GT 4.2.0 GridWay: Developer's Guide

GT 4.2.0 GridWay: Developer's Guide

Published May, 2008
Copyright © 2002-2008 GridWay Team, Distributed Systems Architecture Group, Universidad Complutense de Madrid (dsa-research.org).

Introduction

This guide is intended to help a developer interested in extending GridWay's functionality to understand its architecture, mainly to show how to develop new Middleware Access Drivers (MADs). It is also a good start point and reference for anyone interested in programming applications using GridWay's implementation of the DRMAA library (C, Java, Perl, Python and Ruby bindings).

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0[1]

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Any academic report, publication, or other academic disclosure of results obtained with the GridWay Metascheduler will acknowledge GridWay's use by an appropriate citation to relevant papers by GridWay team members.

DRAFT

[1] http://www.apache.org/licenses/LICENSE-2.0

Table of Contents

1. Before you begin ... 1
    1. Feature summary ... 1
    2. Tested platforms ... 2
    3. Backward compatibility summary ... 2
    4. Technology dependencies ... 2
    5. Security Considerations for GridWay ... 3
2. Usage scenarios ... 4
    1. Simple Applications ... 4
3. Tutorials ... 8
    1. DRMAA Program Structure and Compilation ... 8
    2. DRMAA Programming Howto ... 9
    3. External Tutorials and Material ... 23
4. Architecture and design overview ... 24
    1. Architecture ... 24
    2. Programming Model Overview ... 25
5. APIs ... 27
    1. DRMAA bindings for C and Java ... 27
    2. DRMAA bindings for scripting languages. ... 27
6. Non-WSDL protocols ... 30
    1. Information manager MAD ... 30
    2. Execution manager MAD ... 32
    3. Transfer manager MAD ... 32
    4. Dispatch manager Scheduler ... 33
I. GridWay Commands ... 35
    Job and Array Job submission Command ... 36
    DAG Job submission Command ... 37
    Job Monitoring Command ... 38
    Job History Command ... 40
    Host Monitoring Command ... 41
    Job Control Command ... 43
    Job Synchronization Command ... 44
    User Monitoring Command ... 45
    Accounting Command ... 46
    JSDL To GridWay Job Template Parser Command ... 47
7. Debugging ... 48
    1. Debugging in Java WS Core ... 48
    2. Debugging in GridWay ... 48
8. Troubleshooting ... 50
    1. Errors ... 50
    2. Debugging ... 50
9. GT 4.2.0 Samples for GridWay ... 51
    1. DRMAA examples ... 51

DRAFT

List of Figures

4.1. Components of the GridWay Meta-scheduler. 24
4.2. Grid Development Model with DRMAA 25
4.3. Embarrassingly Distributed Applications schema 25
4.4. Master-Worker Applications schema 26

DRAFT

List of Tables

5.1. Translation from C to Scripting language ... 28
6.1. Attributes that should be defined by the Information MADs. ... 31
3. Field options ... 39
4. Field information ... 40
5. Field information ... 41
6. Queue field information ... 42
7. Field information ... 45
8. Field information ... 46
8.1. Gridway Errors ... 50

DRAFT

Chapter 1. Before you begin

1. Feature summary

Advanced Scheduling Capabilities

GridWay implements several state-of-the-art Grid-aware scheduling policies, comprising *job prioritization* policies (fixed priority, urgency, share, deadline and waiting-time) and *resource prioritization* policies (fixed priority, usage, failure and rank).

These policies are combined with:

- *Adaptive Scheduling*, to periodically adapt the schedule considering applications' demands and Grid resource characteristics.
- *Adaptive Execution* to migrate running applications in terms of resource availability, capacity or cost, and new application requirements or preferences.

Transparent Grid Access

GridWay interfaces infrastructures with different middleware stacks. With GridWay, users can access heterogeneous resources in a transparent way. For example, it can access resources configured with both GRAM pre web services and GRAM web services. It also permits the use of different grids with different software stacks, for example, Teragrid with Globus and EGEE with gLite.

Flexible Deployment Capabilities

GridWay supports multiple-user operation mode, and does not require additional middleware installation (apart from standard Globus services). Globus installation is not required in each end-user system.

GridWay allows different Grid deployment strategies, like Enterprise Grids, Partner Grids or Utility Grids.

Different Application Profiles

GridWay executes different Grid application profiles:

- Array (Bulk) jobs, for parameter sweep applications
- DAG Workflows
- Single-site MPI applications

Fault Detection and Recovery

GridWay is able to detect several problems that can occur when executing a remote job. It also implements mechanisms that make the execution more reliable. It can detect a remote system crash, a job failure (via the job exit code) or even a network disconnection (using the polling mechanism) and migrate the problematic job to another resource.

GridWay also performs periodic saves of its state in order to recover from local failure.

Reporting and Accounting

GridWay provides detailed statistics of Grid usage. In this way, the Grid administrator can properly plan usage policies and forecast workload. In addition, these statistics can be used by the scheduler to predict (per user) Grid resource response time.

Standard Compliance	GridWay is an open-source project, flexible and completely based on standards to leverage its usability and interoperability. For example, users can describe their jobs using JSDL. Similarly, programmers can build grid enabled applications using the DRMAA standard.
User Interface	GridWay provides end-users with a familiar environment similar to that found on classical LRM systems. So GridWay CLI eases the adoption of Grid technologies.

2. Tested platforms

Tested platforms for GridWay:

- GridWay builds successfully for the following platforms:
  - Linux
  - Tru64
  - Mac OS X
  - Solaris
  - Aix

In addition, GridWay has been tested with the major Grid infrastructures. Click the following links to find more information on how to use GridWay with EGEE[1],TeraGrid[2],OSG[3] and NorduGrid[4].

3. Backward compatibility summary

GT 4.2.0 includes a new stable release, GridWay 5.4.0, that ships with functionality additions focused in extending usability for the end users. The following information regards compatibility with the previous stable version included in GT 4.0.7:

API changes since 4.0.7:

- Added DRMAA scripting language bindings for Python, Ruby and Perl.

CLI changes since 4.0.7:

- Added gwdag, a tool to run Condor DAGMAN compatible workflows.

Configuration interface changes since 4.0.7:

- None

4. Technology dependencies

GridWay uses different Globus services to perform the tasks of information gathering, job execution and data transfer.

[1] http://www.gridway.org/documentation/stable/egeehowto
[2] http://www.gridway.org/documentation/stable/tghowto/
[3] http://www.gridway.org/documentation/stable/osghowto/
[4] http://www.gridway.org/documentation/stable/nghowtoo/

- GridWay depends on the following GT components for job execution:
  - GRAM: GridWay can interface with both GRAM 2 (pre web services) and GRAM 4 (web services)
- GridWay depends on the following GT components for data staging:
  - RFT
  - GridFTP
- GridWay depends on the following GT component for information gathering:
  - MDS
- GridWay relies on the Globus basic security infrastructure for authentication and authorization. On top of that, GridWay can use other Globus services and components to complement this infrastructure:
  - Delegation Service
  - MyProxy

5. Security Considerations for GridWay

Access authorization to the GridWay server is done based on the Unix identity of the user (accessing GridWay directly or through a Web Services GRAM, as in GridGateWay[5]). Hence, security in GridWay has the same implications as the Unix accounts of their users.

Also, GridWay uses proxy certificates to use Globus services, so the security implications of managing certificates also must be taken into account

[5] http://www.grid4utility.org

DRAFT

Chapter 2. Usage scenarios

1. Simple Applications

1.1. A simple computational problem

This is a well known exercise. For our purposes, we will calculate the integral of the following function f(x) = 4/(1+x2). So, π will be the integral of f(x) in the interval [0,1].

In order to calculate the whole integral, it's interesting to divide the function in several tasks and compute its area. The following program computes the area of a set of intervals, assigned to a given task:

```
#include <stdio.h>
#include <string.h>

int main (int argc, char** args)
{

int task_id;
int total_tasks;
long long int n;
long long int i;

double l_sum, x, h;

task_id = atoi(args[1]);
total_tasks = atoi(args[2]);
n = atoll(args[3]);

fprintf(stderr, "task_id=%d total_tasks=%d n=%lld\n", task_id,
total_tasks, n);

h = 1.0/n;

l_sum = 0.0;

for (i = task_id; i < n; i += total_tasks)
{
    x = (i + 0.5)*h;
    l_sum += 4.0/(1.0 + x*x);
}

l_sum *= h;

printf("%0.12g\n", l_sum);

return 0;
}
```

DRAFT

We will use this program (**pi**) to develop our DRMAA distributed version.

1.2. The DRMAA code

1.2.1. Setting up the job template

Let us start with the definition of each tasks. As you can see, the previous program needs the number of intervals, total tasks, and the task number. These variables are available to compile job templates through the DRMAA_GW_TASK_ID and DRMAA_GW_TOTAL_TASKS predefined strings.

