DRMAA Proposal

Submitted by
Hrabri Rajic and Robert Kuhn
KSL, Intel Corporation
Hrabri.Rajic@intel.com

December 19, 2001

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. 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.

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 will 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.

User program and DRMAA interaction

All of the DRMSs are asynchronous in nature. They notify the end user of the status of a finished job via e-mail, which might not be acceptable to the users of DRMAA library. To be fair here, at least one of the DRMSs job submission mechanisms could execute synchronously and return only after the remote job completion.

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. This is in contrast to Globus GRAM interface that is modeled on the reactive mode of execution. As a matter of fact the GRAM model should be seriously revisited in the future DRMAA versions, because more and more programs come with graphic user interface these days.

Library functionality

One way to constrain the feature set in DRMAA is that only the DRMS products demonstrated features are acceptable. This test should be applied whenever the feature is not absolutely essential for DRMAA.

Comment:

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

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 DRMS specific options are specified only at run time as command line options. The end user could loose portability this way, but few of them are using multiple DRMSs on their sites. 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 competing with those of other 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 servers names are hidden, the DRMS is a black box.
- The end user interact with the DRMS via native_resource_options parameter.
- Simple types are used for scripting languages API export, the alternative would be to have a “true” C interface and an alternative one that is used for scripting languages.

The routines are grouped in four categories: init/exit, job submit, 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 return the error code upon exit, but drmaa_strerror. that mirrors the standard libc strerror routine is also provided. The parameters types are restricted to integers and C strings for an easy API exports to the scripting languages. This does not come without danger since it compromises the parameter safety and could lead to programmer mistakes.

The calling sequence of the drmaa_init routine allows all of the considered DRMSs to initialize, either by interfacing to the batch queue commands or to the DRMS API.

The job submission routines come in two versions. The first version, routine drmaa_submit, is suited for submitting a set of tasks that do not have everything in common or are not intended for numerous submissions. The second version, pair of routines drmaa_define_task and drmaa_submit_param, is suited for a large number of identical tasks that take different parameters at each invocation. In the latter

approach a task template is constructed first and the submit command is given different parameters at each invocations. Both versions allow a developer to specify:

- Remote command and pre-exec command to execute.
- Mode of job execution, synchronous or asynchronous.
- Common shared or distributed file system.
- The files to be staged.
- Manipulation of standard input output, and error streams.
- Native DRMS options to pass thru.
- Job name to be used for the job submission.

The drmaa_define_task routine shares many parameters of the drmaa_submit routine. The task templates that are constructed via this routine are submitted with the routine drmaa_submit_param that shares the rest of the drmaa_submit routine.

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 untraditional feature is the passing of a NULL job_id to indicate operations on all job_ids.

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. There are extensions for use with the interpretive languages. drmaa_errno is a function call for reentrant library implementation. This is handled correctly for the developer at the compile time.

The next Chapter contains the API as specified here.

3 API Specification

Two versions of the API are given in this Chapter, preceded by the common constants that could be used in the course of the distributed program implementation. The first version is a true C/C++ interface, while the second one is a more verbose version suited for interfacing to scripting languages. The API is presented as if was taken from an include file.

Disclaimer: *The code is used here for illustrative purposes. It is not meant to be an example of a full solution at this stage.*

3.1 C/C++ API

```
/* DRMAA constants */
#define DRMAA_JOB_SIZE                  128  /* the size of the job_id buffer */

/* remote job status constants */
/* constant that indicates that remote job status cannot be determined */
#define DRMAA_PS_UNDETERMINED          -6
/* constant that indicates that remote job is queued */
#define DRMAA_PS_QUEUED                -4
/* constant that indicates that remote job is system suspended */
#define DRMAA_PS_SYSTEM_SUSPENDED  -3
/* constant that indicates that remote job is user suspended */
#define DRMAA_PS_USER_SUSPENDED       -2
```

```
/* constant that indicates that remote job is running */
#define DRMAA_PS_RUNNING                    -1
/* constant that indicates that remote job finished normally */
#define DRMAA_PS_DONE                       0
/* constant that indicates that remote job finished, but failed */
#define DRMAA_PS_FAILED                     1


typedef struct drmaa_job_define {
   int      shared;             /* are shared disk directories used for remote job execution */
   char   files;                /*  files to be transferred to and back from the remote servers */
   char   *pre_command; ; /* check if command can execute, not all of the DRMS have a notion of
                                  pre_command */
   char   command;              /* command to execute on the remote server */
   char   *s_stdin;
   char   *s_stdout;
   char   *s_stderr;
} drmaa_job_define_t

typedef struct drmaa_job_submit_opt {
   int     synch;               /* return after executing remote job or immediately */
   int     hold;                /* hold or suspend the remote job at submission time */
   char  native_resource_options;  /* DRMS native options to be at the remote job submission time */
} drmaa_job_submit_opt_t;

typedef struct drmaa_job_desc {
   drmaa_job_define_t          *job_definition;
   drmaa_job_submit_opt_t  *job_submit_parameters;
} drmaa_job_desc_t;


/* ----------------------------------------  init/exit routines ----------------------------------*/

/* init the application distributor */
int drmaa_init(char *hostname, int hostport, char *program_name);

/* Disengage from the transport layer and clean up the objects we created */
int drmaa_exit( void );


/* ----------------------------------  job submission routines --------------------------------*/

/* construct a task, populate the parameters, and submit the remote job */
int drmaa_submit (char *job_id, char *job_name,  drmaa_job_desc_t *job_desc);

/* define a task template for later execution */
int drmaa_define_task (char *task_name, char *files,
                       drmaa_job_submit_opt_t *job_submit_ops);


/* populate the task template and queue the job for execution*/
int drmaa_submit_param   (char *job_id, char *job_name,
                       char *task_name, drmaa_job_submit_opt_t *job_desc,
                       char *para_names, char *param_values);


/* ----------------------------------------  job control routines ----------------------------------*/
```

```
/* (re)start the job, all if NULL, and return when done if synchronous */
int drmaa_start( char *job_id, int synchronous);

/* stop/pause the job, all if NULL */
int drmaa_stop( char *job_id );

/* kill the job, all if NULL */
int drmaa_kill( char *job_id );

/* synchronize or block till all of the submitted jobs have finished execution*/
int drmaa_synchronize ( void );

/* get program status of the remote job */
int drmaa_job_ps( char *job_id, int *remote_ps );

/* wait for the remote job to finish and get the remote job exit code */
int drmaa_waitpid( char *job_id, int *exit_code);

/* get the exit code of the finished remote job */
int drmaa_getpid_status( char *job_id, int *exit_code);


/* ---------------------------------------  auxiliary routines ---------------------------------*/

/* specify a file for tracing */
int drmaa_set_trace_file(char *file_name);

/* pass the text to output in the trace file or stderr by default*/
int drmaa_trace_text( char *text );

/* record the application distributor error in the trace file, or stderr by default */
int drmaa_perror( char *text);

/* get the error message for the error number*/
char *drmaa_strerror (int error);
```

