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.]
May 14, 2002	Sched-drmaa-1.5	Second pass based on May 14 DRMAA video conference meeting.
June 30, 2002	Sched-drmaa-1.6	Pre-GGF5 status progress incorporated. Notes and DRMAA status presentation put into appendices.

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 by providing an easy to use programming model. 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. The interfaces are described using an IDL like language.

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 authors advocate the latter approach. 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.

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 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.contains the DRMAA 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 could specify native resource options parameter if he/she needs to interact with the DRMS.

- The API should be extensible in a sense that future implementation are backward compatible with earlier ones.

2.2 DRMAA Distributed Application Environment

DRMAA specification 1.0 does not have explicit file staging mechanisms. Setting implementation specific job template attributes could enable file staging, provided the implementation supports it.

2.2.1 Building Portals

The nature of the DRMAA implementatio, as a shared library, makes it a good candidate for inclusion in a Web Server to support a Web Portal to a DRMS. There are several options for doing this:

- Linked by a collection of CGI scripts that are referenced by resident Web Pages.
- Linked in a Web Server as a separate module.
- Linked as unmanaged DLL from .NET Web Forms.
- Built as a Perl module that is
  - included in mod_perl module.
  - accessed from Perl CGI scripts

Some of these options are suitable only for a particular Web Server.

The questions about maintaining a state, security, and authentication and authorization, require that DRMAA implementation is viewed as just one component of a DRM Web Portal. Clearly, this is beyond the scope of the current document and DRMAA Charter.

2.2.2 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 applications 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.

NOTE: Categories require a separate specification, a task that is beyond current DRMAA Charter.

2.2.3 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 five categories: init/exit, job template handling, 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. DRMAA needs an equivalent of libc errno value for internal failures. 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 routines

NOTE: This sections requires complete update.

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 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.

TODO

2.3.4 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.

2.3.5 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 an example of an API as specified here.

3 API Specification

For convenience, the API is divided in its five logical sections: init/exit, job template handling, job submission, job monitoring and control, and auxiliary routines.

Disclaimer #1: IDL like language is used here to avoid questions about allocation/deallocation issues. We plan to fully specify C/C++ bindings to insure binary compatibility.

Disclaimer #2: The routine names are tentative.

3.1 C/C++ DRMAA API

The NOTES and ISSUES that are provided in blue italic font are to be discussed at GGF5.

/* ---------- Major Assumptions/Restrictions ---------- */

- `Callbacks (asynchronous notification) -- Polling only in v1.0`
- `No explicit file staging.`
- `JobID Uniqueness -- "As unique as the underlying DRM makes them"`

/* ---------- 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().

  ADD: If 'contact' is NULL, the default DRM system will be used.


drmaa_exit( )

  Disengage from DRMAA library and allow the DRMAA library to perform
  any necessary internal clean up.
  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.

