

# **GT 5.0.2 GRAM5: Developer's Guide**

---

## **GT 5.0.2 GRAM5: Developer's Guide**

### **Introduction**

This guide is intended to help a developer create GRAM5 clients in C. It provides an overview of the concepts and APIs needed to interact with GRAM services.

---

# Table of Contents

1. Before you begin .....	1
1. Feature summary .....	1
2. Tested platforms .....	1
3. Backward compatibility summary .....	1
4. Technology dependencies .....	2
5. Security Considerations .....	2
2. GRAM5 Concepts for Developers .....	3
1. Blocking and Nonblocking Function Variants .....	3
2. Service Contact Strings .....	3
3. Job State Callbacks and Polling .....	3
4. Credential Management .....	4
5. RSL .....	4
3. Basic GRAM Client Scenarios .....	5
1. "Ping" a Job Manager .....	5
2. Check a Job Manager Version .....	6
3. Submitting a Job .....	8
4. Submitting a Job and Processing Job State Callbacks .....	10
5. Polling Job Status .....	13
6. Canceling a Job .....	15
7. Refreshing Job Credential .....	16
4. Advanced GRAM Client Scenarios .....	18
1. Non-blocking Job Submission .....	18
2. Custom Security Attributes .....	21
3. Modifying RSL .....	24
5. Tutorials .....	28
6. APIs .....	29
1. Programming Model Overview .....	29
7. RSL Specification v1.1 .....	30
1. RSL Syntax Overview .....	30
2. RSL Tokenization Overview .....	31
3. RSL Substitution Semantics .....	32
4. RSL Attribute Summary .....	32
5. Simple RSL Examples .....	35
6. RSL grammar and tokenization rules .....	36
8. Debugging .....	38
1. Basic Debugging Methods .....	38
2. Advanced Debugging Methods .....	39
9. Troubleshooting .....	40
1. Troubleshooting tips .....	40
2. Errors .....	41
10. Semantics and syntax of protocols .....	51
1. GRAM5 Protocol .....	51
11. Related Documentation .....	59
12. Internal Components .....	60
Glossary .....	61
Index .....	62

---

## List of Figures

10.1. GRAM State Transitions .....	58
------------------------------------	----

---

## List of Tables

2.1. GRAM Contact String Types .....	3
6.1. GRAM Client APIs .....	29
9.1. GRAM5 Errors .....	42
10.1. GRAM Job States .....	58

---

## List of Examples

7.1. Quoted Literal Examples .....	32
7.2. GRAM5 Job Request Examples .....	36

---

# Chapter 1. Before you begin

## 1. Feature summary

New Features new since 5.0.2

- A new tool **globus-gram-streamer** implements stdio streaming similar to gram2, for use with the grid-monitor.sh script from Condor. As a result, a Condor-G client which does not know about GRAM5 features will be able to submit many jobs to a GRAM5 server.

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 5.3 x86\_64
  - Debain 4.0 x86\_64
- Mac OS X
  - Mac OS X 10.5.8

## 3. Backward compatibility summary

Protocol changes in GRAM since GT5.0.2 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 coallocation 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.

## 4. Technology dependencies

GRAM depends on the following GT components:

- Globus Common
- GSI C
- GridFTP server

## 5. Security Considerations

No special security considerations exist at this time.

---

# 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**

Type	Meaning
Gatekeeper Service Contact	This string describes how to contact a gatekeeper service. It is used to submit jobs, send "ping" 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 <a href="#">GRAM5 User's Guide</a> .
Callback Contact	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.
Job Contact	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.

## 3. Job State Callbacks and Polling

GRAM clients 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.

---

# 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](#)<sup>1</sup> 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 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;
    }
    /*
```

---

<sup>1</sup> gram\_ping\_example.c

```
* Ping 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_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");
}
/*
 * 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_ping_example.c */
```

## 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](#)<sup>2</sup> 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 "globus_gram_protocol.h"

#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]);
        rc = 1;
    }
}
```

---

<sup>2</sup> [gram\\_version\\_example.c](#)

```

    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](#)<sup>3</sup> 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]);

    /*
```

---

<sup>3</sup> gram\_submit\_example.c

```

* 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;
}
/*
* 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(GLOBUS_GRAM_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](#)<sup>4</sup> 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>

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[])
```

---

<sup>4</sup> gram\_submit\_and\_wait\_example.c

```

{
    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 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_callback_disallow(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.

```

```
    */
    rc = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
out:
    return rc;
}
/* End of gram_submit_and_wait_example.c */
```

## 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](#)<sup>5</sup> 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;
    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
```

---

<sup>5</sup> [gram\\_poll\\_example.c](#)

```

    * this program.
    */
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](#)<sup>6</sup> 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;
    }

    /*
     * 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;
    }

    /*
     * Cancel the job named by the first argument to
     * this program.
     */
    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);
    }

    /*
     * Deactivating the module allows it to free memory and close network

```

<sup>6</sup> gram\_cancel\_example.c

```
    * 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 example](#)<sup>7</sup> 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;
    }
    /*
```

---

<sup>7</sup> [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](#)<sup>1</sup> 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>

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)
```

---

<sup>1</sup> [gram\\_nonblocking\\_submit\\_example.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...\n",
            argv[0]);
        rc = 1;

        goto out;
    }

    printf("Submitting %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);
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_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](#)<sup>2</sup> 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>