Also, each task must generate a different standard output file, with its partial result. We can use the standard DRMAA_PLACEHOLDER_INCR predefined string to set up different filenames for each task, so they will not overwrite each others output.

```
void setup_job_template( drmaa_job_template_t **jt)
{
char         error[DRMAA_ERROR_STRING_BUFFER];
int          rc;
char         cwd[DRMAA_ATTR_BUFFER];

const char   *args[4] = {DRMAA_GW_TASK_ID,
                         DRMAA_GW_TOTAL_TASKS,
                         "10000000",
                         NULL};

rc = drmaa_allocate_job_template(jt, error, DRMAA_ERROR_STRING_BUFFER);

getcwd(cwd, DRMAA_ATTR_BUFFER)

rc = drmaa_set_attribute(*jt,
                          DRMAA_WD,
                          cwd,
                          error,
                          DRMAA_ERROR_STRING_BUFFER);

rc = drmaa_set_attribute(*jt,
                          DRMAA_JOB_NAME,
                          "pi.drmaa",
                          error,
                          DRMAA_ERROR_STRING_BUFFER);

rc = drmaa_set_attribute(*jt,
                          DRMAA_REMOTE_COMMAND,
                          "pi",
                          error,
                          DRMAA_ERROR_STRING_BUFFER);


rc = drmaa_set_vector_attribute(*jt,
                                 DRMAA_V_ARGV,
                                 args,
```

```
                                            error,
                                            DRMAA_ERROR_STRING_BUFFER);

      rc = drmaa_set_attribute(*jt,
                               DRMAA_OUTPUT_PATH,
                               "stdout."DRMAA_PLACEHOLDER_INCR,
                               error,
                               DRMAA_ERROR_STRING_BUFFER);
      }
```

1.2.2. The main DRMAA routine

The DRMAA program just submits a given number of tasks, each one will compute its section of the previous integral. Then we will synchronize these tasks, aggregate the partial results to get the total value. Note that this results are passed to the DRMAA program through the tasks standard output.

```
int main(int argc, char **argv)
{
int  rc;
int  end, i;

char error[DRMAA_ERROR_STRING_BUFFER];
char value[DRMAA_ATTR_BUFFER];
char attr_buffer[DRMAA_ATTR_BUFFER];

const char *job_ids[1] ={DRMAA_JOB_IDS_SESSION_ALL};

drmaa_job_template_t *jt;
drmaa_job_ids_t      *jobids;

FILE *fp;
float pi, pi_t;

if ( argc != 2)
{
  fprintf(stderr,"Usage drmaa_pi <number_of_tasks>\n");
  return -1;
}
else
  end = atoi(argv[1]) - 1;

rc = drmaa_init(NULL, error, DRMAA_ERROR_STRING_BUFFER-1);

setup_job_template(&jt);

rc =drmaa_run_bulk_jobs(&jobids,
                        jt,
                        0,
                        end,
                        1,
```

```
                        error,
                        DRMAA_ERROR_STRING_BUFFER);

fprintf(stderr,"Waiting for bulk job to finish...\n");

rc = drmaa_synchronize(job_ids,
                       DRMAA_TIMEOUT_WAIT_FOREVER,
                       DISPOSE,
                       error,
                       DRMAA_ERROR_STRING_BUFFER);

fprintf(stderr,"All Jobs finished\n");

pi = 0.0;

for(i=0;i<=end;i++)
{
  snprintf(attr_buffer,DRMAA_ATTR_BUFFER,"stdout.%s",i);

  fp = fopen(attr_buffer,"r");
  fscanf(fp,"%f",&pi_t);
  fprintf(stderr,"Partial computed by task %i = %1.30f\n",i,pi_t);
  fclose(fp);

  pi += pi_t;
}

drmaa_release_job_ids(jobids);

fprintf(stderr,"\nPI=%1.30f\n",pi);

drmaa_exit(NULL, 0);

return 0;
}
```

DRAFT

Chapter 3. Tutorials

This chapter is a tutorial for getting started programming DRMAA applications with GridWay. Although is not necessary that you already know how GridWay works, prior experience submitting and controlling jobs with GridWay will come in handy. This tutorial shows the use of the most important functions in the DRMAA standard, and gives you some hints to use the GridWay DRMAA library.

The source code for the following examples can be found in the `$GW_LOCATION/examples` directory.

1. DRMAA Program Structure and Compilation

1.1. DRMAA C Binding

You have to include the following file in your C sources to use the GridWay DRMAA C bindings library:

```
#include "drmaa.h"
```

Also add the following compiler options to link your program with the DRMAA library:

```
-L $GW_LOCATION/lib
-I $GW_LOCATION/include
-ldrmaa
```

1.2. DRMAA JAVA Binding

You have to import the GridWay DRMAA JAVA package:

```
     import org.ggf.drmaa.*;
```

Add the following option to the **javac**:

```
 -classpath $(CLASSPATH):$GW_LOCATION/lib/drmaa.jar
```

Also do not forget to update your shared library path:

```
 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$GW_LOCATION/lib
```

DRAFT

2. DRMAA Programming Howto

2.1. DRMAA Sessions

Let us start with the initialization steps of every DRMAA program. Before any call to a DRMAA function you must start a DRMAA session. The session is used to manage the jobs submitted in your Grid application. Before ending the program you should disengage from the previously started session, to free the internal session structures.

Also, as you will see every DRMAA function returns an error code that allows you to check whether the call was successful or not (DRMAA_ERRNO_SUCCESS, if everything goes well). Through out this tutorial, for the shake of clarity, we will get rid of error checking code in most cases. But remember that you should check return codes.

The following example shows you how to manage a DRMAA session. It also shows some information about the DRMAA implementation and the DRMS you are using.

```
char         error[DRMAA_ERROR_STRING_BUFFER];
int          result;
char         contact[DRMAA_ATTR_BUFFER];
unsigned int major;
unsigned int minor;
char         drm[DRMAA_ATTR_BUFFER];
char         impl[DRMAA_ATTR_BUFFER];

result = drmaa_init (NULL,
                     error,
                     DRMAA_ERROR_STRING_BUFFER-1);        (See 1)

if ( result != DRMAA_ERRNO_SUCCESS)
{
    fprintf(stderr,"drmaa_init() failed: %s\n", error);
    return -1;
}
else
    printf("drmaa_init() success \n");


drmaa_get_contact(contact,
                  DRMAA_ATTR_BUFFER-1,
                  error,
                  DRMAA_ERROR_STRING_BUFFER-1);            (See 2)

drmaa_version(&major,
              &minor,
              error,
              DRMAA_ERROR_STRING_BUFFER);

drmaa_get_DRM_system(drm,
                     DRMAA_ATTR_BUFFER-1,
                     error,
                     DRMAA_ERROR_STRING_BUFFER-1);

drmaa_get_DRMAA_implementation(impl,
```

```
                                DRMAA_ATTR_BUFFER-1,
                                error,
                                DRMAA_ERROR_STRING_BUFFER-1);

printf("Using %s, details:\n",impl);
printf("\t DRMAA version %i.%i\n",major,minor);
printf("\t DRMS %s (contact: %s)\n",drm,contact);

result = drmaa_exit (error, DRMAA_ERROR_STRING_BUFFER-1);  (See 3)

if ( result != DRMAA_ERRNO_SUCCESS)
{
    fprintf(stderr,"drmaa_exit() failed: %s\n", error);
    return -1;
}

printf("drmaa_exit() success \n");
```

1. drmaa_init() is the first DRMAA function in a DRMAA program, and must be called only once per DRMAA program. Almost every DRMAA function uses to special arguments for error reporting pourposes: a string buffer (error), and its length (DRMAA_ERROR_STRING_BUFFER-1). As you can see DRMAA has some predefined buffer sizes for char * variables that you can use, although they are not mandatory.

   The first parameter of the drmaa_init() function is an implementation dependent string which may be used to specify which DRM system to use. In the GridWay library it can be used to select which GW daemon you want to connect to, NULL means the default option, localhost.

   If you try to call a DRMAA function outside a session, it will return the error code DRMAA_ERRNO_NO_ACTIVE_SESSION.

2. This program also shows some information about the DRMAA implementation you are using, by calling the functions drmaa_get_contact(), drmaa_version(), drmaa_get_DRM_system() and drmaa_get_DRMAA_implementation(). You should see something like this:

   ```
   Using DRMAA for GridWay 4.7, details:
   DRMAA version 1.0
   DRMS GridWay (contact: localhost)
   ```

3. At the end of the program you should call drmaa_exit() to clean up the library internal structures. Beyond this point you should not be calling any DRMAA function (there are some exceptions like the information functions above, like drmaa_get_contact(),...)

This example shows the same program using the DRMAA JAVA bindings.

```
import org.ggf.gridway.drmaa.*;

import java.util.*;

public class Howto1
{
```

DRAFT

```
  public static void main (String[] args)
  {
    SessionFactory  factory = SessionFactory.getFactory();
    Session         session = factory.getSession();

    try
    {
      session.init(null);

      System.out.println("Session Init success");

      System.out.println ("Using " + session.getDRMAAImplementation()
                          + ", details:");

      System.out.println ("\t DRMAA version " + session.getVersion());

      System.out.println ("\t DRMS " + session.getDRMSInfo() +
                          "(contact: " + session.getContact() + ")");

      session.exit();

      System.out.println("Session Exit success");

    }
    catch (DrmaaException e)
    {
      e.printStackTrace();
    }
  }
}
```

2.2. Job Template Compilation

So we already have an active session, how do we use it to submit jobs. The first thing to do is to provide a description of your job. A job is described by its job template (a drmaa_job_template_t variable), which in turns is a structure to store information about your job (things like the executable, its arguments or the output files).

The DRMAA standard provides several pre-defined strings to refer to common job template attributes (those staring with DRMAA_ like DRMAA_REMOTE_COMMAND). The GridWay library also defines some macros to refer to GridWay specific attributes (those staring with DRMAA_GW_ like DRMAA_GW_INPUT_FILES).

