DRMAA Interface Specification

by
DRMAA Working Group participants

Document maintained by Hrabri.Rajic@intel.com

History:

Date	Document	Comment
Apr 29, 2002	Sched-drmaa-1.2	This version combines sched-drmaa-1.0 and 1.1 documents and DRMAA working group discussion till April 16.
May 8, 2002	Sched-drmaa-1.3	Addition of “job categories” and “explicit native resource specification” sections contributed by Andreas Haas. Minor polishes in response to DRMAA telecom discussion of Apr 30, ‘02.
May 13, 2002	Sched-drmaa-1.4	First pass at expanding Section 3 based on May 13 DRMAA video conference meeting. [Blame Nitzberg for errors.]

1 Introduction

Distributed Resource Management Application API (DRMAA) hides the differences of the Distributed Resource Management Systems (DRMSs) and provides an API intended for distributed application developers or ISVs. It is one of the DRMAA working group goals to target a very broad audience and consequently require an easy learning curve for the use of the specified API. The mandate of the DRMAA working group is to produce DRMAA specification 1.0.

Language issues

In is our position that the API should be implemented in multiple languages, C/C++ being the primary choice. The secondary choices are scripting languages, Perl and Python. Perl is especially heavily used in the biotech arena where there is great need for numerous parametric calculations.

It is possible to design an Interface Definition Language that will effectively resolve the issue of one interface serving multiple languages. While this is a viable approach, we feel that it will slow the progress on the implementation side significantly. Another viable approach would be to design a protocol instead of the API. Both of these alternatives should be considered in more detail when the time comes to address the long term comprehensive solutions.

Library Issues

An ideal library would have paths to handle all DRMSs and versions to be linked statically or dynamically. This is not something that will be feasible. A real possibility is a situation where one vendor implements multiple, but not all DRMSs. The packaging could come as one library, where a DRMS is selected at run time setting an environmental variable for the desired DRMS, or as one DRMS link per library. The latter approach is advocated by the authors. In this setup the shared library is selected at run time by end users.

It is expected that the developers will be linking the library from serial and multithreaded codes. The library should be thread safe.

It is expected that the debugging of the distributed programs will be more challenging than single machine versions. We advocate providing production and debugging version of the library.

Library should provide the DRMAA API version number to the external programs (such as SCCS’s “what” and RCS’s “ident”) and to the distributed applications programmatically.

<u>*User program and DRMAA interaction*</u>

All of the DRMSs are asynchronous in nature. They notify the end user of the status of a finished job via e-mail, which as an only option is not acceptable to the users of DRMAA library. We propose to deal with the asynchrony similarly to Unix and Windows process interfaces, by blocking on the wait call for a specific job request or possibly to all of them in the same process. This is in contrast to Globus GRAM interface that is modeled on the reactive mode of execution. The support for reactive mode is to be addressed in the future DRMAA versions, because more and more programs come with graphic user interface these days.

The rest of the document is presented as follows. The Chapter 2 deals with the API design issues. Chapter 3. presents a Draft API specification.

2 API Design Issues

Developers have been using Unix system, fork/exec, popen, and the wait interfaces for years to spawn additional processes and wait for the end of their execution to get their exit codes. Windows has equivalent utilities like CreateProcess and WaitForSingleObject. DRMAA provides its own set of interfaces that are OS neutral. It borrows Unix process API simplicity and tries to be consistent with libc interfaces.

2.1 Basic Guidelines

Even though the API should be self-contained, it is not always possible to consolidate all variations of end user and DRMS interactions under the API. For this reason, we advocate that the developers should provide a way for the end user to specify DRMS particular options. The primary mean to achieve this is through use of "job categories".
Additionally, there might be a need for possibility that the DRMS specific options are specified at run time as command line options. The end user could loose portability this way, which is a small price to pay to be able to run the application in uncommon configurations. Besides that, DRMAA providers and ISVs are the ones that target multiple DRMSs; the end user does that at a much lesser degree.

The API centers around job_id parameter that is passed back by the DRMS upon job submission. Job_id is used for all the job control and monitoring purposes. [An additional similar parameter, job_name, that is found in all DRMS implementations is part of the job submission interface. Job_name could be used by the developer and/or internally by the implementation to group the jobs for easier user classification and tracking. This parameter could be a key to achieve scalability for DRMAA implementations, especially since DRMS user jobs could be running concurrently with those of the other DRMS users.]