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
    {

```

<sup>2</sup> [gram\\_attr\\_example.c](#)

```
        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;
    globus_gram_client_attr_t attr;
    struct monitor_t monitor;

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

        goto out;
    }

    printf("Submitting 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](#)<sup>3</sup> can be downloaded.

```

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

```

<sup>3</sup> [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,

```

```
        GLOBUS_GRAM_PROTOCOL_ENVIRONMENT_PARAM);

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 */

```

---

# Chapter 5. Tutorials

The following tutorials are available for GRAM5 developers:

- [GRAM5 Scheduler Interface Tutorial](#)<sup>1</sup>

---

<sup>1</sup> scheduler-tutorial.html

---

# Chapter 6. APIs

## 1. Programming Model Overview

### 1.1. C API Documentation Links

**Table 6.1. GRAM Client APIs**

Name	Purpose
<u>GRAM Protocol</u> <sup>1</sup>	Low-level functions for processing GRAM protocol messages. Symbolic constants for RSL attributes, signals, and job states.
<u>GRAM Client</u> <sup>2</sup>	Functions for submitting job requests, sending signals, and listening for job state updates.
<u>RSL</u> <sup>3</sup>	Functions for parsing and manipulating job specifications in the RSL language.

---

<sup>1</sup> [http://www.globus.org/api/c-globus-5.0.2/globus\\_gram\\_protocol/html/main.html](http://www.globus.org/api/c-globus-5.0.2/globus_gram_protocol/html/main.html)

<sup>2</sup> [http://www.globus.org/api/c-globus-5.0.2/globus\\_gram\\_client/html/main.html](http://www.globus.org/api/c-globus-5.0.2/globus_gram_client/html/main.html)

<sup>3</sup> [http://www.globus.org/api/c-globus-5.0.2/globus\\_rsl/html/main.html](http://www.globus.org/api/c-globus-5.0.2/globus_rsl/html/main.html)

---

# 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 5.0.2 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, eg 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 ( \*. 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.

### 4.1. GRAM5 Common RSL Attributes

<code>arguments</code>	The command line arguments for the executable. Use quotes, if a space is required in a single argument.
<code>count</code>	The number of executions of the executable.
<code>directory</code>	Specifies the path of the directory the jobmanager will use as the default directory for the requested job.

<code>dry_run</code>	If <code>dryrun = yes</code> then the jobmanager will not submit the job for execution and will return success.
<code>environment</code>	The environment variables that will be defined for the executable in addition to default set that is given to the job by the jobmanager.
<code>executable</code>	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.
<code>file_clean_up</code>	Specifies a list of files which will be removed after the job is completed.
<code>file_stage_in</code>	Specifies a list of ("remote URL" "local file") pairs which indicate files to be staged to the nodes which will run the job.
<code>file_stage_in_shared</code>	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.
<code>file_stage_out</code>	Specifies a list of ("local file" "remote URL") pairs which indicate files to be staged from the job to a GASS-compatible file server.
<code>gass_cache</code>	Specifies location to override the GASS cache location.
<code>gram_my_job</code>	Obsolete and ignored.
<code>host_count</code>	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.
<code>job_type</code>	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.)
<code>library_path</code>	Specifies a list of paths to be appended to the system-specific library path environment variables.
<code>max_cpu_time</code>	Explicitly set the maximum cputime for a single execution of the executable. The units is in minutes. The value will go through an <code>atoi()</code> conversion in order to get an integer. If the GRAM scheduler cannot set cputime, then an error will be returned.
<code>max_memory</code>	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 <code>atoi()</code> conversion in order to get an integer. If the GRAM scheduler cannot set <code>maxMemory</code> , then an error will be returned.
<code>max_time</code>	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 <code>atoi()</code> conversion in order to get an integer.
<code>max_wall_time</code>	Explicitly set the maximum walltime for a single execution of the executable. The units is in minutes. The value will go through an <code>atoi()</code> conversion in order to get an integer. If the GRAM scheduler cannot set walltime, then an error will be returned.
<code>min_memory</code>	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 <code>atoi()</code> conversion in order to

	get an integer. If the GRAM scheduler cannot set minMemory, then an error will be returned.