There are two kind of attributes: scalar and vector. Scalar attributes are simple strings (char *) and corresponds to template attributes in the form:

```
attribute = value
```

You can use the drmaa_set_attribute() and drmaa_get_attr_value() to manage these scalar attributes. On the other hand, vector attributes corresponds to job template variables with one or more values i.e:

```
attribute = value1 value2 ... valueN
```

A vector attribute is NULL terminated array of strings (char **). You can use the drmaa_set_vector_attribute() and drmaa_get_next_attr_value() to deal with vector attributes.

We will use a common job template for the rest of the tutorial, so we will make a function to set up this job template. Remember to check the return codes of the DRMAA functions.

```
void setup_job_template(drmaa_job_template_t **jt)
{
 char         error[DRMAA_ERROR_STRING_BUFFER];
 int          rc;
 char         cwd[DRMAA_ATTR_BUFFER];

 const char  *args[3] = {"-l", "-a", NULL};              (See 1)

 rc = drmaa_allocate_job_template(jt, error, DRMAA_ERROR_STRING_BUFFER-1);
                                                         (See 2)
 if ( rc != DRMAA_ERRNO_SUCCESS)
 {         (See 3)
     fprintf(stderr,"drmaa_allocate_job_template() failed: %s\n", error);
     exit(-1);
 }

 if ( getcwd(cwd, DRMAA_ATTR_BUFFER) == NULL )
 {
     perror("Error getting current working directory");
     exit(-1);
 }

 rc = drmaa_set_attribute(*jt,                           (See 4)
                          DRMAA_WD,
                          cwd,
                          error,
                          DRMAA_ERROR_STRING_BUFFER-1);

 if ( rc != DRMAA_ERRNO_SUCCESS )
 {
     fprintf(stderr,"Error setting job template attribute: %s\n",error);
     exit(-1);
 }

 rc = drmaa_set_attribute(*jt,
                          DRMAA_JOB_NAME,
                          "ht2",
                          error,
                          DRMAA_ERROR_STRING_BUFFER-1);

 rc = drmaa_set_attribute(*jt,                           (See 5)
                          DRMAA_REMOTE_COMMAND,
                          "/bin/ls",
```

```
                          error,
                          DRMAA_ERROR_STRING_BUFFER-1);

  rc = drmaa_set_vector_attribute(*jt,                    (See 6)
                                 DRMAA_V_ARGV,
                                 args,
                                 error,
                                 DRMAA_ERROR_STRING_BUFFER-1);

  if ( rc != DRMAA_ERRNO_SUCCESS )
  {
      fprintf(stderr,"Error setting remote command arguments: %s\n",error);
      exit(-1);
  }

  rc = drmaa_set_attribute(*jt,                            (See 7)
                          DRMAA_OUTPUT_PATH,
                          "stdout."DRMAA_GW_JOB_ID,
                          error,
                          DRMAA_ERROR_STRING_BUFFER-1);

  rc = drmaa_set_attribute(*jt,
                          DRMAA_ERROR_PATH,
                          "stderr."DRMAA_GW_JOB_ID,
                          error,
                          DRMAA_ERROR_STRING_BUFFER-1);
```

1. As stated before, every DRMAA function returns an error code that should be checked. Check the DRMAA Reference Guide to find out the error codes returned by each function.
2. In this code we set up a job template for a simple "ls" command. The arguments are a vector attribute, in this case we are using two arguments "-l" and "-a". Note that we use an array with 3 elements, the last one must be always NULL.
3. Before using a job template you must allocate it. This function allocates memory for the library internal structures of your job template. Do not forget to free the job template with drmaa_delete_job_template() function.
4. The GridWay DRMAA implementation will generate a job template file in the job working directory (DRMAA_WD). This attribute value should be defined, it defaults to the program current working directory (DRMAA_PLACEHOLDER_WD) . Please note that all files are named relative to the wotking directory. DRMAA_WD is a local path name, this directory will be "recreated" (by the use of the DRMAA_GW_INPUT_FILES variable) in the remote host, and it will be the working directory of the job on the execution host.
5. This is the executable we are going to submit to the Grid, a simple "ls".
6. Here we set the arguments, note we are using the vector attribute function.
7. The standard output of our "ls" command will be stored (under the DRMAA_WD directory) with name "stdout."DRMAA_GW_JOB_ID. In this case we are using an specific GridWay define DRMAA_GW_JOB_ID, which corresponds to ${JOB_ID}. So, if our job is assigned with id 34 by GridWay, we will have its standard output in "stdout.34". Use always this naming pattern not to over-write previously generated files.

If everything went well, the following job template will be generated:

```
#This file was automatically generated by the GridWay DRMAA library
EXECUTABLE=/bin/ls
ARGUMENTS= -l -a
STDOUT_FILE=stdout.${JOB_ID}
STDERR_FILE=stderr.${JOB_ID}
RESCHEDULE_ON_FAILURE=no
NUMBER_OF_RETRIES=3
```

This fragment of code shows you how to construct the same template using the DRMAA JAVA bindings.

```
   jt = session.createJobTemplate();

   jt.setWorkingDirectory(java.lang.System.getProperty("user.dir"));

   jt.setJobName("ht2");

   jt.setRemoteCommand("/bin/ls");

   jt.setArgs(new String[] {"-l","-a"});

   jt.setOutputPath("stdout." + SessionImpl.DRMAA_GW_JOB_ID);

   jt.setErrorPath ("stderr." + SessionImpl.DRMAA_GW_JOB_ID);
```

2.3. Single Job Submission

We can now submit our "ls" to the Grid. The next example shows you how to submit your job, and how to synchronize its execution. The resource usage made by your job is also shown.

```
int main(int argc, char *argv[])
{
 char                    error[DRMAA_ERROR_STRING_BUFFER];
 int                     result;
 drmaa_job_template_t *  jt;
 char                    job_id[DRMAA_JOBNAME_BUFFER];
 char                    job_id_out[DRMAA_JOBNAME_BUFFER];
 drmaa_attr_values_t *   rusage;
 int                     stat;
 char                    attr_value[DRMAA_ATTR_BUFFER];

 result = drmaa_init (NULL, error, DRMAA_ERROR_STRING_BUFFER-1);

 if ( result != DRMAA_ERRNO_SUCCESS)
 {
   fprintf(stderr,"drmaa_init() failed: %s\n", error);
   return -1;
 }
```

```
  setup_job_template(&jt);                                 (See 1)

  drmaa_run_job(job_id,
                DRMAA_JOBNAME_BUFFER-1,                    (See 2)
                jt,
                error,
                DRMAA_ERROR_STRING_BUFFER-1);

  fprintf(stderr,"Job successfully submitted ID: %s\n",job_id);

  result = drmaa_wait(job_id,
                      job_id_out,                              (See 3)
                      DRMAA_JOBNAME_BUFFER-1,
                      &stat,
                      DRMAA_TIMEOUT_WAIT_FOREVER,
                      &rusage,
                      error,
                      DRMAA_ERROR_STRING_BUFFER-1);

  if ( result != DRMAA_ERRNO_SUCCESS)
  {
      fprintf(stderr,"drmaa_wait() failed: %s\n", error);
      return -1;
  }

  drmaa_wexitstatus(&stat,stat,error,DRMAA_ERROR_STRING_BUFFER-1);
                                                               (See 4)
  fprintf(stderr,"Job finished with exit code %i, usage: %s\n",stat,job_id);

  while (drmaa_get_next_attr_value(rusage,
                    attr_value,                                (See 5)
                    DRMAA_ATTR_BUFFER-1)!= DRMAA_ERRNO_NO_MORE_ELEMENTS )
      fprintf(stderr,"\t%s\n",attr_value);

  drmaa_release_attr_values (rusage);                          (See 6)

  drmaa_delete_job_template(jt,
                            error,
                            DRMAA_ERROR_STRING_BUFFER-1);

  drmaa_exit (error, DRMAA_ERROR_STRING_BUFFER-1);

  return 0;
 }
```

1. Let us setup the job template with the previous function
2. Now we can submit a single job to the Grid, with this job template. The job id assigned by GridWay to our job is returned in the job_id variable. So let us print it and you can check the progress of your ls job with the gwps command.

3. This function will block (DRMAA_TIMEOUT_WAIT_FOREVER) until the job has finished or failed. There are two interesting values returned by the drmaa_wait() function: stat (the exit status of your job), and rusage (the resource usage made by the "ls" command, execution and transfer times). If the job is killed the drmaa_wait() will return with DRMAA_ERRNO_NO_RUSAGE.

4. DRMAA provides some functions to deal with the stat value. In this case we want to get the exit status of our job (you can also check if the job was signaled, the signal it received...)

5. Lets print the resource usage. As this is of type drmaa_attr_values_t we have to iterate over the list to get all the values. We finish when we find the DRMAA_ERRNO_NO_MORE_ELEMENTS return code.

6. Do not forget to free the DRMAA variables, like the job template, or the rusage list allocated by the drmaa_wait() function.