There are few guidelines that were used in designing the uniform API:

- The API calling sequences should be simple and the API set small.
- The routine names should convey the semantic of the routine.
- The set should be as convenient as possible, even with the risk of being forced to emulate some functionality if missing from a DRMS.
- All job manipulation is available without explicit job iterating.
- The server names are hidden, the DRMS is a black box.
- The end user does not need to interact with the DRMS since the DRMS environment is specified with job categories by the application administrator. If he has to he could specify native resource options parameter.
- The API should be extensible.

2.2 DRMAA Distributed Application Environment

DRMAA specification 1.0 does not have explicit file staging mechanisms. File staging is enabled by setting job template attributes.

2.2.1 Job categories

The DRMAA interface specification should allow ISVs to write DRM-enabled applications even though the properties of a concrete DRM installation, in particular the configuration of the DRM system, cannot be known in advance.

Experiences made with integrations based on DRM CLI show that even when the same ISV application is run as a job with the same DRM system the site specific policies in effect differ widely. These policies are typically about questions like

- what resources are to be used by the job
- preferences where to run the job
- how prior the job should be treated by the DRM scheduler compared to other jobs

For supporting the variety of policies, job specific requests expressed by DRM submit options are very common in the DRM product space.

Despite of these differences between two sites with the "same" job passed to the DRM system the application actually does not change when seen from the perspective of the ISV. Also for the end user who just wants a job to be started nothing changes due to different policies. This is an indication that there must be possibility for hiding these site-specific differences behind the DRMAA interface.

The job "categories concept" is the approach the DRMAA working group recommends for encapsulating site-specific details and completely hiding these details from applications making use of the DRMAA interface. The core of the idea is to have these application only supplying a string attribute specifying a job category, i.e. a name specifying what kind of application that is to be dispatched by the DRMS. The category name can be used by the DRMAA library to determine site specific resource and functional requirements of jobs in this category. Such requirements need to be configurable by the site operating a DRM system and deploying an ISV application on top of it.

An example can help to illustrate this idea:

- At site A rendering application X is used in a heterogeneous clustered environment which is managed by a DRMS. Since application X is only available at a subset of these machines the administrator sets up the DRMS in a way requiring from the end-users to put a **-l X=true** into their submit command line.

- At site B the same application is used in a homogenous clustered environment with rendering application X supported at all machines managed by the DRMS. However since X jobs do compete with applications Y sharing the same resources and X applications are to be treated with higher priority than Y jobs end-users need to put a **-p 1023** into their submit command line for raising the dispatch priority.

An integration based on categories will allow to submit X jobs through the DRMAA interface in compliance with the policies of both sites A and B without the need to know about these policies. The ISV does this by specifying "X" as the category used for X rendering jobs submitted through the DRMAA interface and by mentioning this in the "DRM integration" section of the X rendering software documentation.

The administrators at the sites A and B site read the documentation or installation instructions about the "X" DRMAA category. The documentation of their DRMS contains directions about the category support of their DRMAA interface implementation. From this documentation they learn how to configure their DRMS in a way that "-l X=true" is used for "X" jobs at site A while "-p 1023" is used at site B for those jobs.

As far as the DRMAA interface specification is concerned only a standardized mechanism for specifying the category is required. The mechanism for associating the policy related portion of the submit command line to the job is to be delivered by each DRMAA implementation. A standardization of this mechanism is beyond the DRMAA standardization effort, because it is too much related to the administrative interface and it is anticipated that for different DRMS different mechanisms will be appropriate.

2.2.2 Native resource specification

The benefit of the categories concept from the last chapter is that it provides a means for completely hiding site-specific policy details to be considered with a DRMAA job submission for a whole class of jobs. The drawback however of this concept is that it **requires** one job category to be maintained for each policy to be used.

To allow the DRMAA interface to be used also for submission of jobs where job-individual policy specification is required "native resource specification" is supported. Native resource specification can be used without the requirement to maintain job categories. Instead of specifying a category name and having the DRMAA implementation associate the corresponding job submit options, the use of native resource specification will allow directly specifying these submit options.