project	Target the job to be allocated to a project account as defined by the scheduler at the defined (remote) resource.
proxy_timeout	Obsolete and ignored. Now a job-manager-wide setting.
queue	Target the job to a queue (class) name as defined by the scheduler at the defined (remote) resource.
remote_io_url	Writes the given value (a URL base string) to a file, and adds the path to that file to the environment 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.
restart	Start a new job manager, but instead of submitting a new job, start managing an existing job. The job manager will search for the job state file created by the original job manager. If it finds the file and successfully reads it, it will become the new manager of the job, sending callbacks on status and streaming stdout/err if appropriate. It will fail if it detects that the old jobmanager is still alive (via a timestamp in the state file). If stdout or stderr was being streamed over the network, new stdout and stderr attributes can be specified in the restart RSL and the jobmanager will stream to the new locations (useful when output is going to a GASS server started by the client that's listening on a dynamic port, and the client was restarted). The new job manager will return a new contact string that should be used to communicate with it. If a jobmanager is restarted multiple times, any of the previous contact strings can be given for the restart attribute.
rsl_substitution	Specifies a list of values which can be substituted into other rsl attributes' values through the \$(SUBSTITUTION) mechanism.
save_state	Causes the jobmanager to save its job state information to a persistent file on disk. If the job manager exits or is suspended, the client can later start up a new job manager which can continue monitoring the job.
scratch_dir	Specifies the location to create a scratch subdirectory in. A SCRATCH_DIRECTORY RSL substitution will be filled with the name of the directory which is created.
stderr	The name of the remote file to store the standard error from the job. If the value is a GASS URL, the standard error from the job is transferred dynamically during the execution of the job.
stderr_position	Specifies where in the file remote standard error streaming should be restarted from. Must be 0.
stdin	The name of the file to be used as standard input for the executable on the remote machine. If the value is a GASS URL, the file is transferred to the remote gass cache before executing the job and removed after the job has terminated.
stdout	The name of the remote file to store the standard output from the job. If the value is a GASS URL, the standard output from the job is transferred dynamically during the execution of the job.
stdout_position	Specifies where in the file remote output streaming should be restarted from. Must be 0.

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

`username` Verify that the job is running as this user.

## 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` /\* relation \*/  
                   | '+' `spec-list` /\* multi-re-quest \*/  
                   | '&' `spec-list` /\* conjunct-re-quest \*/  
                   | '|' `spec-list` /\* disjunct-request \*/
- [2] `spec-list` ::= '(' `specification` ')' `spec-list`  
                   | '(' `specification` ')'
- [3] `relation` ::= 'rsl\_substitution' '=' `binding-sequence` /\* Substitution variable definition \*/  
                   | `attribute op value-sequence` /\* Attribute binding relation \*/
- [4] `binding-sequence` ::= `binding binding-sequence`  
                           | `binding`
- [5] `binding` ::= '(' `string-literal simple-value` ')' /\* Substitution variable definition \*/
- [6] `attribute` ::= `string-literal` /\* attribute \*/
- [7] `op` ::= '=' | '!=' | '>' | '>=' | '<' | '<='
- [8] `value-sequence` ::= `value value-sequence` | `value`
- [9] `value` ::= '(' `value-sequence` ')' | `simple-value`
- [10] `simple-value` ::= `string-literal` /\* String \*/  
                   | `simple-value '#' simple-value` /\* Concatenation \*/  
                   | `implicit-concat` | `variable-reference`
- [11] `variable-reference` ::= '\$(' `string-literal` ')' /\* Variable Reference \*/
- [12] `implicit-concat` ::= (`unquoted-literal`)? (`implicit-concat-core`)+
- [13] `implicit-concat-core` ::= `variable-reference` | (`variable-reference`) (`unquoted-literal`)
- [14] `string-literal` ::= `quoted-literal` | `unquoted-literal`
- [15] `quoted-literal` ::= ''' ([[<sup>^</sup>]]) | (''''')\* ''' /\* Single-quote delimiter with escaping \*/  
                   | "" ([[<sup>^</sup>]]) | (""""')\* "" /\* Double-quote delimiter with escaping \*/  
                   | '^' c ([[<sup>^</sup>c]]) (cc)\* c /\* User defined delimiter c with escaping \*/
- [16] `unquoted-literal` ::= ([[<sup>^</sup>\t\v\n+&|()=<>!'\"#\$%]])+ /\* Non-special characters \*/
- [17] `comment` ::= ('\*' ([[<sup>^</sup>]])|('['<sup>^</sup>]))\* '\*' /\* 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:

