. The protocol is based on a subset of the HTTP/1.1 protocol, with a small set of message types and responses sent as the body of the HTTP requests and responses. This document describes GRAM Protocol version 2 as used by GRAM5. This is compatible with with the GRAM Protocol parsers in GRAM2 with extensions.

1.1. Framing

GRAM messages are framed in HTTP/1.1 messages. However, only a small subset of the HTTP specification is used or understood by the GRAM system. All GRAM requests are HTTP POST messages. Only the following HTTP headers are understood:

- Host
- Content-Type (set to "application/x-globus-gram" in all cases)
- Content-Length
- Connection (set to "close" in all HTTP responses)

Only the following status codes are supported in response's HTTP Status-Line:

- 200 OK
- 403 Forbidden
- 404 Not Found
- 500 Internal Server Error
- 400 Bad Request

1.2. Message Format

All messages use the carriage return (ASCII value 13) followed by line feed (ASCII value 10) sequence to delimit lines. In all cases, a blank line separates the HTTP header from the message body. All application/x-globus-gram message bodies consist of attribute names followed by a colon, a space, and then the value of the attribute. When the value may contain a newline or double-quote character, a special escaping rule is used to encapsulate the complete string. This encapsulation consists of surrounding the string with double-quotes, and escaping all double-quote and backslash characters within the string with a backslash. All other characters are sent without modification. For example, the string

\[
\text{rsl: \\& ( executable = \\"/bin/echo" )}\\
( \text{ arguments = "hello" )}
\]

becomes
In GRAM5, protocol extensions are supported in the status update messages. These extensions are implemented as extra attribute names after all of the attributes defined in the messages below. Older GRAM protocol parsers will ignore those extensions that occur after the attributes in the messages defined below. In GRAM5, the following extensions are used:

- **exit-code**: Job exit code. Sent in job state callbacks and in job status replies when the job completes.
- **gt3-failure-type**: Failure detail type for staging errors. Sent in job state callbacks and in job status replies when a job fails.
- **gt3-failure-message**: Failure detail message for more context for errors. Sent in job state callbacks and in job status replies when a job fails.
- **gt3-failure-source**: Failure detail message for the source of a failed file transfer. Sent in job state callbacks and in job status replies when a job fails.
- **gt3-failure-destination**: Failure detail message for the destination of a failed file transfer. Sent in job state callbacks and in job status replies when a job fails.
- **version**: Job manager package version. Sent in all messages from the job manager.
- **toolkit-version**: Toolkit release that the job manager is running. Sent in all messages from the job manager.

This is the only form of quoting which application/x-globus-gram messages support. Use of % HEX HEX escapes (such as seen in URL encodings) is not meaningful for this protocol.

### 1.3. Message Types

#### 1.3.1. Ping Request

A ping request is used to verify that the gatekeeper is configured properly to handle a named service. The ping request consists of the following:

```
POST ping/job-manager-name HTTP/1.1
Host: host-name
Content-Type: application/x-globus-gram
Content-Length: message-size

protocol-version: version
```

The values of the message-specific strings are:

- **job-manager-name**: The name of the service to have the gatekeeper check. The service name corresponds to one of the gatekeeper's configured grid-services, and is usually of the form "jobmanager-LRM".
- **host-name**: The name of the host on which the gatekeeper is running. This exists only for compatibility with the HTTP/1.1 protocol.
- **message-size**: The length of the content of the message, not including the HTTP/1.1 header.
version

The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string "2".