This example shows the same program using the DRMAA JAVA bindings.

```
try
{
 session.init(null);

 String id = session.runJob(jt);

 System.out.println("Job successfully submitted ID: " + id);

 JobInfo info = session.wait(id, Session.DRMAA_TIMEOUT_WAIT_FOREVER);

 System.out.println("Job usage:");

 Map rmap = info.getResourceUsage();
 Iterator r = rmap.keySet().iterator();

 while(r.hasNext())
 {
   String name2 = (String) r.next();
   String value = (String) rmap.get(name2);
   System.out.println(" " + name2 + "=" + value);
 }

 session.deleteJobTemplate(jt);

 session.exit();
 }
 catch (DrmaaException e)
 {
   e.printStackTrace();
 }
```

DRAFT

2.4. Job Status and Control

But you can do more things with a job than just submit it. The DRMAA standard allows you to control your jobs (kill, hold, release, stop,...) even they are not submitted within a DRMAA session. See the following example:

```
int main(int argc, char *argv[])
{
   char                    error[DRMAA_ERROR_STRING_BUFFER];
   int                     rc;
   drmaa_job_template_t *  jt;
   char                    job_id[DRMAA_JOBNAME_BUFFER];
   const char              *job_ids[2]={DRMAA_JOB_IDS_SESSION_ALL,NULL};
   int                     status;

   drmaa_init (NULL, error, DRMAA_ERROR_STRING_BUFFER-1);

   setup_job_template(&jt);

   drmaa_set_attribute(jt,                                 (See 1)
                    DRMAA_JS_STATE,
                    DRMAA_SUBMISSION_STATE_HOLD,
                    error,
                    DRMAA_ERROR_STRING_BUFFER-1);

   drmaa_run_job(job_id,
              DRMAA_JOBNAME_BUFFER,
              jt,
              error,
              DRMAA_ERROR_STRING_BUFFER-1);

   fprintf(stdout,"Your job has been submitted with id: %s\n", job_id);

   sleep(5);

   drmaa_job_ps(job_id, &status, error, DRMAA_ERROR_STRING_BUFFER);
                                                           (See 2)
   fprintf(stdout,"Job state is: %s\n",drmaa_gw_strstatus(status));

   sleep(1);

   fprintf(stdout,"Releasing the Job\n");

   rc = drmaa_control(job_id,                              (See 3)
                   DRMAA_CONTROL_RELEASE,
                   error,
                   DRMAA_ERROR_STRING_BUFFER-1);

   if ( rc != DRMAA_ERRNO_SUCCESS)
   {
       fprintf(stderr,"drmaa_control() failed: %s\n", error);
       return -1;
   }
```

```
    drmaa_job_ps(job_id, &status, error, DRMAA_ERROR_STRING_BUFFER);

    fprintf(stdout,"Job state is: %s\n",drmaa_gw_strstatus(status));

    fprintf(stdout,"Synchronizing with job...\n");

    rc = drmaa_synchronize(job_ids,                        (See 4)
                           DRMAA_TIMEOUT_WAIT_FOREVER,
                           0,
                           error,
                           DRMAA_ERROR_STRING_BUFFER-1);

    if ( rc != DRMAA_ERRNO_SUCCESS)
    {
        fprintf(stderr,"drmaa_synchronize failed: %s\n", error);
        return -1;
    }

    fprintf(stdout,"Killing the Job\n");

    drmaa_control(job_id,                                  (See 5)
                  DRMAA_CONTROL_TERMINATE,
                  error,
                  DRMAA_ERROR_STRING_BUFFER-1);

    if ( rc != DRMAA_ERRNO_SUCCESS)
    {
        fprintf(stderr,"drmaa_control() failed: %s\n", error);
        return -1;
    }

    fprintf(stdout,"Your job has been deleted\n");

    drmaa_delete_job_template(jt,
                              error,
                              DRMAA_ERROR_STRING_BUFFER-1);

    drmaa_exit (error,DRMAA_ERROR_STRING_BUFFER-1);

    return 0;
}
```

1. We are adding a new attribute to our job template the job submission state (DRMAA_JS_STATE). Our job will be held on submission
2. Let's check that the job is in the HOLD state. Note that we use a GridWay specific function to show the state drmaa_gw_strstatus(). This function is not part of the DRMAA standard.
3. OK, so we can now release our job, so it will begin its execution (i.e. will enter the QUEUED_ACTIVE state)
4. Now let us wait for this job, note that we are not using drmaa_wait() in this case. drmaa_synchronize() can be used to wait for a set of jobs. Its first argument is a NULL terminated array of job ids; we are using an special job

name: DRMAA_JOB_IDS_SESSION_ALL, to wait for all the jobs in submitted within your DRMAA session (other special job name is DRMAA_JOB_IDS_SESSION_ANY). The third argument (dispose) if equal to 1 will dispose (kill) the job. In this example we will do it by hand.

5. Kill the job with the TERMINATE control action.

Let see the same program in JAVA

```
try
{
  session.init(null);

  setup_job_template();

  String id = session.runJob(jt);

  System.out.println("Job successfully submitted ID: " + id);

  try
  {
    Thread.sleep(5 * 1000);
  }
  catch (InterruptedException e)
  { // Don't care }

  printJobStatus(session.getJobProgramStatus(id));

  try
  {
    Thread.sleep(1000);
  }
  catch (InterruptedException e)
  { // Don't care }

  System.out.println("Releasing the Job");

  session.control(id, Session.DRMAA_CONTROL_RELEASE);

  printJobStatus(session.getJobProgramStatus(id));

  System.out.println("Synchronizing with job...");

  session.synchronize(
        Collections.singletonList(Session.DRMAA_JOB_IDS_SESSION_ALL),
        Session.DRMAA_TIMEOUT_WAIT_FOREVER,
        false);

  System.out.println("Killing the Job");

  session.control(id, Session.DRMAA_CONTROL_TERMINATE);

  session.deleteJobTemplate(jt);
```

```
    session.exit();
  }
  catch (DrmaaException e)
  {
    e.printStackTrace();
  }
```

2.5. Array Jobs (bulk)

Bulk jobs are a direct way to express parametric computations. A bulk job is a set of independent (and very similar) tasks that use the same job template. You can use the DRMAA_PLACEHOLDER_INCR constat to assign different input/output files for each task. The DRMAA_PLACEHOLDER_INCR is a unique identifier of each job (task) in the bulk (array) job. In the GridWay DRMAA library it corresponds to the ${TASK_ID} parameter.

```
int main(int argc, char *argv[])
{
   char                    error[DRMAA_ERROR_STRING_BUFFER];
   int                     rc;
   int                     stat;

   drmaa_job_template_t *  jt;
   drmaa_attr_values_t  *  rusage;
   drmaa_job_ids_t      *  jobids;

   char                    value[DRMAA_ATTR_BUFFER];
   const char *            job_ids[2] ={DRMAA_JOB_IDS_SESSION_ALL,NULL};
   char                    job_id_out[DRMAA_JOBNAME_BUFFER];

   int                 rcj;

   drmaa_init (NULL, error, DRMAA_ERROR_STRING_BUFFER-1);

   setup_job_template(&jt);

   drmaa_set_attribute(jt,
                       DRMAA_OUTPUT_PATH,
                       "stdout."DRMAA_PLACEHOLDER_INCR,    (See 1)
                       error,
                       DRMAA_ERROR_STRING_BUFFER-1);

   rc = drmaa_run_bulk_jobs(&jobids,
                            jt,
                            0,        (See 2)
                            4,
                            1,
                            error,
                            DRMAA_ERROR_STRING_BUFFER-1);

   if ( rc != DRMAA_ERRNO_SUCCESS)
```

```
{
    fprintf(stderr,"drmaa_run_bulk_job() failed: %s\n", error);
    return -1;
}

fprintf(stderr,"Bulk job successfully submitted IDs are:\n");

do           (See 3)
{
    rc = drmaa_get_next_job_id(jobids, value, DRMAA_ATTR_BUFFER-1);

    if ( rc == DRMAA_ERRNO_SUCCESS )
        fprintf(stderr,"\t%s\n", value);

}while (rc != DRMAA_ERRNO_NO_MORE_ELEMENTS);

fprintf(stderr,"Waiting for bulk job to finish...\n");

drmaa_synchronize(job_ids,                                      (See 4)
                  DRMAA_TIMEOUT_WAIT_FOREVER,
                  0,
                  error,
                  DRMAA_ERROR_STRING_BUFFER-1);

fprintf(stderr,"All Jobs finished\n");

do
{
    rcj = drmaa_get_next_job_id(jobids, value, DRMAA_ATTR_BUFFER-1);

    if ( rcj == DRMAA_ERRNO_SUCCESS )
    {
      drmaa_wait(value,                                         (See 5)
                 job_id_out,
                 DRMAA_JOBNAME_BUFFER-1,
                 &stat,
                 DRMAA_TIMEOUT_WAIT_FOREVER,
                 &rusage,
                 error,
                 DRMAA_ERROR_STRING_BUFFER-1);

      drmaa_wexitstatus(&stat,stat,error,DRMAA_ERROR_STRING_BUFFER-1);

      fprintf(stderr,"Rusage for task %s (exit code %i)\n", value, stat);

      do
      {
        rc = drmaa_get_next_attr_value(rusage, value, DRMAA_ATTR_BUFFER-1);

        if ( rc == DRMAA_ERRNO_SUCCESS )
          fprintf(stderr,"\t%s\n", value);

      }while (rc != DRMAA_ERRNO_NO_MORE_ELEMENTS);
```

DRAFT

```
          drmaa_release_attr_values(rusage);
       }
    }while (rcj != DRMAA_ERRNO_NO_MORE_ELEMENTS);

    drmaa_release_job_ids(jobids);

    drmaa_delete_job_template(jt,
                              error,
                              DRMAA_ERROR_STRING_BUFFER-1);

    drmaa_exit (error,DRMAA_ERROR_STRING_BUFFER-1);

    return 0;
 }
```

1. The output file of each task in the bulk job will be "stdout."DRMAA_PLACEHOLDER_INCR. So the tasks will not over-write each others outputs.
2. We submit a bulk job with 5 tasks (0,1,2,3,4), note that we will get five different output files (stdout.0,stdout.1,...). The job ids assigned to each task are stored in the first argument of the function, a drmaa_job_ids_t list.
3. In order to get each job id you must use the drmaa_get_next_job_id() function. We iterate through the list until DRMAA_ERRNO_NO_MORE_ELEMENTS is returned.Do not forget to free the job id list when you no longer need it.
4. We wait for all the jobs in the array. Each you will be disposed with a call to the drmaa_wait() function.
5. We now use the drmaa_wait() function to get each task exit status and resource usage. As we have previously synchronize the bulk job this function will not block. drmaa_wait() will also remove each task from the GridWay system.