```
% 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:

```
% 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:

```
% 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 2, “Errors”](#).

For information about sys admin logging, see [Chapter 9, Admin Debugging](#) in the GRAM5 Admin Guide.

## 1. Troubleshooting tips

In case you run into problems you can do the following

- Check the GRAM5 documentation. Maybe you'll find hints here to solve your problem.
- Check the GRAM5 log for errors.

In case you don't find anything suspicious you can increase the log-level of GRAM5 or other relevant components. Maybe the additional logging-information will tell you what's going wrong.

- Send e-mails to <gram-user@globus.org>. You'll have to subscribe to a list before you can send an e-mail to it. See [here](#)<sup>1</sup> for general e-mail lists and information on how to subscribe to a list and [here](#)<sup>2</sup> for GRAM specific lists.

---

<sup>1</sup> [http://dev.globus.org/wiki/Mailing\\_Lists](http://dev.globus.org/wiki/Mailing_Lists)

<sup>2</sup> [http://dev.globus.org/wiki/GRAM#Mailing\\_Lists](http://dev.globus.org/wiki/GRAM#Mailing_Lists)

## 2. Errors

**Table 9.1. GRAM5 Errors**

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
1	one of the RSL parameters is not supported	Check RSL documentation
2	the RSL length is greater than the maximum allowed	Use RSL substitutions to reduce length of RSL strings
3	an I/O operation failed	Enable trace logging and report to <a href="mailto:gram-dev@globus.org">gram-dev@globus.org</a>
4	jobmanager unable to set default to the directory requested	Check that RSL <code>directory</code> attribute refers to a directory that exists on the target system.
5	the executable does not exist	Check that the RSL <code>executable</code> attribute refers to an executable that exists on the target system.
6	of an unused <code>INSUFFICIENT_FUNDS</code>	Unimplemented feature.
7	authentication with the remote server failed	Check that the contact string contains the proper X.509 DN.
8	the user cancelled the job	Don't cancel jobs you want to complete.
9	the system cancelled the job	Check RSL requirements such as maximum time and memory are valid for the job.
10	data transfer to the server failed	Check gatekeeper and/or job manager logs to see why the process failed.
11	the stdin file does not exist	Check that the RSL <code>stdin</code> attribute refers to a file that exists on the target system or has a valid ftp, gsiftp, http, or https URL.
12	the connection to the server failed (check host and port)	Check that the service is running on the expected TCP/IP port. Check that no firewall prevents contacting that TCP/IP port. Check <code>\$GLOBUS_LOCATION/var/globus-gatekeeper.log</code> for runtime configuration errors.
13	the provided RSL 'maxtime' value is not an integer	Check that the RSL <code>maxtime</code> value evaluates to an integer.
14	the provided RSL 'count' value is not an integer	Check that the RSL <code>count</code> value evaluates to an integer.
15	the job manager received an invalid RSL	Check that the RSL string can be parsed by using <b><code>globusrun -p RSL</code></b> .
16	the job manager failed in allowing others to make contact	Check job manager log.
17	the job failed when the job manager attempted to run it	Verify that the LRM is configured properly.
18	an invalid paradyn was specified	OBSOLETE IN GRAM2
19	the provided RSL 'jobtype' value is invalid	The RSL <code>jobtype</code> attribute is not indicated as supported by the LRM. Valid <code>jobtype</code> values are <code>single</code> , <code>multiple</code> , <code>mpi</code> , and <code>condor</code> .
20	the provided RSL 'myjob' value is invalid	OBSOLETE IN GRAM5

Error Code	Reason	Possible Solutions
21	the job manager failed to locate an internal script argument file	Check that <code>\$GLOBUS_LOCATION/libexec/globus-job-manager-script.pl</code> exists and is executable. Check that the LRM-specific perl module is located in <code>\$GLOBUS_LOCATION/lib/perl/Globus/GRAM/JobManager/</code> directory and is valid. The command <b>perl -I\$GLOBUS_LOCATION/lib/perl \$GLOBUS_LOCATION/lib/perl/Globus/GRAM/JobManager/LRM.pm</b> can be used to check if there are any syntax errors in the script.
22	the job manager failed to create an internal script argument file	Check that your home directory is writable and not full.
23	the job manager detected an invalid job state	Check job manager logs.
24	the job manager detected an invalid script response	Check job manager logs. This is likely a bug in the LRM script.
25	the job manager detected an invalid script status	Check job manager logs. This is likely a bug in the LRM script.
26	the provided RSL 'jobtype' value is not supported by this job manager	Check that the RSL <code>jobtype</code> attribute is implemented by the LRM script. Note that some job types require configuration
27	unused ERROR_UNIMPLEMENTED	LRM does not support some feature included in the job request.
28	the job manager failed to create an internal script submission file	Check that the user's home file system is not full. Check job manager log
29	the job manager cannot find the user proxy	Check that client is delegating a proxy when authenticating with the gatekeeper. Check that the user's home filesystem and the <code>/tmp</code> file system are not full.
30	the job manager failed to open the user proxy	Check that the user's home filesystem and the <code>/tmp</code> file system are not full.
31	the job manager failed to cancel the job as requested	Check that the user's home filesystem and the <code>/tmp</code> file system are not full.
32	system memory allocation failed	Check job manager log for details.
33	the interprocess job communication initialization failed	OBSOLETE IN GRAM5
34	the interprocess job communication setup failed	OBSOLETE IN GRAM5
35	the provided RSL 'host count' value is invalid	Check that the RSL <code>host_count</code> attribute evaluates to an integer.
36	one of the provided RSL parameters is unsupported	Check job manager log for details about invalid parameter.
37	the provided RSL 'queue' parameter is invalid	Check that the RSL <code>queue</code> attribute evaluates to a string that corresponds to an LRM-specific queue name.
38	the provided RSL 'project' parameter is invalid	Check that the RSL <code>project</code> attribute evaluates to a string that corresponds to an LRM-specific project name.

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
39	the provided RSL string includes variables that could not be identified	Check that all RSL substitutions are defined before being used in the job description.
40	the provided RSL 'environment' parameter is invalid	Check that the RSL <code>environment</code> attribute contains a sequence of <code>VARIABLE VALUE</code> pairs.
41	the provided RSL 'dryrun' parameter is invalid	Remove the RSL <code>dryrun</code> attribute from the job description.
42	the provided RSL is invalid (an empty string)	Include a non-empty RSL string in your job submission request.
43	the job manager failed to stage the executable	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.
44	the job manager failed to stage the stdin file	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.
45	the requested job manager type is invalid	OBSOLETE IN GRAM5
46	the provided RSL 'arguments' parameter is invalid	OBSOLETE IN GRAM2
47	the gatekeeper failed to run the job manager	Check the gatekeeper or job manager logs for more information.
48	the provided RSL could not be properly parsed	Check that the RSL string can be parsed by using <b>globusrun -p RSL</b> .
49	there is a version mismatch between GRAM components	Ask system administrator to upgrade GRAM service to GRAM2 or GRAM5
50	the provided RSL 'arguments' parameter is invalid	Check that the RSL <code>arguments</code> attribute evaluates to a sequence of strings.
51	the provided RSL 'count' parameter is invalid	Check that the RSL <code>count</code> attribute evaluates to a positive integer value.
52	the provided RSL 'directory' parameter is invalid	Check that the RSL <code>directory</code> attribute evaluates to a string.
53	the provided RSL 'dryrun' parameter is invalid	Check that the RSL <code>dryrun</code> attribute evaluates to either <code>yes</code> or <code>no</code> .
54	the provided RSL 'environment' parameter is invalid	Check that the RSL <code>environment</code> attribute evaluates to a sequence of <code>VARIABLE, VALUE</code> pairs.
55	the provided RSL 'executable' parameter is invalid	Check that the RSL <code>executable</code> attribute evaluates to a string value.
56	the provided RSL 'host_count' parameter is invalid	Check that the RSL <code>host_count</code> attribute evaluates to a positive integer value.
57	the provided RSL 'jobtype' parameter is invalid	Check that the RSL <code>jobtype</code> attribute evaluates to one of <code>single</code> , <code>multiple</code> , <code>mpi</code> , or <code>condor</code>