An example can help to illustrate this idea:

> In order to implement the example from section 2.2.1 via native resource specifications, the native option string "-l X=true" had to be passed directly to the DRMAA interface while "-p 1023" had to be used at site B.

As far as the DRMAA interface specification is concerned the native resource specification is an opaque string and interpreted by each DRMAA library. It is possible to use job categories and native resource specification with the same job submission for policy specification. It is assumed that in this case the DRMAA library is capable of joining the outcome of the two policy sources in a reasonable way.

2.3 Interface Routines General Description

The routines are naturally grouped in four categories: init/exit, job submission, job monitoring and control, and auxiliary or system routines like trace file specification and error message routines. All the routines have a prefix "drmaa_".

All of the routines should return an error code upon exit. A possible exception is an auxiliary error message routine that could be modeled after the standard libc strerror routine. In connection to the error routine there should be mechanism for getting DRMAA equivalent of libc errno value. In libc errno is a macro that expands to a modifiable lvalue, such as a dereferenced function pointer to address libc use in reentrant mode.

2.3.1 Init and exit routines

The calling sequence of the init routine should allow all of the considered DRMSs to be properly initialized, either by interfacing to the batch queue commands or to the DRMS API. Likewise, the exit routine should require parameters that will permit proper DRMS disengagement.

2.3.2 Job template and job submission routines

The job submission routines come in two versions. There is one version for submitting individual jobs and one version for submitting bulk jobs. The remote jobs and their attributes are specified with a job template opaque parameter. The job attributes are divided in three groups:

- Core/base or implicit. These have provided setter and getter routines.
- Extended or reserved. These attributes are needed to address desired functionality for particular DRMSs. They are set by using a generic setter routine where the name of the attribute is passed as an extra parameter. All DRMAA libraries need to provide this routine with the default of ignoring the request. Some of these attributes could move to the core/base set.

- Native. These attributes are particular to one or possibly few DRMSs. They are specified via job category mechanism or via the generic setter routine.

The core/base or implicit attributes are:

- Remote command to execute, including the input parameters.
- Job state at submission (suspended/on hold or active).
- Job environment.
- Job working directory.
- Real or wall clock time limit.
- Standard input, output, and error streams.
- E-mail to report the job completion and status.

The extended or reserved attributes are:

- Job execution mode, synchronous or asynchronous.
- Input/output files to be staged and a parameter denoting shared or distributed file system. DRMAA specification 1.0 assumes shared file system. This is to be used with care.
- Job name to be used for the job submission. (Alphanumeric and _ character allowed.)
- Time of execution.

2.3.3 Job monitoring and controlling routines

Job monitoring and controlling API group needs to handle:

- job stopping, resuming, and killing
- waiting for the remote job till the end of its execution
- checking the exit code of the finished remote job
- checking the remote job status
- waiting for all the jobs to finish execution (this is a useful synchronization mechanism)

The Unix and Windows signals are replaced with the job control routines that have counterparts in DRMSs. The only nontraditional feature is the passing of a NULL job_id to indicate operations on all job_ids in the current process.

The remote job could be in following states:

- queued
- system suspended
- user suspended
- running
- finished (un)successfully

To this list we need to add a possibility of DRMAA library not being able to determine the status of the remote job.

2.3.4 Auxiliary routines

The auxiliary routines are needed for execution tracing and error monitoring. The tracing is especially useful for the situations when there is multiple processes spawned few levels deep. The error codes routines and variable drmaa_errno are libc equivalents. drmaa_errno is a macro such as a function reference for reentrant DRMAA library implementation.

The next Chapter contains the an example of an API as specified here.

3 API Specification

The API is preceded by the common constants that are used in the course of the distributed program implementation. For convenience, the API is divided in its four logical sections: init/exit, job submission, job monitoring and control, and auxiliary routines.

To prevent excessive number of job template setter/getter attribute routines, an alternative approach is given in section 3.2.

Disclaimer #1: The code is used here for illustrative purposes. It is not meant to be an example of a full solution at this stage.
Disclaimer #2: The routine names are tentative.