Note 1: Not all of the DRMS have a notion of pre_command, a test for determining command execution.

3.2 Simple types API

This interface is suited for scripting language binding

```
/* DRMAA constants */
#define DRMAA_JOB_SIZE                  128  /* the size of the job_id buffer */

/* remote job status constants */
/* constant that indicates that remote job status cannot be determined */
#define DRMAA_PS_UNDETERMINED          -6
/* constant that indicates that remote job is queued */
#define DRMAA_PS_QUEUED                -4
/* constant that indicates that remote job is system suspended */
#define DRMAA_PS_SYSTEM_SUSPENDED  -3
/* constant that indicates that remote job is user suspended */
#define DRMAA_PS_USER_SUSPENDED       -2
/* constant that indicates that remote job is running */
#define DRMAA_PS_RUNNING               -1
```

```
/* constant that indicates that remote job finished normally */
#define DRMAA_PS_DONE                    0
/* constant that indicates that remote job finished, but failed */
#define DRMAA_PS_FAILED                  1


/* ----------------------------------------  init/exit routines ----------------------------------*/

/* init the application distributor */
int drmaa_init(char *hostname, int hostport, char *program_name);

/* Disengage from the transport layer and clean up the objects we created */
int drmaa_exit( void );


/* ----------------------------------- job submission routines[1] ---------------------------------*/

/* construct a task, populate the parameters, and submit the remote job */
int drmaa_submit (char *job_id, char *job_name,
                  int shared, int synchronous, int hold,
                  char* native_resource_options, char *files,
                  char *pre_command, char *command,
                  char *s_stdin, char *s_stdout, char *s_stderr);

/* define a task template for later execution */
int drmaa_define_task (char *task_name, int shared, char *files,
                  char *pre_command, char *command,
                  char *s_stdin, char *s_stdout, char *s_stderr);


/* populate the task template and queue the job for execution*/
int drmaa_submit_param   (char *job_id, char *job_name,
                  char *task_name,  int synchronous, int hold,
                  char *native_resource_options, char *para_names,
                  char *param_values);


/* ----------------------------------------  job control routines ----------------------------------*/

/* (re)start the job, all if NULL, and return when done if synchronous */
int drmaa_start( char *job_id, int synchronous);

/* stop/pause the job, all if NULL */
int drmaa_stop( char *job_id );

/* kill the job, all if NULL */
int drmaa_kill( char *job_id );

/* synchronize or block till all of the submitted jobs have finished execution*/
int drmaa_synchronize ( void );

/* get program status of the remote job */
int drmaa_job_ps( char *job_id, int *remote_ps );
```

```
/* wait for the remote job to finish and get the remote job exit code */
int drmaa_waitpid( char *job_id, int *exit_code);

/* get the exit code of the finished remote job */
int drmaa_getpid_status( char *job_id, int *exit_code);



/* ---------------------------------------  auxiliary routines ----------------------------------*/


/* specify a file for tracing */
int drmaa_set_trace_file(char *file_name);

/* pass the text to output in the trace file or stderr by default*/
int drmaa_trace_text( char *text );

/* get the drmaa_errno ( used from scripting languages) */
int drmaa_errno_get( void );

/* record the application distributor error in the trace file, or stderr by default */
int drmaa_perror( char *text);

/* get the error message for the error number*/
char * drmaa_strerror (int error);
```

Note 1: Not all of the DRMS have a notion of pre_command, a test for determining command execution.