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
58	the provided RSL 'maxtime' parameter is invalid	Check that the RSL <code>maxtime</code> attribute evaluates to a positive integer value.
59	the provided RSL 'myjob' parameter is invalid	OBSOLETE IN GRAM5.
60	the provided RSL 'paradyn' parameter is invalid	OBSOLETE IN GRAM2.
61	the provided RSL 'project' parameter is invalid	Check that the RSL <code>project</code> attribute evaluates to a string value.
62	the provided RSL 'queue' parameter is invalid	Check that the RSL <code>queue</code> attribute evaluates to a string value.
63	the provided RSL 'stderr' parameter is invalid	Check that the RSL <code>stderr</code> attribute evaluates to a string value or a sequence of <i>DESTINATION</i> URLs with optional <i>CACHE_TAG</i> string parameters.
64	the provided RSL 'stdin' parameter is invalid	Check that the RSL <code>stdin</code> attribute evaluates to a string value.
65	the provided RSL 'stdout' parameter is invalid	Check that the RSL <code>stdout</code> attribute evaluates to a string value or a sequence of <i>DESTINATION</i> URLs with optional <i>CACHE_TAG</i> string parameters.
66	the job manager failed to locate an internal script	Check job manager log for more details.
67	the job manager failed on the system call <code>pipe()</code>	OBSOLETE IN GRAM5
68	the job manager failed on the system call <code>fcntl()</code>	OBSOLETE IN GRAM2
69	the job manager failed to create the temporary stdout filename	OBSOLETE IN GRAM5
70	the job manager failed to create the temporary stderr filename	OBSOLETE IN GRAM5
71	the job manager failed on the system call <code>fork()</code>	OBSOLETE IN GRAM2
72	the executable file permissions do not allow execution	Check that the RSL <code>executable</code> attribute refers to an executable program or script.
73	the job manager failed to open stdout	Check that the RSL <code>stdout</code> attribute refers to one or more valid destination files or URLs.
74	the job manager failed to open stderr	Check that the RSL <code>stderr</code> attribute refers to one or more valid destination files or URLs.
75	the cache file could not be opened in order to relocate the user proxy	Check that the user's home directory is writable and not full on the GRAM5 service node.
76	cannot access cache files in <code>~/globus/.gass_cache</code> , check permissions, quota, and disk space	Check that the user's home directory is writable and not full on the GRAM5 service node.
77	the job manager failed to insert the contact in the client contact list	Check job manager log

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
78	the contact was not found in the job manager's client contact list	Don't attempt to unregister callback contacts that are not registered
79	connecting to the job manager failed. Possible reasons: job terminated, invalid job contact, network problems, ...	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.
80	the syntax of the job contact is invalid	Check the syntax of job contact string.
81	the executable parameter in the RSL is undefined	Include the RSL <code>executable</code> in all job requests.
82	the job manager service is misconfigured. <code>condor arch</code> undefined	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.
83	the job manager service is misconfigured. <code>condor os</code> undefined	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.
84	the provided RSL 'min_memory' parameter is invalid	Check that the RSL <code>min_memory</code> attribute evaluates to a positive integer value.
85	the provided RSL 'max_memory' parameter is invalid	Check that the RSL <code>max_memory</code> attribute evaluates to a positive integer value.
86	the RSL 'min_memory' value is not zero or greater	Check that the RSL <code>min_memory</code> attribute evaluates to a positive integer value.
87	the RSL 'max_memory' value is not zero or greater	Check that the RSL <code>max_memory</code> attribute evaluates to a positive integer value.
88	the creation of a HTTP message failed	Check job manager log.
89	parsing incoming HTTP message failed	Check job manager log.
90	the packing of information into a HTTP message failed	Check job manager log.
91	an incoming HTTP message did not contain the expected information	Check job manager log.
92	the job manager does not support the service that the client requested	Check that the client is talking to the correct service
93	the gatekeeper failed to find the requested service	OBSOLETE IN GRAM2
94	the jobmanager does not accept any new requests (shutting down)	Execute queries before the job has been cleaned up.
95	the client failed to close the listener associated with the callback URL	Call <code>globus_gram_client_callback_disallow()</code> with a valid the callback contact.
96	the gatekeeper contact cannot be parsed	Check the syntax of the gatekeeper contact string you are attempting to contact.
97	the job manager could not find the 'poe' command	OBSOLETE IN GRAM2
98	the job manager could not find the 'mpirun' command	Configure the LRM script with <code>mpirun</code> in your path.