3.1 C/C++ API, explicit job template setter/getter attribute routines

```
/* ---------- Initialization & Exit Routines ---------- */

drmaa_init(contact)
   IN  contact        /* contact information for DRM system (string) */

  Initialize DRMAA API library and create a new DRMAA Session.  'Contact'
  is an implementation dependent string which may be used to specify
  which DRM system to use.  This routine must be called before any
  other DRMAA calls, except for drmaa_version().


drmaa_exit( )

  Disengage from DRMAA library and let the DRMAA library clean up any
  objects it created.  Once this routine is called, no more DRMAA API
  calls can be made.  This routine ends this DRMAA Session, but does
  not effect any jobs (e.g., queued and running jobs remain queued
  and running).


drmaa_version(major, minor)
   OUT major    /* major version number (non-negative integer) */
   OUT minor    /* minor version number (non-negative integer) */

  Returns the major and minor version numbers of the DRMAA library;
  for DRMAA 1.0, 'major' is 1 and 'minor' is 0.


/* ----------  Job Template and Job Submission Routines ---------- */


drmaa_allocate_job_template( )
   RETURNS       /* new job template (opaque handle) */

  Allocate a new job template.


drmaa_delete_job_template(jt)
   INOUT jt     /* job template (opaque handle) */
```

Deallocate a job template. This routine has no effect on jobs.

```
drmaa_set_attribute(jt, name, value)
   INOUT jt     /* job template (opaque handle) */
   IN    name   /* attribute name (string) */
   IN    value  /* attribute value (string) */
```

Adds ('name', 'value') pair to list of attributes in job template 'jt'.

The following are reserved attribute names available in all implementations of DRMAA v1.0 (and their respective meanings):

- remote command to execute, including the input parameters
- job state at submission (suspended, on hold, active)
- job environment
- job working directory
- wall clock time limit
- job category

The following are reserved attribute names available which are not required to be implemented by a conforming DRMAA v1.0 implementation. For attributes that are implemented, the meanings are required to be as follows:

- standard input, output, and error streams
- e-mail to report he job completion and status
- input/output files to be staged and
  a parameter denoting shared or distributed file system
- job name to be used for the job submission ([A-Za-z0-9_]+)
- start after

NOTE: We may break each of the above out into a separate set/get routines for improved type checking (and ease of passing complex parameters). They are listed in the form above as a short-hand. As with all naming issues, it is proposed to postpone this decision.

```
drmaa_get_attribute(jt, name, value)
   IN    jt     /* job template (opaque handle) */
   IN    name   /* attribute name (string) */
   RETURNS      /* attribute value (string) */
```

If 'name' is an existing attribute name in the job template 'jt', then the value of 'name' is returned; otherwise, NULL is returned.

ISSUE: Do we want to allow one to get the attributes associated with a job category? What about just getting a list of available native attributes?

```
drmaa_run_job(job_id, jt, synchronous)
   OUT job_id      /* job identifier (string) */
   IN  jt          /* job template (opaque handle) */
   IN  synchronous /* block until action completes (boolean) */
```

Submit a job with attributes defined in the job template 'jt'.
The job identifier 'job_id' is a printable, NULL terminated string,

identical to that returned by the underlying DRM system.
If 'synchronous' is TRUE, this routine will not return
until the job has been submitted (or an error occurs).

```
drmaa_run_bulk_job(job_id, jt, njobs, synchronous)
    OUT job_ids     /* job identifiers (array of strings) */
    IN  jt          /* job template (opaque handle) */
    IN  njobs       /* number of jobs (non-negative integer) */
    IN  synchronous /* block until action completes (boolean) */
```

Submit a set of 'njobs' jobs, each with attributes defined in the
job template 'jt'. The job identifiers 'job_ids' are all printable,
NULL terminated strings, identical to those returned by the underlying
DRM system. If 'synchronous' is TRUE, this routine will not return
until all the jobs have been submitted (or an error occurs).

ISSUE: How do you pass different parameters to each job, or
otherwise differentiate them?

```
/* ---------- Job Control Routines ---------- */
```

ISSUE: Do we want to add a timeout argument to the routines
that might block?

```
drmaa_getpid_status(job_id, exit_code)
    IN  job_id      /* job identifier (string) */
    OUT exit_code   /* exit code of job (integer) */
```

Wait for the job identified by 'job_id' to exit. The exit status
of the job is returned in 'exit_code'; note that this status is
machine dependent.

```
drmaa_control(job_id, action, synchronous)
    IN job_id       /* job identifier (string) */
    IN action       /* control action (const) */
    IN synchronous  /* block until action completes (boolean) */
```