DRM_engine drmaa_get_DRM_engine( )
      Output (string) is implementation dependent and could contain the DRM
      engine and the implementation vendor as its parts.
```

General Notes:
NOTE: There is only one DRMAA session open at the time. Another session could be opened only after the current one is closed. Nesting of sessions is not allowed. It is expected that the DRMAA library will free all the session resources, although this is not guaranteed, so old session resources are not to be used later.

ISSUE 1: Extensibility issue with regard to backward compatibility. If DRMAA 2.0 allows multiple connections that means drmaa_session parameter is part of the interface. Visit this after all of the issues have been resolved.

/* ---------- Job Template 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'.
  Only non-vector attributes may be passed.

drmaa_set_vector_attribute(jt, name, value)
    INOUT jt     /* job template (opaque handle) */
    IN    name   /* attribute name (string) */
    IN    values /* vector of attribute value (string vector) */

  Adds ('name', 'values') pair to list of vector attributes in job template
'jt'.
  Only vector attributes may be passed.

The following are reserved attribute names available in all
implementations of DRMAA v1.0 (and their respective meanings).
Vector attributes are marked with a 'V':
      remote command to execute
    V input parameters
         These parameters are passed as arguments to the job.
         The attribute name is "argv".
      job state at submission (on hold, active)
    V job environment
         This environment is set for the job. The attribute name
         is "envv".
      job working directory
      wall clock time limit
      job category
      e-mail to report the job completion and status
         If e-mail is to be sent to the submitter by the DRM system
         this e-mail adress is used. The attribute name is "email".
      standard input, output, and error streams (staging cannot be assumed)

  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:
    input/output files (including stdin/out/err) 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 job not later then

  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.

  ISSUE: We need to set up a mechanism to discuss (and flesh-out)
         these attributes & add/delete attributes.   We really need
         a formal proposal to discuss ...


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

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

drmaa_get_vector_attribute(jt, name, value)
    IN    jt       /* job template (opaque handle) */
    IN    name     /* attribute name (string) */
    RETURNS values /* vector of attribute value (string vector) */

  If 'name' is an existing vector attribute name in the job template 'jt',
  then the values of 'name' are returned; otherwise, NULL is returned.
  ISSUE 1: Do we want to allow one to get the attributes associated
         with native attributes?


  ISSUE 2: Enumerated parameter job template setter/getter attribute routine
       The idea here is to prevent having proliferation of number of setter and
       getter routines while ensuring type safety.   drmaa_basic_attributes_t
       enumeration will have basic/core attributes listed.   drmaa_set_attribute
       and drmaa_get_attribute functions are affected.  Vector setter/getter
       routines could be handled similarly. drmaa_basic_attributes_t enumeration
       and their replacements could be defined as:

       typedef enum drmaa_basic_attributes {
           command = 0,
           env,
           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);



/* ----------  Job Submission Routines ---------- */

drmaa_run_job(job_id, jt)
    OUT job_id      /* job identifier (string) */
    IN  jt          /* job template (opaque handle) */

  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.


drmaa_run_bulk_jobs(job_ids, pjt, start, end, incr)
    OUT job_ids     /* job identifiers (array of strings) */
    IN  pjt         /* parameterized job template (opaque handle) */
    IN  start       /* beginning index ( unsigned integer?)*/
    IN  end         /* ending index ( unsigned integer?) */
    IN  incr        /* loop increment (integer)*/

  Submit a set of parametric jobs, dependent on the implied loop index, each
  with attributes defined in the parameterized job template 'pjt'.
  The job identifiers 'job_ids' are all printable,
  NULL terminated strings, identical to those returned by the underlying
  DRM system.  Nonnegative loop bounds are suggested to avoid file names
  that start with minus sign like command line options.


  The special index placeholder is a DRMAA defined string
       drmaa_incr_ph /* == $incr_pl$ */
  that is used to construct parametric job templates.
  For example:
       drmaa_set_attribute(pjt, "stderr", drmaa_incr_ph + ".err" ); /*
  C++/java string syntax used */
```

NOTE: Job template and parameterized job template could be the same structure. Type-wise, they do not differ at all, the parameterized job templates need to be parsed and substituted before they could be used.

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

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

  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.

This routine returns once the action has been acknowledged by
the DRM system, but does not necessarily wait until the action
has been completed.
```

ISSUE: You can now do an action for which there is no corresponding way to wait (or tell) if it completes in a nice way.

```
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, stat, timeout, rusage)
    IN  job_id      /* job identifier (string) */
    OUT stat        /* status code of job (integer) */
    IN  timeout     /* how long we block in this call (long) */
    OUT rusage      /* resource usage */

This routine waits for a job with job_id to finish execution. This routine is
modeled on wait3 POSIX routine.
```

ISSUE 1: Revisit 'timeout' parameter. Defined values for "wait forever" and "don't wait at all"

ISSUE 2: What about "reaping" the job_id (so DRMAA_JOB_IDS_ALL) doesn't include "exited" jobs... If we need this, we should add a separate DRMAA routine to reap.
If someone wants this, they should make a proposal.