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
99	the provided RSL 'start_time' parameter is invalid	OBSOLETE IN GRAM2
100	the provided RSL 'reservation_handle' parameter is invalid	OBSOLETE IN GRAM2
101	the provided RSL 'max_wall_time' parameter is invalid	Check that the RSL <code>max_wall_time</code> attribute evaluates to a positive integer.
102	the RSL 'max_wall_time' value is not zero or greater	Check that the RSL <code>max_wall_time</code> attribute evaluates to a positive integer.
103	the provided RSL 'max_cpu_time' parameter is invalid	Check that the RSL <code>max_cpu_time</code> attribute evaluates to a positive integer.
104	the RSL 'max_cpu_time' value is not zero or greater	Check that the RSL <code>max_cpu_time</code> attribute evaluates to a positive integer.
105	the job manager is misconfigured, a scheduler script is missing	Check that the administrator has configured the LRM by running its setup script.
106	the job manager is misconfigured, a scheduler script has invalid permissions	Check that the administrator has installed the <code>GLLOBUS_LOCATION/libexec/globus-job-manager-script.pl</code> script. Check that the file system containing that script allows file execution.
107	the job manager failed to signal the job	OBSOLETE IN GRAM2
108	the job manager did not recognize/support the signal type	Check that your signal operation is using the correct signal constant.
109	the job manager failed to get the job id from the local scheduler	OBSOLETE IN GRAM2
110	the job manager is waiting for a commit signal	Send a two-phase commit signal to the job manager to acknowledge receiving the job contact from the job manager.
111	the job manager timed out while waiting for a commit signal	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.
112	the provided RSL 'save_state' parameter is invalid	Check that the RSL <code>save_state</code> attribute is set to <code>yes</code> or <code>no</code> .
113	the provided RSL 'restart' parameter is invalid	Check that the RSL <code>restart</code> attribute evaluates to a string containing a job contact string.
114	the provided RSL 'two_phase' parameter is invalid	Check that the RSL <code>two_phase</code> attribute evaluates to a positive integer.
115	the RSL 'two_phase' value is not zero or greater	Check that the RSL <code>two_phase</code> attribute evaluates to a positive integer.
116	the provided RSL 'stdout_position' parameter is invalid	OBSOLETE IN GRAM5
117	the RSL 'stdout_position' value is not zero or greater	OBSOLETE IN GRAM5

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
118	the provided RSL 'stderr_position' parameter is invalid	OBSOLETE IN GRAM5
119	the RSL 'stderr_position' value is not zero or greater	OBSOLETE IN GRAM5
120	the job manager restart attempt failed	OBSOLETE IN GRAM2
121	the job state file doesn't exist	Check that the job contact you are trying to restart matches one that the job manager returned to you.
122	could not read the job state file	Check that the state file directory is not full.
123	could not write the job state file	Check that the state file directory is not full.
124	old job manager is still alive	Contact the returned job manager contact to manage the job you are trying to restart.
125	job manager state file TTL expired	OBSOLETE in GRAM2
126	it is unknown if the job was submitted	Check job manager log.
127	the provided RSL 'remote_io_url' parameter is invalid	Check that the RSL <code>remote_io_url</code> attribute evaluates to a string value.
128	could not write the remote io url file	Check that the user's home file system on the job manager service node is writable and not full.
129	the standard output/error size is different	Send a stdio update signal to redirect the job manager output to a new URL
130	the job manager was sent a stop signal (job is still running)	Submit a restart request to monitor the job.
131	the user proxy expired (job is still running)	Generate a new proxy and then submit a restart request to monitor the job.
132	the job was not submitted by original job-manager	OBSOLETE IN GRAM2
133	the job manager is not waiting for that commit signal	Do not send a commit signal to a job that is not waiting for a commit signal.
134	the provided RSL scheduler specific parameter is invalid	Check the LRM-specific documentation to determine what values are legal for the RSL extensions implemented by the LRM.
135	the job manager could not stage in a file	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.
136	the scratch directory could not be created	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.
137	the provided 'gass_cache' parameter is invalid	Check that the RSL <code>gass_cache</code> attribute evaluates to a string.
138	the RSL contains attributes which are not valid for job submission	Do not use restart- or signal-only RSL attributes when submitting a job.