1.3.2. Job Request

A job request is used to scheduler a job remotely using GRAM. The ping request consists of the HTTP framing described above with the request-URI consisting of job-manager-name, where job-manager-name is the name of the service to use to schedule the job. The format of a job request message consists of the following:

POST job-manager-name[@user-name] HTTP/1.1
Host: host-name
Content-Type: application/x-globus-gram
Content-Length: message-size

protocol-version: version
job-state-mask: mask
callback-url: callback-contact
rsl: rsl-description

The values of the emphasized text items are as below:

job-manager-name The name of the service to submit the job request to. The service name corresponds to one of the gatekeeper's configured grid-services, and is usually of the form jobmanager-LRM.

user-name Starting with GT4.0, a client may request that a certain account by used by the gatekeeper to start the job manager. This is done optionally by appending the @ symbol and the local user name that the job should be run as to the job-manager-name. If the @ and user-name are not present, then the first grid map entry will be used. If the client credential is not authorized in the grid map to use the specified account, an authorization error will occur in the gatekeeper.

host-name The name of the host on which the gatekeeper is running. This exists only for compatibility with the HTTP/1.1 protocol.

message-size The length of the content of the message, not including the HTTP/1.1 header.

version The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string 2.

mask An integer representation of the job state mask. This value is obtained from a bitwise-OR of the job state values which the client wishes to receive job status callbacks about. These meanings of the various job state values are defined in the GRAM Protocol API documentation.

callback-contact A https URL which defines a GRAM protocol listener which will receive job state updates. The from a bitwise-OR of the job state values which the client wishes to receive job status callbacks about. The job status update messages are defined below.

rsl-description A quoted string containing the RSL description of the job request.

1.3.3. Status Request

A status request is used by a GRAM client to get the current job state of a running job. This type of message can only be sent to a job manager's job-contact (as returned in the reply to a job request message). The format of a job request message consists of the following:
POST `job-contact` HTTP/1.1
Host: `host-name`
Content-Type: application/x-globus-gram
Content-Length: `message-size`
protocol-version: `version`

"status"

The values of the emphasized text items are as below:

`job-contact` The job contact string returned in a response to a job request message, or determined by querying the MDS system.

`host-name` The name of the host on which the job manager is running. This exists only for compatibility with the HTTP/1.1 protocol.

`message-size` The length of the content of the message, not including the HTTP/1.1 header.

`version` The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string 2.

### 1.3.4. Callback Register Request

A callback register request is used by a GRAM client to register a new callback contact to receive GRAM job state updates. This type of message can only be sent to a job manager's `job-contact` (as returned in the reply to a job request message). The format of a job request message consists of the following:

POST `job-contact` HTTP/1.1
Host: `host-name`
Content-Type: application/x-globus-gram
Content-Length: `message-size`

```
protocol-version: version
"register mask callback-contact"
```

The values of the emphasized text items are as below:

`job-contact` The job contact string returned in a response to a job request message, or determined by querying the MDS system.

`host-name` The name of the host on which the job manager is running. This exists only for compatibility with the HTTP/1.1 protocol.

`message-size` The length of the content of the message, not including the HTTP/1.1 header.

`version` The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string 2.

`mask` An integer representation of the job state mask. This value is obtained from a bitwise-OR of the job state values which the client wishes to receive job status callbacks about. These meanings of the various job state values are defined in the GRAM Protocol API documentation.

`callback-contact` A https URL which defines a GRAM protocol listener which will receive job state updates. The from a bitwise-OR of the job state values which the client wishes to receive job status callbacks about. The job status update messages are defined below.
1.3.5. Callback Unregister Request

A callback unregister request is used by a GRAM client to request that the job manager no longer send job state updates to the specified callback contact. This type of message can only be sent to a job manager's job-contact (as returned in the reply to a job request message). The format of a job request message consists of the following:

POST job-contact HTTP/1.1
Host: host-name
Content-Type: application/x-globus-gram
Content-Length: message-size

protocol-version: version
"unregister callback-contact"

The values of the emphasized text items are as below:

job-contact The job contact string returned in a response to a job request message, or determined by querying the MDS system.

host-name The name of the host on which the job manager is running. This exists only for compatibility with the HTTP/1.1 protocol.

message-size The length of the content of the message, not including the HTTP/1.1 header.

version The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string "2".

callback-contact A https URL which defines a GRAM protocol listener which should no longer receive job state updates. The from a bitwise-OR of the job state values which the client wishes to receive job status callbacks about. The job status update messages are defined @ref globus_gram_protocol_job_state_updates "below".