```
drmaa_wifexited(OUT exited, IN stat)
     Can optionally evaluate into 'exited' a zero value if status was
     returned for a job that terminated not normally. A non-zero 'exited'
     value indicates more detailed diagnosis can be provided by means of
     drmaa_wifsignaled(), drmaa_wtermsig() and drmaa_wcoredump().

drmaa_wexitstatus(OUT exit_code, IN stat)
     If the OUT parameter 'exited' of drmaa_wifexited() is non-zero,
     this function evaluates into 'exit_code' the exit code that the
     job passed to _exit() (see exit(2)) or exit(3C), or the value that
     the child process returned from main.

drmaa_wifsignaled(OUT signaled, IN stat)
     Evaluates into 'signaled' a non-zero value if status was returned
     for a job that terminated due to the receipt of a signal.

drmaa_wtermsig(OUT signal, IN stat)
     If the OUT parameter 'signaled' of drmaa_wifsignaled(stat) is
     non-zero, this function evaluates into signal the number of the
     signal that caused the termination of the job.

drmaa_wcoredump(OUT core_dumped, IN stat)
     If the OUT parameter 'signaled' of drmaa_wifsignaled(stat) is
     non-zero, this function evaluates into 'core_dumped' a non-zero value
     if a core image of the terminated job was created.
```

The 'stat' drmaa_wait parameter is used in a series of functions, defined above, for providing more detailed information about job termination if available. An analogous set of macros is defined in POSIX for analyzing wait3(2) OUT parameter 'stat'. The misleading upper-case function names reminding to macros are changed to lower-case names.

ISSUE: Implementation needs to know on which architecture the remote job was running, Win32 or Unix/Linux, to correctly provide more information. Could rusage OUT parameter from drmaa_wait routine help here?

```
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.
```

General Notes:
NOTE: The users could use jobIDs from old DRMAA sessions to query jobs. The implementation could be able to resolve the requests only if the original DRM server is connected to in both sessions. Nothing is guaranteed. The queried jobIDs, even if able to resolve, will not become part of the current session.

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

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

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

NOTE 1:
In C, we would probably want to return 0 on success and an errno on failure (this is different from libc and open to debate), 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...

NOTE 2: From libc docs:
The external variable errno is used to hold implementation-defined error codes from library routines. All errno's are positive. Library routines should never clear errno.

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

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


error_string drmaa_strerror (errno);
    IN errno   /* Errno number (integer) */
    RETURNS    /* Readable text version of errno (constant string) */

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


contact drmaa_get_contact();
       OUT  contact        /* Current contact information for DRM system
                              (string) */
```

NOTE: Is this function needed if detailed error reporting is in place that provides this information (assumes a need only when drmaa_init fails)?

APPENDIX A   DRMAA C/C++ API Notes

/* ---------- Major Assumptions/Restrictions ---------- */

```
Callbacks (asynchronous notification) -- Polling only in v1.0
No explicit file staging.
JobID Uniqueness -- "As unique as the underlying DRM makes them"
```

/* ---------- 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().

  ADD: If 'contact' is NULL, the default DRM system will be used.

  STRAW VOTE: 7/1/1 (Yes/No/Abstain) in favor of keeping contact'.

  PROPOSAL: Allow multiple calls to drmaa_init() and return a
            handle that is used in all calls to specify which
            DRM to use.
  STRAW VOTE: 1/6/2 (Yes/No/Abstain)

  PROPOSAL: Add argument "OUT 'output' (string)", which would return
            an implementation dependent string the DRMAA implementation used to
            connect to DRM system, useful to know if default connection fails.
  (Discussion was postponed.)
DONE


drmaa_exit( )

  Disengage from DRMAA library and allow the DRMAA library to perform
  any necessary internal clean up.
  This routine ends this DRMAA Session, but does not effect any jobs (e.g.,
  queued and running jobs remain queued and running).

  PROPOSAL: After drmaa_exit(), allow drmaa_init() be be called
            to start up again.  Nesting is not allowed.
  VOTE:  7/0/2 (yes/no/abstain)
DONE

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.