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
139	the RSL contains attributes which are not valid for stdio update	Do not use submit- or restart-only RSL attributes when sending a stdio update signal to a job.
140	the RSL contains attributes which are not valid for job restart	Do not use submit- or signal-only RSL attributes when restarting a job.
141	the provided RSL 'file_stage_in' parameter is invalid	Check that the RSL <code>file_stage_in</code> attribute evaluates to a sequence of <i>SOURCE DESTINATION</i> pairs.
142	the provided RSL 'file_stage_in_shared' parameter is invalid	Check that the RSL <code>file_stage_in_shared</code> attribute evaluates to a sequence of <i>SOURCE DESTINATION</i> pairs.
143	the provided RSL 'file_stage_out' parameter is invalid	Check that the RSL <code>file_stage_out</code> attribute evaluates to a sequence of <i>SOURCE DESTINATION</i> pairs.
144	the provided RSL 'gass_cache' parameter is invalid	Check that the RSL <code>gass_cache</code> attribute evaluates to a string.
145	the provided RSL 'file_cleanup' parameter is invalid	Check that the RSL <code>file_clean_up</code> attribute evaluates to a sequence of strings.
146	the provided RSL 'scratch_dir' parameter is invalid	Check that the RSL <code>scratch_dir</code> attribute evaluates to a string.
147	the provided scheduler-specific RSL parameter is invalid	Check the LRM-specific documentation to determine what values are legal for the RSL extensions implemented by the LRM.
148	a required RSL attribute was not defined in the RSL spec	Check that the RSL <code>executable</code> attribute is present in your job request RSL. Check that the RSL <code>restart</code> attributes is present in your restart RSL.
149	the <code>gass_cache</code> attribute points to an invalid cache directory	Check that the RSL <code>gass_cache</code> attributes evaluates to a directory that exists or can be created. Check that the user's home file system is writable and not full.
150	the provided RSL 'save_state' parameter has an invalid value	Check that the RSL <code>save_state</code> attribute has a value of <code>yes</code> or <code>no</code> .
151	the job manager could not open the RSL attribute validation file	Check that <code>\$GLOBUS_LOCATION/share/globus_gram_job_manager/globus-gram-job-manager.rvf</code> is present and readable on the job manager service node. Check that <code>\$GLOBUS_LOCATION/share/globus_gram_job_manager/LRM.rvf</code> is readable on the job manager service node if present.
152	the job manager could not read the RSL attribute validation file	Check that <code>\$GLOBUS_LOCATION/share/globus_gram_job_manager/globus-gram-job-manager.rvf</code> is valid. Check that <code>\$GLOBUS_LOCATION/share/globus_gram_job_manager/LRM.rvf</code> is valid if present.
153	the provided RSL 'proxy_timeout' is invalid	Check that RSL <code>proxy_timeout</code> attribute evaluates to a positive integer.
154	the RSL 'proxy_timeout' value is not greater than zero	Check that RSL <code>proxy_timeout</code> attribute evaluates to a positive integer.

<b>Error Code</b>	<b>Reason</b>	<b>Possible Solutions</b>
155	the job manager could not stage out a file	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.
156	the job contact string does not match any which the job manager is handling	Check that the job contact string matches one returned from a job request.
157	proxy delegation failed	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.
158	the job manager could not lock the state lock file	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.
159	an invalid globus_io_clientattr_t was used.	Check that you have initialized the globus_io_clientattr_t attribute prior to using it with the GRAM client API.
160	an null parameter was passed to the gram library	Check that you are passing legal values to all GRAM API calls.
161	the job manager is still streaming output	OBSOLETE IN GRAM5
162	the authorization system denied the request	Check with your GRAM system administrator to allow a particular certificate to be authorized.
163	the authorization system reported a failure	Check with your system administrator to verify that the authorization system is configured properly.
164	the authorization system denied the request - invalid job id	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.
165	the authorization system denied the request - not authorized to run the specified executable	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.
166	the provided RSL 'user_name' parameter is invalid.	Check that the RSL user_name attribute evaluates to a string.
167	the job is not running in the account named by the 'user_name' parameter.	Ask with the GRAM system administrator to add an authorization entry to allow your credential to run jobs as the specified user account.

---

# 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

```
rs1: &( executable = "/bin/echo" )  
    ( arguments = "hello" )
```

becomes

```
rsl: "&( executable = \"bin/echo\" )
      (arguments = \"hello\" )"
```

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:

<code>exit-code</code>	Job exit code. Sent in job state callbacks and in job status replies when the job completes.
<code>gt3-failure-type</code>	Failure detail type for staging errors. Sent in job state callbacks and in job status replies when a job fails.
<code>gt3-failure-message</code>	Failure detail message for more context for errors. Sent in job state callbacks and in job status replies when a job fails.
<code>gt3-failure-source</code>	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.
<code>gt3-failure-destination</code>	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.
<code>version</code>	Job manager package version. Sent in all messages from the job manager.
<code>toolkit-version</code>	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

<code>job-manager-name</code>	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".
<code>host-name</code>	The name of the host on which the gatekeeper is running. This exists only for compatibility with the HTTP/1.1 protocol.
<code>message-size</code>	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 username 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:

<i>job-contact</i>	The job contact string returned in a response to a job request message, or determined by querying the MDS system.
<i>host-name</i>	The name of the host on which the job manager is running. This exists only for compatibility with the HTTP/1.1 protocol.
<i>message-size</i>	The length of the content of the message, not including the HTTP/1.1 header.
<i>version</i>	The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string "2".
<i>callback-contact</i>	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:

<i>job-contact</i>	The job contact string returned in a response to a job request message, or determined by querying the MDS system.
<i>host-name</i>	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.