1.3.6. Job Cancel Request

A job cancel request is used by a GRAM client to request that the job manager terminate a job. This type of message can only be sent to a job manager's job-contact (as returned in the reply to a job request message). The format of a job request message consists of the following:

POST job-contact HTTP/1.1
Host: host-name
Content-Type: application/x-globus-gram
Content-Length: message-size

protocol-version: version
"cancel"

The values of the emphasized text items are as below:

job-contact The job contact string returned in a response to a job request message, or determined by querying the MDS system.

host-name The name of the host on which the job manager is running. This exists only for compatibility with the HTTP/1.1 protocol.
message-size The length of the content of the message, not including the HTTP/1.1 header.

version The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string 2.

1.3.7. Job Signal Request

A job signal request is used by a GRAM client to request that the job manager process a signal for a job. The arguments to the various signals are discussed in the protocol library documentation. The format of a job request message consists of the following:

POST job-contact HTTP/1.1
Host: host-name
Content-Type: application/x-globus-gram
Content-Length: message-size

protocol-version: version
"signal"

The values of the emphasized text items are as below:

job-contact The job contact string returned in a response to a job request message, or determined by querying the MDS system.

host-name The name of the host on which the job manager is running. This exists only for compatibility with the HTTP/1.1 protocol.

message-size The length of the content of the message, not including the HTTP/1.1 header.

version The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string 2.

signal A quoted string containing the signal number and its parameters.

1.3.8. Job State Updates

A job status update message is sent by the job manager to all registered callback contacts when the job’s status changes. The format of the job status update messages is as follows:

POST callback-contact HTTP/1.1
Host: host-name
Content-Type: application/x-globus-gram
Content-Length: message-size

protocol-version: version
job-manager-url: job-contact
status: status-code
failure-code: failure-code

The values of the emphasized text items are as below:

callback-contact The callback contact string registered with the job manager either by being passed as the callback-contact in a job request message or in a callback register message.

host-name The host part of the callback-contact URL. This exists only for compatibility with the HTTP/1.1 protocol.
**Semantics and syntax of protocols**

`message-size` The length of the content of the message, not including the HTTP/1.1 header.

`version` The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string 2.

`job-contact` The job contact of the job which has changed states.

### 1.3.9. Proxy Delegation

A proxy delegation message is sent by the client to the job manager to initiate a delegation handshake to generate a new proxy credential for the job manager. This credential is used by the job manager or the job when making further secured connections. The format of the delegation message is as follows:

**POST callback-contact HTTP/1.1**

**Host: host-name**

**Content-Type: application/x-globus-gram**

**Content-Length: message-size**

**protocol-version: version**

"renew"

If a successful (200) reply is sent in response to this message, then the client will proceed with a GSI delegation handshake. The tokens in this handshake will be framed with a 4 byte big-endian token length header. The framed tokens will then be wrapped using the GLOBUS_IO_SECURE_CHANNEL_MODE_SSL_WRAP wrapping mode. The job manager will frame response tokens in the same manner. After the job manager receives its final delegation token, it will respond with another response message that indicates whether the delegation was processed or not. This response message is a standard GRAM response message.

### 1.3.10. Security Attributes

The following security attributes are needed to communicate with the Gatekeeper:

- Authentication must be done using GSSAPI mutual authentication
- Messages must be wrapped with support for the delegation message. When using Globus I/O, this is accomplished by using the the GLOBUS_IO_SECURE_CHANNEL_MODE_GSI_WRAP wrapping mode.