char *drmaa_get_DRM_engine( )
      Output string is implementation dependent and could contain the DRM
      engine and the implementation vendor as its parts.
```

General Notes:

ISSUE 1: There is only one DRMAA session open at the time. Another session could be opened only after the current one is closed. Nesting of sessions is not allowed. It is expected that the DRMAA library will free all the session resources, although this is not guaranteed, so old session resources are not to be used later.

ISSUE 2: Extensibility issue. If DRMAA 2.0 allows multiple connections that means drmaa_session parameter is part of the interface. Why not make this step now? This was on the table before ...

/* ---------- Job Template 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'.
  Only non-vector attributes may be passed.

drmaa_set_vector_attribute(jt, name, value)
    INOUT jt     /* job template (opaque handle) */
    IN    name   /* attribute name (string) */
    IN    values /* vector of attribute value (string vector) */

  Adds ('name', 'values') pair to list of vector attributes in job template
'jt'.
  Only vector attributes may be passed.

  The following are reserved attribute names available in all
  implementations of DRMAA v1.0 (and their respective meanings).
  Vector attributes are marked with a 'V':
      remote command to execute
    V input parameters
         These parameters are passed as arguments to the job.
         The attribute name is "argv".
      job state at submission (on hold, active)
    V job environment
         This environment is set for the job. The attribute name
         is "envv".
      job working directory
      wall clock time limit
      job category
      e-mail to report the job completion and status
         If e-mail is to be sent to the submitter by the DRM system
         this e-mail adress is used. The attribute name is "email".
      standard input, output, and error streams (staging cannot be assumed)

  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:
    input/output files (including stdin/out/err) 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 job not later then

  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.

  ISSUE 1: We need to set up a mechanism to discuss (and flesh-out)
         these attributes & add/delete attributes.   We really need
         a formal proposal to discuss ...
```

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

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

drmaa_get_vector_attribute(jt, name, value)
    IN    jt     /* job template (opaque handle) */
    IN    name   /* attribute name (string) */
    RETURNS values /* vector of attribute value (string vector) */

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

  ISSUE 1: 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?


  ISSUE 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);
```

/* ---------- Job Submission Routines ---------- */

```
drmaa_run_job(job_id, jt)
    OUT job_id      /* job identifier (string) */
    IN  jt          /* job template (opaque handle) */

  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.


drmaa_run_bulk_jobs(job_ids, pjt, start, end, incr)
```

```
    OUT job_ids     /* job identifiers (array of strings) */
    IN  pjt         /* parameterized job template (opaque handle) */
    IN  start       /* beginning index ( unsigned integer?)*/
    IN  end         /* ending index ( unsigned integer?) */
    IN  incr        /* loop increment (integer)*/
```

  Submit a set of parametric jobs, dependent on the implied loop index, each
  with attributes defined in the parameterized job template 'pjt'.
  The job identifiers 'job_ids' are all printable,
  NULL terminated strings, identical to those returned by the underlying
  DRM system. Nonnegative loop bounds are suggested to avoid file names
  that start with minus sign like command line options.

  The special index placeholder is a DRMAA defined string

```
       drmaa_incr_ph /* == $incr_pl$ */
```

  that is used to construct parametric job templates.

  For example:

```
       drmaa_set_attribute(pjt, "stderr", drmaa_incr_ph + ".err" ); /*
  C++/java string syntax used */
```

NOTE: Job template and parameterized job template could be the same structure.
Type-wise, they do not differ at all, the parameterized job templates need to be
parsed and substituted before they could be used.

ISSUE 1: It seems a bit tedious to set up and populate a job template. Doing it
many times over would require CPU cycles and memory. The latter one requires
special consideration. We need mechanisms to avoid this inconvenience. What
about either having a job definition template that has placeholders that get
populated at the job submission instance? One alternative would be to have a
job template duplication mechanism. Another alternative would be to recycle the
old job templates by changing some of its attribute values.
Job definition template would require named placeholders. That could be
accomplished by adding a special character at the beginning of the attribute
name, making it a variable. Job submission interface would have an extra
parameter that holds the variable/value pairs that would be used to create a
valid job request. This resembles to a class definition and an object
instantiation scenario.

The job template would stay the same. Some of the attributes will have named
placeholders or variables that would need to be substituted with provided
values.

A proposed new interface:

```
drmaa_run_parametric_job(job_id, jt, sub_pairs)
    OUT job_id      /* job identifier (string) */
    IN  jt          /* job template (opaque handle) */
    IN  sub_pairs   /* array of variable/value pairs */
```

  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.

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

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

PROPOSAL: A job_id returned during a DRMAA Session will be valid

```
          for use in all routines during that Session.
APPROVED by consensus.


drmaa_get_exit_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
  implementation dependent.
PROPOSAL: Change name to drmaa_get_exit_status
Editor just made the change...