<i>host-name</i>	The host part of the callback-contact URL. This exists only for compatibility with the HTTP/1.1 protocol.
<i>message-size</i>	The length of the content of the message, not including the HTTP/1.1 header.
<i>version</i>	The version of the GRAM protocol which is being used. For the protocol defined in this document, the value must be the string 2.
<i>job-contact</i>	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:

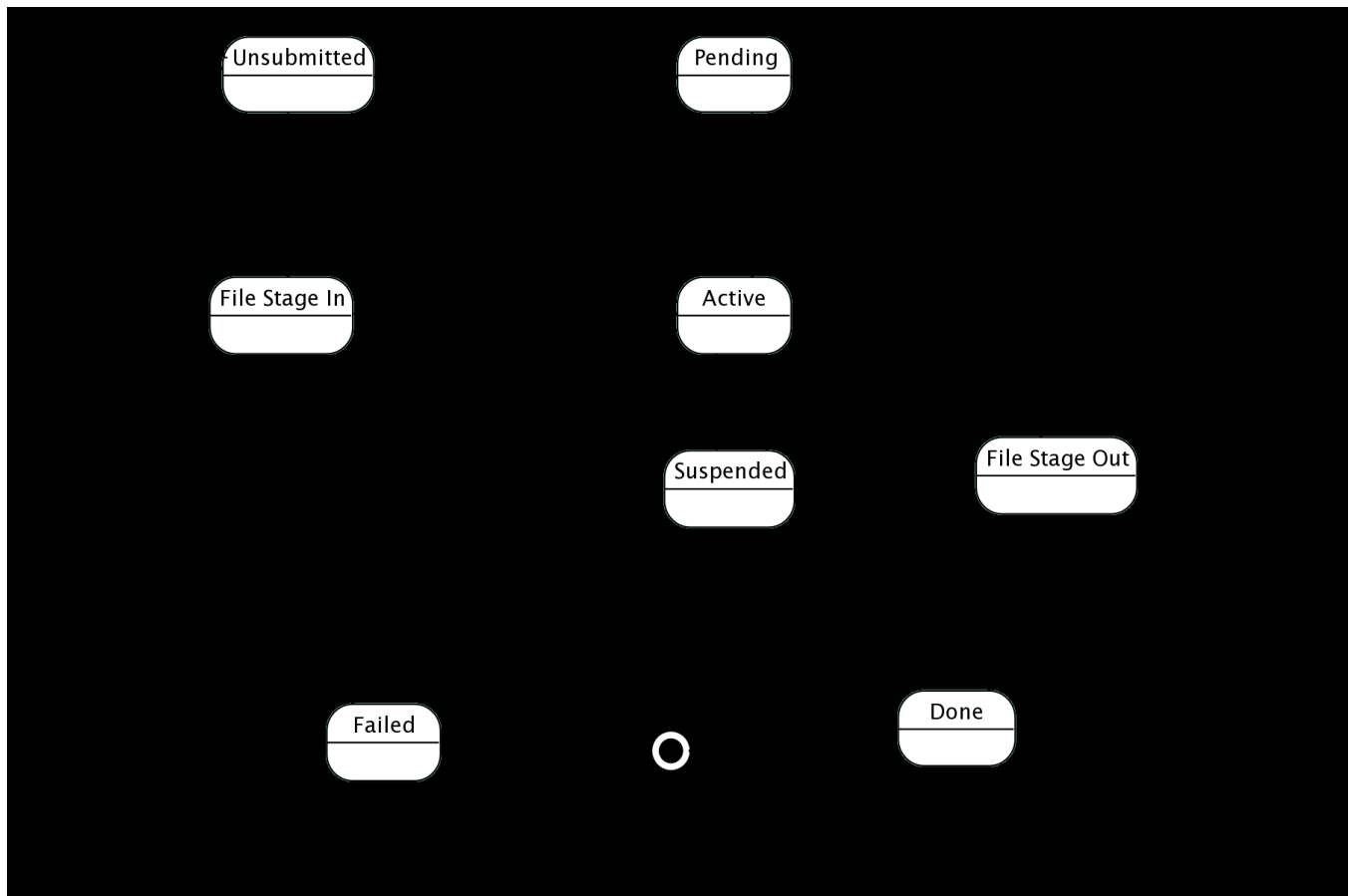
- Authentication must be done using GSSAPI mutual authentication
- Messages must be wrapped with support for the delegation message. When using Globus I/O, this is accomplished by using the the GLOBUS\_IO\_SECURE\_CHANNEL\_MODE\_GSI\_WRAP wrapping mode.

## 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**

State	Meaning
GLOBUS_GRAM_PROTOCOL_JOB_STATE_UNSUBMITTED	Initial job state
GLOBUS_GRAM_PROTOCOL_JOB_STATE_STAGE_IN	Job staging in progress
GLOBUS_GRAM_PROTOCOL_JOB_STATE_PENDING	Job submitted to LRM, awaiting execution
GLOBUS_GRAM_PROTOCOL_JOB_STATE_ACTIVE	Job executing
GLOBUS_GRAM_PROTOCOL_JOB_STATE_SUSPENDED	Job made progress executing but is now suspended
GLOBUS_GRAM_PROTOCOL_JOB_STATE_STAGE_OUT	Job staging in progress after job completed
GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE	Job completed successfully
GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED	Job was canceled or failed

**Figure 10.1. GRAM State Transitions**

---

# Chapter 11. Related Documentation

No related documentation links have been determined at this time.

---

# Chapter 12. Internal Components

Internal Components<sup>1</sup>

---

<sup>1</sup> [internal-components.html](#)

---

# Glossary

---

# Index

## A

apis, 29  
    overview, 29

## C

compatibility, 1

## D

debugging, 38  
dependencies, 2

## E

errors, 41

## F

features, 1

## P

platforms, tested, 1

## T

troubleshooting, 40  
    check documentation, 40  
    errors, 40  
    gram log, 40  
    mailing lists, 40