Finally a bulk job in JAVA.

```
 try
 {
  session.init(null);

  int start = 0;
  int end   = 4;
  int step  = 1;

  int i;

  String id;

  java.util.List          ids = session.runBulkJobs(jt, start, end, step);
  java.util.Iterator      iter = ids.iterator();

  System.out.println("Bulk job successfully submitted IDs are: ");

  while(iter.hasNext())
  {
```

```
    System.out.println("\t" + iter.next());
  }

  session.deleteJobTemplate(jt);

  session.synchronize(
            Collections.singletonLsectionist(Session.DRMAA_JOB_IDS_SESSION_ALL),
            Session.DRMAA_TIMEOUT_WAIT_FOREVER,
            false);

  for (int count = start; count <= end;count += step)
  {
    JobInfo info = session.wait(Session.DRMAA_JOB_IDS_SESSION_ANY,
                                Session.DRMAA_TIMEOUT_WAIT_FOREVER);

    System.out.println("Job usage:");
    Map rmap = info.getResourceUsage();
    Iterator r = rmap.keySet().iterator();

    while(r.hasNext())
    {
      String name2 = (String) r.next();
      String value = (String) rmap.get(name2);
      System.out.println(" " + name2 + "=" + value);
    }
  }

  session.exit();
 }
 catch (DrmaaException e)
 {
    System.out.println("Error:   " + e.getMessage());
 }
```

3. External Tutorials and Material

- GridWay Tutorial[1]
- Tutorial from drmaa.org[2]

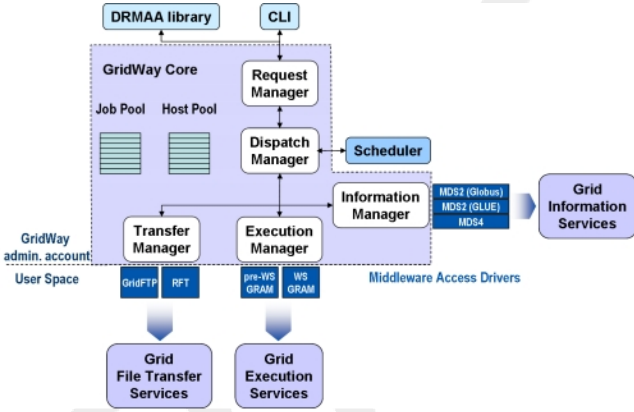
[1] http://www.gridway.org/documentation/tutorials.php
[2] https://forge.gridforum.org/sf/docman/do/listDocuments/projects.drmaa-wg/docman.root.ggf_13_drmaa_tutorial

Chapter 4. Architecture and design overview

1. Architecture

In GridWay 4.0.2, we introduced an architecture for the execution manager module based on a MAD (Middleware Access Driver) to interface Grid execution services. In this release we have taken advantage of this architecture to implement an information manager module with a MAD that interfaces Grid information services, and a transfer manager module with a MAD that interfaces Grid data services. Moreover, we have decoupled the scheduling process from the dispatch manager through the use of an external and selectable scheduler module.

Figure 4.1. Components of the GridWay Meta-scheduler.



GridWay architecture consists of the following components:

- *User Interface* provides the end user with DRM-like commands to submit, kill, migrate, monitor and synchronize jobs and includes DRMAA (Distributed Resource Management Application API) GGF (Global Grid Forum) standard support to develop distributed applications (C and JAVA bindings).
- *GridWay core* is responsible for job execution management and resource brokering providing advanced scheduling and job failure & recovery capabilities. The Dispatch Manager performs all submission stages and watches over the efficient execution of the job. The Information Manager, through its MADs (Middleware Access Driver), is responsible for host discovery and monitoring. The Execution Manager, through its MADs, is responsible job execution and management. The Transfer Manager, through its MADs, is responsible for file staging, remote working directory set-up and remote host clean-up.
- *Scheduler* takes scheduling decisions of jobs on available resources.
- *Information Manager MAD* interfaces to the monitoring and discovering services available in the Grid infrastructure.
- *Execution Manager MAD* interfaces to the Job Management Services available in the Grid resources.
- *Transfer Manager MAD* interfaces to the Data Management Services available in the Grid resources.

GWD communicates with MADs through the standard input/output streams. This makes easier the debugging process of new MADs.

2. Programming Model Overview

2.1. Programming Model

One of the most important aspects of Grid Computing is its potential ability to execute distributed communicating jobs. The Distributed Resource Management Application API (DRMAA) specification constitutes a homogeneous interface to different Distributed Resource Management Systems (DRMS) to handle job submission, monitoring and control, and retrieval of finished job status.

In this way, DRMAA could aid scientists and engineers to express their computational problems by providing a portable direct interface to DRMS. There are several projects underway to implement the DRMAA specification on different DRMS, like Sun Grid Engine (SGE), Condor or Torque.

GridWay provides a full-featured native implementation of the DRMAA standard to interface DRMS through the Globus Toolkit. The GridWay DRMAA library can successfully compile and execute the DRMAA test suite (version 1.4.0). Please check the GridWay DRMAA Reference Guide for a complete description of the DRMAA routines.

Although DRMAA could interface with DRMS at different levels, for example at the intranet level with SGE or Condor, with GridWay we will only consider its application at Grid level. In this way, the DRMS (GridWay in our case) will interact with the local job managers (Condor, PBS, SGE...) through the Grid middleware (Globus). This development and execution scheme with DRMAA, GridWay and Globus is depicted in the next figure.

Figure 4.2. Grid Development Model with DRMAA

2.2. Application Profiles

DRMAA allows scientists and engineers to express their computational problems in a Grid environment. The capture of the job exit code allows users to define complex jobs, where each depends on the output and exit code from the previous job. They may even involve branching, looping and spawning of subtasks, allowing the exploitation of the parallelism on the workflow of certain type of applications. Let's review some typical scientific application profiles that can benefit from DRMAA.

2.2.1. Embarrassingly distributed

Applications that can be obviously divided into a number of independent tasks. The application is asynchronous when it requires distinct instruction streams and so different execution times. A sample of this schema with its DRMAA implementation is showed in the following figure.

Figure 4.3. Embarrassingly Distributed Applications schema

```
rc = drmaa_init (contact, err);
// Execute initial job and wait for it
rc = drmaa_run_job(job_id, jt, err);
rc = drmaa_wait(job_id, &stat, timeout, rusage, err);
// Execute n jobs simultaneously and wait
rc = drmaa_run_bulk_jobs(job_ids,jt,1, JOB_NUM,1,err);
```

```
rc = drmaa_synchronize(job_ids, timeout, 1, err);
// Execute final job and wait for it
rc = drmaa_run_job(job_id, jt, err);
rc = drmaa_wait(job_id,&stat, timeout, rusage, err);
rc = drmaa_exit(err_diag);
```

2.2.2. Master-worker

A Master task assigns a description (input less) of the task to be performed by each Worker. Once all the Workers are completed, the Master task performs some computations in order to evaluate a stop criterion or to assign new tasks to more workers. Again, it could be synchronous or asynchronous. The following figure shows a example of Master-worker optimization loop and a DRMAA implementation sample.

Figure 4.4. Master-Worker Applications schema

```
rc = drmaa_init(contact, err_diag);
// Execute initial job and wait for it
rc = drmaa_run_job(job_id, jt, err_diag);
rc = drmaa_wait(job_id, &stat, timeout, rusage, err_diag);
while (exitstatus != 0)
{
 // Execute n Workers concurrently and wait
 rc = drmaa_run_bulk_jobs(job_ids, jt, 1, JOB_NUM, 1, err_diag);
 rc = drmaa_synchronize(job_ids, timeout, 1, err_diag);
 // Execute the Master, wait and get exit code
 rc = drmaa_run_job(job_id, jt, err_diag);
 rc = drmaa_wait(job_id, &stat, timeout, rusage, err_diag);
 rc = drmaa_wexitstatus(&exitstatus, stat, err_diag);
}
rc = drmaa_exit(err_diag);
```

DRAFT

Chapter 5. APIs

1. DRMAA bindings for C and Java

The following links reference the API for the C and Java bindings of GridWay's implementation of DRMAA:

- DRMAA C reference[1]
- DRMAA JAVA reference[2]

2. DRMAA bindings for scripting languages.

Most functions in DRMAA C library use reference parameters to get results when calling them. Using scripting languages makes this way of getting information unfeasible. For example python strings are immutable so it is not possible to fill an empty string with the information needed from the extension. The way the functions are called are the same as Perl and Python SGE DRMAA bindings, reference variables are omitted and what functions return is an array of result code, reference variables and error string. For example in C this call:

```
result=drmaa_version(&major,&minor,error,DRMAA_ERROR_STRING_BUFFER-1);
```

is translated to ruby as:

```
(result, major, minor, error)=drmaa_version
```

[1] http://www.gridway.org/documentation/stable/drmaa_c/
[2] http://www.gridway.org/documentation/stable/drmaa_java/