PROPOSAL: Delete drmaa_get_exit_status (drmaa_getpid_status), since
          this is redundant.   Use drmaa_wait.
Approved by consensus
DONE

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.
PROPOSAL: remove synchronous
APPROVED by consensus
PROPOSAL: This routine returns once the action has been acknowledged by
          the DRM system, but does not necessarily wait until the action
          has been completed.
Approved by consensus.
NEW ISSUE: You can now do an action for which there is no corresponding
           way to wait (or tell) if it completes in a nice way.
We will revisit this later.


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.
PROPOSAL: rename 'options' to 'synchronous', and add text from
elsewhere for synchronous.
Approved by consensus
PROPOSAL: Make this a standard set of exit codes... at a minimum,
we should have a standard SUCCESS code.
Andreas has offered to create a real proposal (it is below)

PROPOSAL: Change 'synchronous' to 'timeout' with defined values
for "wait forever" and "don't wait at all"
STRAW VOTE: 4/1/3 (yes/no/abstain)

ISSUE 1: What about "reaping" the job_id (so DRMAA_JOB_IDS_ALL)
doesn't include "exited" jobs... If we need this, we
should add a separate DRMAA routine to reap.
If someone wants this, they should make a proposal.

ISSUE 2: How to reconcile lack of mechanisms on Win32 for getting more
information about job termination when it fails.

The required compromise with this Win32/POSIX conflict should be to
allow for providing more information about job termination if available
but not demand it as necessary in our interface spec. With the modified
semantics specification below of drmaa_wifexited() this is possible. Also
the misleading upper-case function names reminding to macros are changed
to lower-case names:

In the 1.5 version the 'exit_code' OUT parameter of drmaa_wait()
is described as a machine dependent value. To prevent misunderstandings
about the meaning of this OUT parameter it should be renamed to 'stat'
in the next version and DRMAA standard should cover a set of functions
accepting the 'stat' OUT parameter and providing more detailed
information
about job termination if available. An analogous set of macros is defined
in POSIX for analyzing wait3(2) OUT parameter 'stat'.

drmaa_wifexited(OUT exited, IN stat)
Can optionally evaluate into 'exited' a zero value if status was
returned for a job that terminated not normally. A non-zero 'exited'
value indicates more detailed diagnosis can be provided by means of
drmaa_wifsignaled(), drmaa_wtermsig() and drmaa_wcoredump().

drmaa_wexitstatus(OUT exit_code, IN stat)
If the OUT parameter 'exited' of drmaa_wifexited() is non-zero,
this function evaluates into 'exit_code' the exit code that the
job passed to _exit() (see exit(2)) or exit(3C), or the value that
the child process returned from main.

drmaa_wifsignaled(OUT signaled, IN stat)
Evaluates into 'signaled' a non-zero value if status was returned
for a job that terminated due to the receipt of a signal.

drmaa_wtermsig(OUT signal, IN stat)
If the OUT parameter 'signaled' of drmaa_wifsignaled(stat) is
non-zero, this function evaluates into signal the number of the
signal that caused the termination of the job.

drmaa_wcoredump(OUT core_dumped, IN stat)
If the OUT parameter 'signaled' of drmaa_wifsignaled(stat) is
non-zero, this macro evaluates into 'core_dumped' a non-zero value
if a core image of the terminated job was created.

```
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.
```

General Notes: The users could use jobIDs from old DRMAA sessions to query jobs. The implementation could be able to resolve the requests only if the original DRM server is connected to in both sessions. Nothing is guaranteed. The queried jobIDs, even if able to resolve, will not become part of the current session.

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

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

NOTE 1:
In C, we would probably want to return 0 on success and an errno on failure (this is different from libc and open to debate), 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...