### 1.4. Job State Model

As the GRAM service processes a job, the job undergoes a series of state transitions. These states and their meanings follow:

#### Table 10.1. GRAM Job States

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_UNSUBMITTED</td>
<td>Initial job state</td>
</tr>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_STAGE_IN</td>
<td>Job staging in progress</td>
</tr>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_PENDING</td>
<td>Job submitted to LRM, awaiting execution</td>
</tr>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_ACTIVE</td>
<td>Job executing</td>
</tr>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_SUSPENDED</td>
<td>Job made progress executing but is now suspended</td>
</tr>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_STAGE_OUT</td>
<td>Job staging in progress after job completed</td>
</tr>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE</td>
<td>Job completed successfully</td>
</tr>
</tbody>
</table>
Semantics and syntax of protocols

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED</td>
<td>Job was canceled or failed</td>
</tr>
</tbody>
</table>

**Figure 10.1. GRAM State Transitions**

[Diagram of GRAM state transitions showing various states such as Unsubmitted, Pending, Active, Failed, and Done, with transitions between states such as Submit to LRM, Staging Begins, Execution Complete, etc.]
Chapter 11. Related Documentation

No related documentation links have been determined at this time.
Chapter 12. Internal Components

Internal Components

1 internal-components.html
Glossary

C

certificate A public key plus information about the certificate owner bound together by the
digital signature of a CA. In the case of a CA certificate, the certificate is self
signed, i.e. it was signed using its own private key.

G

Gatekeeper A part of GRAM that runs as root and authenticates clients prior to starting the
Job Manager.

grid map file A file containing entries mapping certificate subjects to local user names. This
file can also serve as a access control list for GSI enabled services and is typi-
cally found in /etc/grid-security/grid-mapfile. For more informa-
tion see the Gridmap section here.

J

Job Manager A part of GRAM that runs as a local user and interfaces with a Local Resource
Manager for that user.

L

Local Resource Manager (LRM) A system which controls access to a compute resource, such as a compute clus-
ter or parallel computer. Such systems provide batch execution interfaces, which
GRAM uses to execute jobs. Condor, Portable Batch System, GridEngine are
examples of local resource managers.
See Also Condor, Portable Batch System, Oracle GridEngine.

LRM Adapter The interface code between a Local Resource Manager and GRAM.
In most cases, this consists of a Perl module that implements the
Globus::GRAM::JobManager class and a Scheduler Event Generator
module.
See Also Local Resource Manager.

P

proxy certificate A short lived certificate issued using a EEC. A proxy certificate typically has the
same effective subject as the EEC that issued it and can thus be used in its place.
GSI uses proxy certificates for single sign on and delegation of rights to other
entities.

For more information about types of proxy certificates and their compatibility
in different versions of GT, see http://dev.globus.org/wiki/Security/ProxyCert-
Types.
<table>
<thead>
<tr>
<th><strong>R</strong></th>
<th>Resource Specification Language (RSL)</th>
<th>Term used to describe a GRAM job for GT2 and GT3. (Note: This is not the same as RLS - the Replica Location Service)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td>Scheduler Event Generator (SEG)</td>
<td>The Scheduler Event Generator (SEG) is a program which uses scheduler-specific monitoring modules to generate job state change events. Depending on scheduler-specific requirements, the SEG may need to run with privileges to enable it to obtain scheduler event notifications. As such, one SEG runs per scheduler resource. For example, on a host which provides access to both PBS and fork jobs, two SEGs, running at (potentially) different privilege levels will be running. One SEG instance exists for any particular scheduled resource instance (one for all homogeneous PBS queues, one for all fork jobs, etc). The SEG is implemented in an executable called the globus-scheduler-event-generator, located in the Globus Toolkit's libexec directory.</td>
</tr>
</tbody>
</table>
Index

A
apis, 63
  overview, 63

C
compatibility, 4

D
debugging, 92
dependencies, 4

E
errors, 99

F
features, 3

P
platforms, tested, 3

T
troubleshooting, 94, 94
  errors, 94, 94