Table 5.1. Translation from C to Scripting language

C	Scripting Language
result=drmaa_get_next_attr_name(values, &value, value_len)	(result, value)=drmaa_get_next_attr_name(values)
result=drmaa_get_next_attr_value(values, &value, value_len)	(result, value)=drmaa_get_next_attr_value(values)
result=drmaa_get_next_job_id(values, &value, value_len)	(result, value)=drmaa_get_next_job_id(values)
result=drmaa_get_num_attr_names(values, &size)	(result, size)=drmaa_get_num_attr_names(values)
result=drmaa_get_num_attr_values(values, &size)	(result, size)=drmaa_get_num_attr_values(values)
result=drmaa_get_num_job_ids(values, &size)	(result, size)=drmaa_get_num_job_ids(values)
drmaa_release_attr_names(values)	drmaa_release_attr_names(values)
drmaa_release_attr_values(values)	drmaa_release_attr_values(values)
drmaa_release_job_ids(values)	drmaa_release_job_ids(values)
result=drmaa_init(contact, error, error_len)	(result, error)=drmaa_init(nil)
result=drmaa_exit(error, error_len)	(result, error)=drmaa_exit()
result=drmaa_allocate_job_template(&jt, error, error_len)	(result, jt, error)=drmaa_allocate_job_template()
result=drmaa_delete_job_template(jt, error, error_len)	(result, error)=drmaa_delete_job_template(jt)
result=drmaa_set_attribute(jt, name, value, error, error_len)	(result, error)=drmaa_set_attribute(jt, name, value)
result=drmaa_get_attribute(jt, name, &value, error, error_len)	(result, value, error)=drmaa_get_attribute(jt, name)
result=drmaa_set_vector_attribute(jt, name, value, error, error_len)	(result, error)=drmaa_set_vector_attribute(jt, name, value)
result=drmaa_get_vector_attribute(jt, name, &values, error, error_len)	(result, values, error)=drmaa_get_vector_attribute(jt, name)
result=drmaa_get_attribute_names(&values, error, error_len)	(result, values, error)=drmaa_get_attribute_names()
result=drmaa_get_vector_attribute_names(&values, error, error_len)	(result, values, error)=drmaa_get_vector_attribute_names()
result=drmaa_run_job(job_id, job_id_len, jt, error, error_len)	(result, job_id, error)=drmaa_run_job(jt)
result=drmaa_run_bulk_jobs(&jobids, jt, start, end, incr, error, error_len)	(result, jobids, error)=drmaa_run_bulk_jobs(jt, start, end, incr)
result=drmaa_control(jobid, action, error, error_len)	(result, error)=drmaa_control(jobid, action)
result=drmaa_job_ps(job_id, &remote_ps, error, error_len)	(result, remote_ps, error)=drmaa_job_ps(job_id)
result=drmaa_synchronize(job_ids, timeout, dispose, error, error_len)	(result, error)=drmaa_synchronize(job_ids, timeout, dispose)
result=drmaa_wait(job_id, job_id_out, job_id_out_len, &stat, timeout, &rusage, error, error_len)	(result, job_id_out, stat, rusage, error)=drmaa_wait(job_id, timeout)
result=drmaa_wifexited(&exited, stat, error, error_len)	(result, exited, error)=drmaa_wifexited(stat)

C	Scripting Language
result=drmaa_wexitstatus(&exit_status, stat, error, error_len)	(result, exit_status, error)=drmaa_wexitstatus(stat)
result=drmaa_wifsignaled(&signaled, stat, error, error_len)	(result, signaled, error)=drmaa_wifsignaled(stat)
result=drmaa_wtermsig(signal, signal_len, stat, error, error_len)	(result, signal, error)=drmaa_wtermsig(stat)
result=drmaa_wcoredump(&core_dumped, stat, error, error_len)	(result, core_dumped, error)=drmaa_wcoredump(stat)
result=drmaa_wifaborted(&aborted, stat, error, error_len)	(result, aborted, error)=drmaa_wifaborted(stat)
error_string=drmaa_strerror(drmaa_errno)	error_string=drmaa_strerror(drmaa_errno)
result=drmaa_get_contact(contact, contact_len, error, error_len)	(result, contact, error)=drmaa_get_contact()
result=drmaa_version(&major, &minor, error, error_len)	(result, major, minor, error)=drmaa_version()
result=drmaa_get_DRM_system(drm_system, drm_system_len, error, error_len)	(result, drm_system, error)=drmaa_get_DRM_system()
result=drmaa_get_DRMAA_implementation(drmaa_impl, drmaa_impl_len, error, error_len)	(result, drmaa_impl, error)=drmaa_get_DRMAA_implementation()
str_status=drmaa_gw_strstatus(drmaa_state)	str_status=drmaa_gw_strstatus(drmaa_state)

Note

In **drmaa_init** call we have to pass a **NULL** argument as GridWay DRMAA library requires it. Here as an example I used **nil** as it is the ruby object describing **NULL** but in perl you have to use **undef** and in python **None**.

DRAFT

Chapter 6. Non-WSDL protocols

1. Information manager MAD

In order to provide an abstraction with the monitoring and discovery middleware layer (or Grid Information System), GridWay uses a Middleware Access Driver (MAD) module to discover and monitor hosts. This module provides basic operations with the monitoring and discovery middleware.

The format to send a request to the Execution MAD, through its standard input, is:

```
OPERATION HID HOST -
```

Where:

- OPERATION: Can be one of the following:
  - INIT: Initializes the MAD.
  - DISCOVER: Discovers hosts.
  - MONITOR: Monitors a host.
  - FINALIZE: Finalizes the MAD.
- HID: If the operation is MONITOR, it is a host identifier, chosen by GridWay. Otherwise it is ignored.
- HOST: If the operation is MONITOR it specifies the host to monitor. Otherwise it is ignored.

On the other side, the format to receive a response from the MAD, through its standard output, is:

```
OPERATION HID RESULT INFO
```

Where:

- OPERATION: Is the operation specified in the request that originated the response.
- HID: It is the host identifier, as provided in the submission request.
- RESULT: It is the result of the operation. Could be SUCCESS or FAILURE.
- INFO: If RESULT is FAILURE, it contains the cause of failure. Otherwise, if OPERATION is DISCOVER, it contains a list of discovered host, or if OPERATION is MONITOR, it contains a list of host attributes.

DRAFT

Table 6.1. Attributes that should be defined by the Information MADs.

HOSTNAME	FQDN (Fully Qualified Domain Name) of the execution host (e.g. "hydrus.dacya.ucm.es")
ARCH	Architecture of the execution host (e.g. "i686", "alpha")
OS_NAME	Operating System name of the execution host (e.g. "Linux", "SL")
OS_VERSION	Operating System version of the execution host (e.g. "2.6.9-1.66", "3")
CPU_MODEL	CPU model of the execution host (e.g. "Intel(R) Pentium(R) 4 CPU 2", "PIV")
CPU_MHZ	CPU speed in MHz of the execution host
CPU_FREE	Percentage of free CPU of the execution host
CPU_SMP	CPU SMP size of the execution host
NODECOUNT	Total number of nodes of the execution host
SIZE_MEM_MB	Total memory size in MB of the execution host
FREE_MEM_MB	Free memory in MB of the execution hosts
SIZE_DISK_MB	Total disk space in MB of the execution hosts
FREE_DISK_MB	Free disk space in MB of the execution hosts
LRMS_NAME	Name of local DRM system (job manager) for execution, usually not fork (e.g. "jobmanager-pbs", "Pbs", "jobmanager-sge", "SGE")
LRMS_TYPE	Type of local DRM system for execution (e.g. "PBS", "SGE")
QUEUE_NAME[i]	Name of queue i (e.g. "default", "short", "dteam")
QUEUE_NODECOUNT[i]	Total node count of queue i
QUEUE_FREENODECOUNT[i]	Free node count of queue i
QUEUE_MAXTIME[i]	Maximum wall time of jobs in queue i
QUEUE_MAXCPUTIME[i]	Maximum CPU time of jobs in queue i
QUEUE_MAXCOUNT[i]	Maximum count of jobs that can be submitted in one request to queue i
QUEUE_MAXRUNNINGJOBS[i]	Maximum number of running jobs in queue i
QUEUE_MAXJOBSINQUEUE[i]	Maximum number of queued jobs in queue i
QUEUE_DISPATCHTYPE[i]	Dispatch type of queue i (e.g. "batch", "inmediate")
QUEUE_PRIORITY[i]	Priority of queue i
QUEUE_STATUS[i]	Status of queue i (e.g. "active", "production")

The information drivers interface to the grid information services to collect the resource attributes. These attributes can be used by the end-user to set requirement and rank expressions (job template), for filtering, prioritizing and selecting the candidate hosts. GridWay can simultaneously use as many Information drivers as needed. For example, GridWay allows you to simultaneously use MDS2 and MDS4 services, so you can also use resources from different Grids at the same time. Drivers for MDS 2 and MDS 4 provide the variables described in Table 2-1. However, the information manager is able to receive from the driver other parameters. The GridWay team has used other information parameters that could be very important to improve application efficiency (HTC apps) and for job migration: BANDWIDTH, LATENCY, SPEC_INT, SPEC_FLOAT...