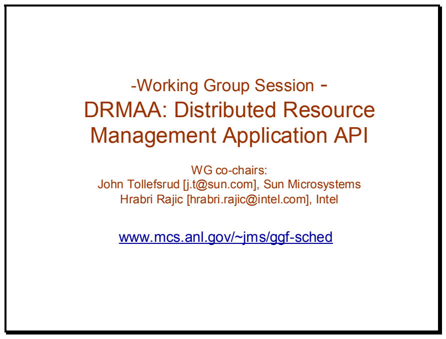
NOTE 2: From libc docs:
The external variable errno is used to hold implementation-defined error codes from library routines. All errno's are positive. Library routines should never clear errno.

```
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*/


contact drmaa_get_contact();
      OUT  contact        /* default contact information for DRM system
(string) */

NOTE 3: Is this function needed if detailed error reporting is in place that
provides this information?
```

APPENDIX B DRMAA pre-GGF5 Status Presentation

Slide 1



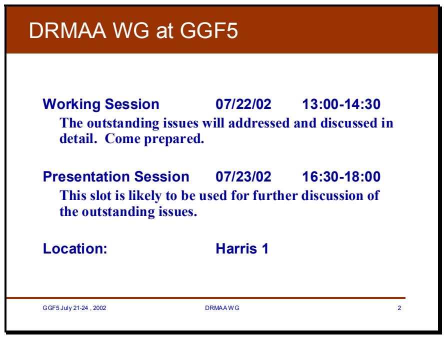
-Working Group Session -

DRMAA: Distributed Resource Management Application API

WG co-chairs:
John Tollefsrud [j.t@sun.com], Sun Microsystems
Hrabri Rajic [hrabri.rajic@intel.com], Intel

www.mcs.anl.gov/~jms/ggf-sched

Slide 2



DRMAA WG at GGF5

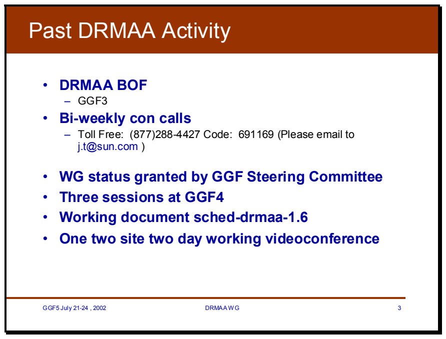
Working Session 07/22/02 13:00-14:30
The outstanding issues will addressed and discussed in detail. Come prepared.

Presentation Session 07/23/02 16:30-18:00
This slot is likely to be used for further discussion of the outstanding issues.

Location: Harris 1

GGF5 July 21-24 , 2002 DRMAA WG 2

Slide 3

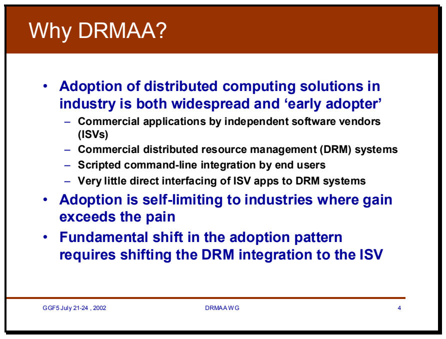


Past DRMAA Activity

- DRMAA BOF
  - GGF3
- Bi-weekly con calls
  - Toll Free: (877)288-4427 Code: 691169 (Please email to j.t@sun.com)
- WG status granted by GGF Steering Committee
- Three sessions at GGF4
- Working document sched-drmaa-1.6
- One two site two day working videoconference

GGF5 July 21-24 , 2002 DRMAA WG 3

Slide 4

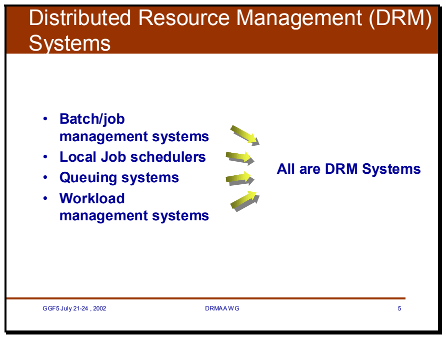


Why DRMAA?