Start, stop, restart, or kill the job identified by 'job_id'.
If 'job_id' is DRMAA_JOB_ID_ALL, then this routine
acts on all jobs *submitted* during this DRMAA session.
The legal values for 'action' and their meanings are:

```
  DRMAA_CONTROL_SUSPEND:    stop the job,
  DRMAA_CONTROL_RESUME:     (re)start the job,
  DRMAA_CONTROL_HOLD:       put the job on-hold,
  DRMAA_CONTROL_RELEASE:    release the hold on the job, and
  DRMAA_CONTROL_TERMINATE:  kill the job.
```

If 'action' is DRMAA_CONTROL_RESUME and 'synchronous' is TRUE,
this routine will not return until the requested action has been
completed (or an error occurs); otherwise, 'synchronous' has
no effect.

```
drmaa_synchronize(job_ids)
```

```
    IN  job_ids       /* job identifiers (array of strings) */

  Wait until all jobs specified by 'job_ids' have finished
  execution.  If 'job_ids' is DRMAA_JOB_IDS_ALL, then this routine
  waits for all jobs *submitted* during this DRMAA Session.


drmaa_wait(job_id, exit_code, options, rusage)
    IN  job_id      /* job identifier (string) */
    OUT exit_code   /* exit code of job (integer) */
    IN  options     /* options (integer) */
    OUT rusage      /* resource usage (???) */

  Wait for the job identified by 'job_id' to exit.  The exit status
  of the job is returned in 'exit_code'; note that this status is
  machine dependent.  Legal values for 'options', and their
  meanings are:
    ???.
  The resource usage of the job is returned in 'rusage'; note that
  this value is also machine dependent.


drmaa_job_ps( char *job_id, int *remote_ps );
    IN  job_id      /* job identifier (string) */
    OUT remote_ps   /* program status (constant) */

  Get the program status of the job identified by 'job_id'.
  The possible values returned in 'remote_ps' and their meanings are:
    DRMAA_PS_UNDETERMINED:     process status cannot be determined,
    DRMAA_PS_QUEUED:           job is queued,
    DRMAA_PS_SYSTEM_SUSPENDED: job is system suspended,
    DRMAA_PS_USER_SUSPENDED:   job is user suspended,
    DRMAA_PS_RUNNING:          job is running,
    DRMAA_PS_DONE:             job finished normally, and
    DRMAA_PS_FAILED:           job finished, but failed.


/* ----------  Auxiliary Routines ---------- */
```

PROPOSAL: Postpone discussions of error handling and tracing until
we get everything else in better shape.

In C, we would probably want to return 0 on success and an errno on failure, except when we're returning a structure pointer (when we can't return an errno). Of course, in C++, we might want to use real exception handling...

```
drmaa_set_trace_file(file_name)
    IN  file_name   /* File name (string) */

  Specify a file for tracing.  By default, all tracing information
  is written to stderr.


drmaa_trace_text(text)
```

```
    IN  text   /* Message to be logged in trace file (string) */
```

Write 'text' into the trace file.

```
drmaa_perror( char *text);
    IN  text   /* Error message to be logged in trace file (string) */
```

Record the error message 'text' in the trace file.

```
char *drmaa_strerror (int error);
    IN errno   /* Error number (integer) */
    RETURNS    /* Readable text version of error (constant string) */
```

Get the error message text associated for the error number*/

3.2 Enumerated parameter job template setter/getter attribute routine

The idea here is to prevent having proliferation of number of setter and getter routines. Instead, drmaa_basic_attributes_t enumeration has all the basic/core attributes listed. drmaa_set_attrName and drmaa_get_attrName are only one affected form the previous section. drmaa_basic_attributes_t enumeration and their replacements are defined as:

```
typedef enum drmaa_basic_attributes {
    command = 0,
    email,
    stdinput,
    stdoutput,
    stderror,
    time_limit,
    initial_job_state,
} drmaa_basic_attributes_t;


/* set an attrName attribute */
int  drmaa_set_attribute( drmaa_job_template *jt,  drmaa_basic_attributes_t dbat,  char *value );

/* get attribute value for attribute attrName */
char*  drmaa_get_attribute( drmaa_job_template *jt,  drmaa_basic_attributes_t dbat);
```