2. Execution manager MAD

In order to provide an abstraction with the resource management middleware layer, GridWay uses a Middleware Access Driver (MAD) module to submit, control and monitor the execution of jobs. This module provides basic operations with the resource management middleware.

The format to send a request to the Execution MAD, through its standard input, is:

```
OPERATION JID HOST/JM RSL
```

Where:

- OPERATION: Can be one of the following:
  - INIT: Initializes the MAD.
  - SUBMIT: Submits a job.
  - POLL: Polls a job to obtain its state.
  - CANCEL: Cancels a job.
  - FINALIZE: Finalizes the MAD.
- JID: Is a job identifier, chosen by GridWay.
- HOST: If the operation is SUBMIT, it specifies the resource contact to submit the job. Otherwise it is ignored.
- JM: If the operation is SUBMIT, it specifies the job manager to submit the job. Otherwise it is ignored.
- RSL: If the operation is SUBMIT, it specifies the resource specification to submit the job. Otherwise it is ignored.

On the other side, the format to receive a response from the MAD, through its standard output, is:

```
OPERATION JID RESULT INFO
```

Where:

- OPERATION: Is the operation specified in the request that originated the response or CALLBACK, in the case of an asynchronous notification of a state change.
- JID: It is the job identifier, as provided in the submission request.
- RESULT: It is the result of the operation. Could be SUCCESS or FAILURE.
- INFO: If RESULT is FAILURE, it contains the cause of failure. Otherwise, if OPERATION is POLL or CALLBACK, it contains the state of the job.

3. Transfer manager MAD

In order to provide an abstraction with the file transfer management middleware layer, GridWay uses a Middleware Access Driver (MAD) module to transfer job files. This module provides basic operations with the file transfer middleware.

The format to send a request to the Transfer MAD, through its standard input, is:

DRAFT

```
OPERATION JID TID EXE_MODE SRC_URL DST_URL
```

Where:

- OPERATION: Can be one of the following:
  - INIT: Initializes the MAD, JID should be max number of jobs.
  - START: Init transfer associated with job JID
  - END: Finish transfer associated with job JID
  - MKDIR: Creates directory SRC_URL
  - RMDIR: Removes directory SRC_URL
  - CP: start a copy of SRC_URL to DST_URL, with identification TID, and associated with job JID.
  - FINALIZE: Finalizes the MAD.
- JID: Is a job identifier, chosen by GridWay.
- TID: Transfer identifier, only relevant for command CP.
- EXE_MODE: If equal to 'X' file will be given execution permissions, only relevant for command CP.

On the other side, the format to receive a response from the MAD, through its standard output, is:

```
OPERATION JID TID RESULT INFO
```

Where:

- OPERATION: Is the operation specified in the request that originated the response or CALLBACK, in the case of an asynchronous notification of a state change.
- JID: It is the job identifier, as provided in the START request.
- TID: It is the transfer identifier, as provided in the CP request.
- RESULT: It is the result of the operation. Could be SUCCESS or FAILURE.
- INFO: If RESULT is FAILURE, it contains the cause of failure.

4. Dispatch manager Scheduler

In order to decouple the scheduling process, GridWay uses a Scheduler module to schedule jobs.

The format to send a scheduling request to the Scheduler, through its standard input, is:

```
SCHEDULE - - -
```

On the other side, the format to receive a response from the Scheduler, through its standard output, is:

```
OPERATION JID RESULT INFO
```

Where:

- OPERATION: Is the operation requested to the Dispatch Manager. The Dispatch Manager only supports the SCHEDULE_JOB operation.
- JID: It is a job identifier.
- RESULT: It is the result of the operation. Could be SUCCESS or FAILURE.
- INFO: If RESULT is FAILURE, it contains the cause of failure. Otherwise, if OPERATION is SCHEDULE_JOB it contains a resource specification in the form HID:QNAME:RANK, where:
  - HID: It is the host identifier, as provided by **gwhosts** command.
  - QNAME: It is the queue name.
  - RANK: It is the rank of the host.

GridWay includes a scheduler template (`gw_scheduler_skel.c`) to develop custom schedulers. As an example a Round-Robin/Flooding scheduler can be found in the GridWay distribution (`gw_flood_scheduler.c`, in `$GW_LOCATION/src/sched/`). This is a very simple scheduling algorithm, which maximizes the number of jobs submitted to the Grid.

DRAFT

GridWay Commands

DRAFT

Name

Job and Array Job submission Command -- job submission utility for the GridWay system

```
gwsubmit <-t template> [-n tasks] [-h] [-v] [-o] [-s start] [-i increment] [-d
"id1 id2 ..."]
```

Description

Submit a job or an array job (if the number of tasks is defined) to gwd

Command options

-h	Prints help.
-t <template>	The template file describing the job.
-n <tasks>	Submit an array job with the given number of tasks. All the jobs in the array will use the same template.
-s <start>	Start value for custom param in array jobs. Default 0.
-i <increment>	Increment value for custom param in array jobs. Each task has associated the value PARAM=start + increment * TASK_ID, and MAX_PARM = start+increment*(tasks-1). Default 1.
-d <"id1 id2...">	Job dependencies. Submit the job on hold state, and release it once jobs with id1,id2,.. have successfully finished.
-v	Print to stdout the job ids returned by gwd.
-o	Hold job on submission.
-p <priority>	Initial priority for the job.

DRAFT

Name

DAG Job submission Command -- dag job submission utility for the GridWay system

```
gwdag [-h] [-d] <DAG description file>
```

Description

Submit a dag job to gwd

Command options

-h Prints help.

-d Writes to STDOUT a DOT description for the specified DAG job.

DRAFT

Name

Job Monitoring Command -- report a snapshot of the current jobs

```
gwps [-h] [-u user] [-r host] [-A AID] [-s job_state] [-o output_format] [-c
delay] [-n] [job_id]
```

Description

Prints information about all the jobs in the GridWay system (default)

Command options

-h Prints help.

-u user Monitor only jobs owned by user.

-r host Monitor only jobs executed in host.

-A AID Monitor only jobs part of the array AID.

-s job_state Monitor only jobs in states matching that of job_state.

-o output_format Formats output information, allowing the selection of which fields to display.

-c <delay> This will cause gwps to print job information every <delay> seconds continuously (similar to top command).

-n Do not print the header.

job_id Only monitor this job_id.

DRAFT

Output field description

Table 3. Field options

FIELD NAME	FIELD OP-TION	DESCRIPTION
USER	u	owner of this job
JID	J	job unique identification assigned by the Gridway system
AID	i	array unique identification, only relevant for array jobs
TID	i	task identification, ranges from 0 to TOTAL_TASKS -1, only relevant for array jobs
FP	p	fixed priority of the job
TYPE	y	type of job (simple, multiple or mpi)
NP	n	number of processors
DM	s	dispatch Manager state, one of: pend, hold, prol, prew, wrap, epil, canl, stop, migr, done, fail
EM	e	execution Manager state (Globus state): pend, susp, actv, fail, done
RWS	f	flags: R — times this job has been restarted W — number of processes waiting for this job S — re-schedule flag
START	t|T	the time the job entered the system
END	t|T	the time the job reached a final state (fail or done)
EXEC	t|T	total execution time, includes suspension time in the remote queue system
XFER	t|T	total file transfer time, includes stage-in and stage-out phases
EXIT	x	job exit code
TEM-PLATE	j	filename of the job template used for this job
HOST	h	hostname where the job is being executed

DRAFT

Name

Job History Command -- shows history of a job

```
gwhistory [-h] [-n] <job_id>
```

Description

Prints information about the execution history of a job

Command options

-h Prints help.

-n Do not print the header lines

job_id Job identification as provided by gwps.

Output field description

Table 4. Field information

NAME	DESCRIPTION
HID	host unique identification assigned by the Gridway system.
START	the time the job start its execution on this host.
END	the time the job left this host, because it finished or it was migrated.
PROLOG	total prolog (file stage-in phase) time.
WRAPPER	total wrapper (execution phase) time.
EPILOG	total epilog (file stage-out phase) time.
MIGR	total migration time.
REASON	the reason why the job left this host.
QUEUE	name of the queue.
HOST	FQDN of the host.

DRAFT

Name

Host Monitoring Command -- shows hosts information

```
gwhost [-h] [-c delay] [-nf] [-m job_id] [host_id]
```

Description

Prints information about all the hosts in the GridWay system (default)

Command options

-h Prints help.

-c <delay> This will cause gwhost to print job information every <delay> seconds continuously (similar to top command)

-n Do not print the header.

-f Full format.

-m <job_id> Prints hosts matching the requirements of a given job.

host_id Only monitor this host_id, also prints queue information.

Output field description

Table 5. Field information

FIELD	DESCRIPTION
HID	host unique identification assigned by the Gridway system
PRIO	priority assigned to the host
OS	operating system
ARCH	architecture
MHZ	CPU speed in MHZ
%CPU	free CPU ratio
MEM(F/T)	system memory: F = Free, T = Total
DISK(F/T)	secondary storage: F = Free, T = Total
N(U/F/T)	number of slots: U = used by GridWay, F = free, T = total
LRMS	local resource management system, the jobmanager name
HOSTNAME	FQDN of this host

DRAFT

Table 6. Queue field information

FIELD	DESCRIPTION
QUEUENAME	name of this queue
SL(F/T)	slots: F = Free, T = Total
WALLT	queue wall time
CPUT	queue cpu time
COUNT	queue count number
MAXR	max. running jobs
MAXQ	max. queued jobs
STATUS	queue status
DISPATCH	queue dispatch type
PRIORITY	queue priority

DRAFT

Name

Job Control Command -- controls job execution

```
gwkill [-h] [-a] [-k | -t | -o | -s | -r | -l | -9] <job_id [job_id2 ...] | -A
array_id>
```

Description

Sends a signal to a job or array job

Command options

-h Prints help.