- Adoption of distributed computing solutions in industry is both widespread and 'early adopter'
  - Commercial applications by independent software vendors (ISVs)
  - Commercial distributed resource management (DRM) systems
  - Scripted command-line integration by end users
  - Very little direct interfacing of ISV apps to DRM systems
- Adoption is self-limiting to industries where gain exceeds the pain
- Fundamental shift in the adoption pattern requires shifting the DRM integration to the ISV

GGF5 July 21-24 , 2002 DRMAA WG 4

Slide 5



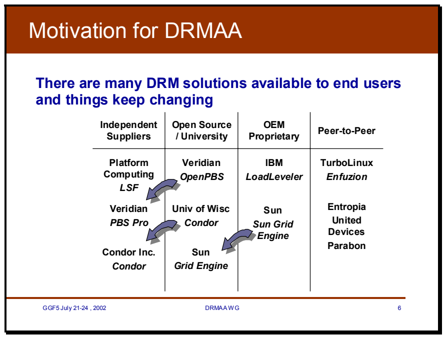
Distributed Resource Management (DRM) Systems

- Batch/job management systems
- Local Job schedulers
- Queuing systems
- Workload management systems

All are DRM Systems

GGF5 July 21-24 , 2002 DRMAA WG 5

Slide 6



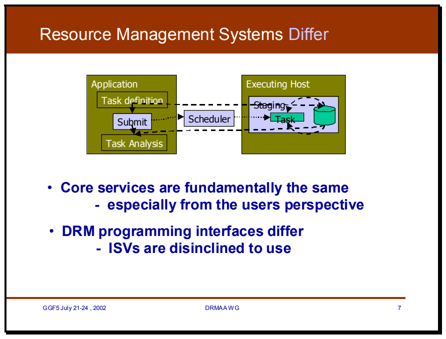
Motivation for DRMAA

There are many DRM solutions available to end users and things keep changing

Independent Suppliers	Open Source / University	OEM Proprietary	Peer-to-Peer
Platform Computing *LSF*	Veridian *OpenPBS*	IBM *LoadLeveler*	TurboLinux *Enfuzion*
Veridian *PBS Pro*	Univ of Wisc *Condor*	Sun *Sun Grid Engine*	Entropia United Devices Parabon
Condor Inc. *Condor*	Sun *Grid Engine*		

GGF5 July 21-24 , 2002 DRMAA WG 6

Slide 7



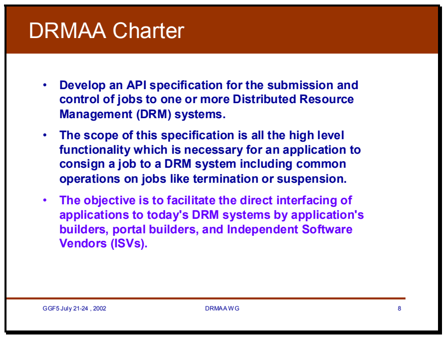
Resource Management Systems Differ

Application
Task definition
Submit
Task Analysis
Scheduler
Executing Host
Staging
Task

- Core services are fundamentally the same
  - especially from the users perspective
- DRM programming interfaces differ
  - ISVs are disinclined to use

GGF5 July 21-24 , 2002 DRMAA WG 7

Slide 8

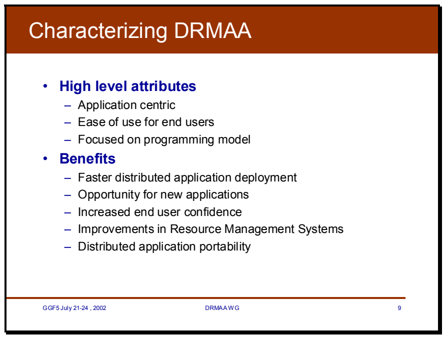


DRMAA Charter

- Develop an API specification for the submission and control of jobs to one or more Distributed Resource Management (DRM) systems.
- The scope of this specification is all the high level functionality which is necessary for an application to consign a job to a DRM system including common operations on jobs like termination or suspension.
- The objective is to facilitate the direct interfacing of applications to today's DRM systems by application's builders, portal builders, and Independent Software Vendors (ISVs).