-a Asynchronous signal, only relevant for KILL and STOP.

-k Kill (default, if no signal specified).

-t Stop job.

-r Resume job.

-o Hold job.

-l Release job.

-s Re-schedule job.

-9 Hard kill, removes the job from the system without synchronizing remote job execution or cleaning remote host.

job_id [job_id2 ...] Job identification as provided by gwps. You can specify a blank space separated list of job ids.

-A <array_id> Array identification as provided by gwps.

DRAFT

Name

Job Synchronization Command -- synchronize a job

`gwwait [-h] [-a] [-v] [-k] <job_id ...| -A array_id>`

Description

Waits for a job or array job

Command options

-h Prints help.

-a Any, returns when the first job of the list or array finishes.

-v Prints job exit code.

-k Keep jobs, they remain in fail or done states in the GridWay system. By default, jobs are killed and their resources freed.

-A <array_id> Array identification as provided by gwps.

job_id ... Job ids list (blank space separated).

DRAFT

Name

User Monitoring Command -- monitors users in GridWay

`gwuser [-h] [-n]`

Description

Prints information about users in the GridWay system

Command options

-h Prints help.

-n Do not print the header.

Output field description

Table 7. Field information

FIELD	DESCRIPTION
UID	user unique identification assigned by the Gridway system
NAME	name of this user
JOBS	number of Jobs in the GridWay system
RUN	number of running jobs
IDLE	idle time, (time with JOBS = 0)
EM	execution manager drivers loaded for this user
TM	transfer manager drivers loaded for this user
PID	process identification of driver processes

DRAFT

Name

Accounting Command -- prints accounting information

```
gwacct [-h] [-n] [<-d n | -w n | -m n | -t s>] <-u user|-r host>
```

Description

Prints usage statistics per user or resource. Note: accounting statistics are updated once a job is killed.

Command options

-h Prints help.

-n Do not print the header.

<-d n | -w n | -m n | -t s> Take into account jobs submitted after certain date, specified in number of days (-d), weeks (-w), months (-m) or an epoch (-t).

-u user Print usage statistics for user.

-r hostname Print usage statistics for host.

Output field description

Table 8. Field information

FIELD	DESCRIPTION
HOST/USER	host/user usage summary for this user/host
XFR	total transfer time on this host (for this user)
EXE	total execution time on this host (for this user), without suspension time
SUSP	total suspension (queue) time on this host (for this user)
TOTS	total executions on this host (for this user). Termination reasons: • SUCC success • ERR error • KILL kill • USER user requested • SUSP suspension timeout • DISC discovery timeout • SELF self migration • PERF performance degradation • S/R stop/resume

DRAFT

Name

JSDL To GridWay Job Template Parser Command -- parser to translate JSDL file into GridWay Job Template file

```
jsdl2gw [-h] input_jsdl [output_gwjt]
```

Description

Converts a jsdl document into a gridway job template. If no output file is defined, it defaults to the standard output. This enables the use of pipes with gwsubmit in the following fashion:

```
jsdl2gw jsdl-job.xml | gwsubmit
```

Command options

-h Prints help.

input_jsdl Reads the jsdl document from the input_jsdl

output_gwjt Stores the GridWay Job Template specification in the output_gwjt.jt file

DRAFT

Chapter 7. Debugging

We'll begin with debugging the underlying Java WS Core and then discuss debugging in GridWay in particular. For information about sys admin logging, see Chapter 7, Debugging in the Admin Guide.

1. Debugging in Java WS Core

As GridWay relies on Globus services, it is assumed that a Globus grid infrastructure has been installed and configured. Failures related to Globus services (e.g. GRAM or MDS) can be debugged as described in Chapter 10, Debugging.

1.1. Development Logging in Java WS Core

The following information applies to Java WS Core and those services built on it.

Logging in the Java WS Core is based on the Jakarta Commons Logging[1] API. Commons Logging provides a consistent interface for instrumenting source code while at the same time allowing the user to plug-in a different logging implementation. Currently we use Log4j[2] as a logging implementation. Log4j uses a separate configuration file to configure itself. Please see Log4j documentation for details on the configuration file format[3].

1.1.1. Configuring server side developer logs

Server side logging can be configured in `$GLOBUS_LOCATION/container-log4j.properties`, when the container is stand alone container. For tomcat level logging, refer to Logging for Tomcat[4], . The logger `log4j.appender.A1` is used for developer logging and by default writes output to the system output. By default it is set for all warnings in the Globus Toolkit package to be displayed.

Additional logging can be enabled for a package by adding a new line to the configuration file. Example:

```
#for debug level logging from org.globus.package.FooClass
log4j.category.org.globus.package.name.FooClass=DEBUG
#for warnings from org.some.warn.package
log4j.category.org.some.warn.package=WARN
```

1.1.2. Configuring client side developer logs

Client side logging can be configured in `$GLOBUS_LOCATION/log4j.properties`. The logger `log4j.appender.A1` is used for developer logging and by default writes output to the system output. By default it is set for all warnings in the Globus Toolkit package to be displayed.

2. Debugging in GridWay

Due to GridWay's architecture, mainly its MADs components, debugging it is not a trivial task. The most obvious way to see what is going on is to monitor what happens in the GridWay log files. Here are the files to look into in case of trouble:

[1] http://jakarta.apache.org/commons/logging/
[2] http://logging.apache.org/log4j/
[3] http://logging.apache.org/log4j/docs/api/org/apache/log4j/PropertyConfigurator.html#doConfigure(java.lang.String, org.apache.log4j.spi.LoggerRepository)
[4] http://tomcat.apache.org/tomcat-5.5-doc/logging.html

- **$GW_LOCATION/var/gwd.log**: This is the general log file, where the gwd daemon logs whatever the Resource, Dispatch, Transfer, Execution, Information managers inform it about.
- **$GW_LOCATION/var/sched.log**: The scheduler is a separate process that communicates with the daemon using the standard input/output. It writes log information to this file.
- **$GW_LOCATION/var/<job_id>/job.log**: Each job has its own log file, with information regarding its context (input/output files, MADs, resource) and it's life cycle. In this folder also reside the job.template, the job.env with the environment variable and the standard output and error of the wrapper script (stdout.wrapper and stderr.wrapper)

In order to get the maximum amount of debug information in the gwd.log file (especially more information about what the MADs are doing), you should compile GridWay with the following configure option:

```
./configure --enable-debug
```

If there is a problem with GridWay that makes any MAD crash, it will be useful to have a coredump. To tell the MADs that they should write to a coredump file when they crash, use the following environment variable before you execute your first job:

```
export MADDEBUG=yes
```

Sometimes it is the daemon (the gwd process) that crashes. In order to obtain a coredump of the daemon, run the following command before executing the daemon:

```
ulimit -c unlimited
```

The coredump file will be written to the $GW_LOCATION/var directory, with a filename corresponding to the process PID, that is,

```
$GW_LOCATION/var/core.<process_pid>
```

If you cannot figure out what is wrong, you can always use the mailing list gridway-user[5] to get support. Please provide a detailed explanation of your problem so the community can reproduce it and give advice. Also, send along:

- $GW_LOCATION/var/gwd.log
- $GW_LOCATION/var/sched.log
- $GW_LOCATION/var/<job_id>/{job.log,stderr.wrapper}: If relevant. The stderr.wrapper file is specially useful for debugging; it shows step by step the wrapper script being executed.

DRAFT

[5] http://dev.globus.org/wiki/GridWay#Mailing_Lists

Chapter 8. Troubleshooting

For a list of common errors in GT, see Error Codes.

1. Errors

Table 8.1. Gridway Errors

Error Code	Definition	Possible Solutions
`Lock file exists`	Another GWD may be running.	Be sure that no other GWD is running, then remove the lock file and try again.
`Error in MAD initialization`	There may be problems with the proxy certificate, bin directory, or the executable name of a MAD may not be in the correct location.	Check that you have generated a valid proxy (for example with the **grid-proxy-info** command). Also, check that the directory `$GW_LOCATION/bin` is in your path, and the executable name of all the MADs is defined in `gwd.conf`.
`Could not connect to gwd`	GridWay may not be running or there may be something wrong with the connection.	Be sure that GWD is running; for example: `pgrep -l gwd` If it is running, check that you can connect to GWD; for example: ``telnet `cat $GW_LOCATION/var/gwd.port` ``

2. Debugging

For more detailed developer debugging information, see Chapter 7, Debugging. For information about sys admin logging, see Chapter 7, Debugging.

Chapter 9. GT 4.2.0 Samples for GridWay

1. DRMAA examples

The following links contain samples demonstrating GridWay's DRMAA bindings:

- DRMAA C Howtos[1]
- DRMAA Java Howtos[2]
- DRMAA Python Howtos[3]
- DRMAA Ruby Howtos[4]
- DRMAA Perl Howtos[5]

DRAFT

[1] http://www.gridway.org/files/examples/drmaaC.tgz
[2] http://www.gridway.org/files/examples/drmaaJ.tgz
[3] http://www.gridway.org/files/examples/drmaa_python.tar.gz
[4] http://www.gridway.org/files/examples/drmaa_ruby.tar.gz
[5] http://www.gridway.org/files/examples/drmaa_perl.tar.gz