GGF5 July 21-24 , 2002 DRMAA WG 8

Slide 9

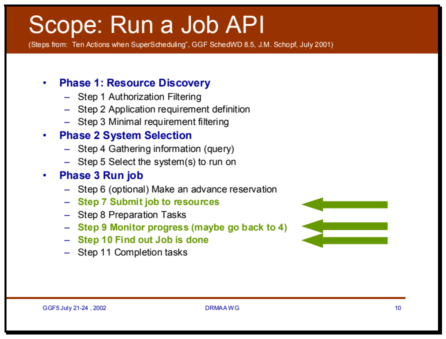


Characterizing DRMAA

- **High level attributes**
  - Application centric
  - Ease of use for end users
  - Focused on programming model
- **Benefits**
  - Faster distributed application deployment
  - Opportunity for new applications
  - Increased end user confidence
  - Improvements in Resource Management Systems
  - Distributed application portability

GGF5 July 21-24 , 2002 DRMAA WG 9

Slide 10



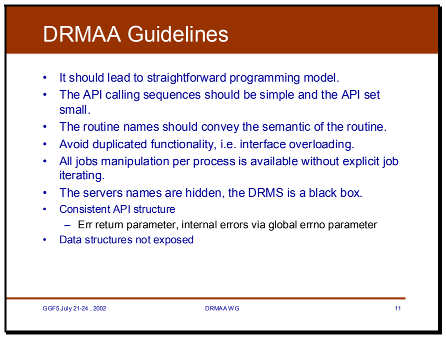
Scope: Run a Job API

(Steps from: Ten Actions when SuperScheduling", GGF SchedWD 8.5, J.M. Schopf, July 2001)

- **Phase 1: Resource Discovery**
  - Step 1 Authorization Filtering
  - Step 2 Application requirement definition
  - Step 3 Minimal requirement filtering
- **Phase 2 System Selection**
  - Step 4 Gathering information (query)
  - Step 5 Select the system(s) to run on
- **Phase 3 Run job**
  - Step 6 (optional) Make an advance reservation
  - **Step 7 Submit job to resources**
  - Step 8 Preparation Tasks
  - **Step 9 Monitor progress (maybe go back to 4)**
  - **Step 10 Find out Job is done**
  - Step 11 Completion tasks

GGF5 July 21-24 , 2002 DRMAA WG 10

Slide 11

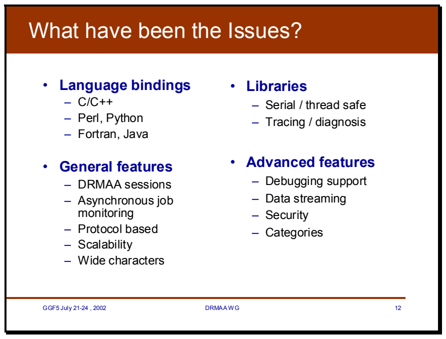


DRMAA Guidelines

- It should lead to straightforward programming model.
- The API calling sequences should be simple and the API set small.
- The routine names should convey the semantic of the routine.
- Avoid duplicated functionality, i.e. interface overloading.
- All jobs manipulation per process is available without explicit job iterating.
- The servers names are hidden, the DRMS is a black box.
- Consistent API structure
  - Err return parameter, internal errors via global errno parameter
- Data structures not exposed

GGF5 July 21-24 , 2002 DRMAA WG 11

Slide 12

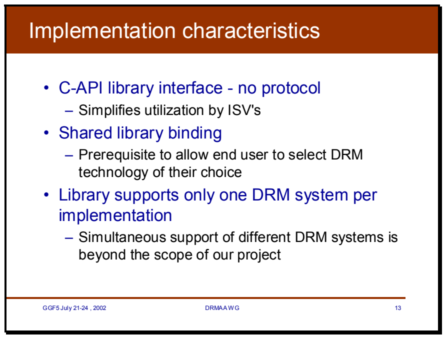


What have been the Issues?

- **Language bindings**
  - C/C++
  - Perl, Python
  - Fortran, Java
- **General features**
  - DRMAA sessions
  - Asynchronous job monitoring
  - Protocol based
  - Scalability
  - Wide characters
- **Libraries**
  - Serial / thread safe
  - Tracing / diagnosis
- **Advanced features**
  - Debugging support
  - Data streaming
  - Security
  - Categories

GGF5 July 21-24 , 2002 DRMAA WG 12

Slide 13

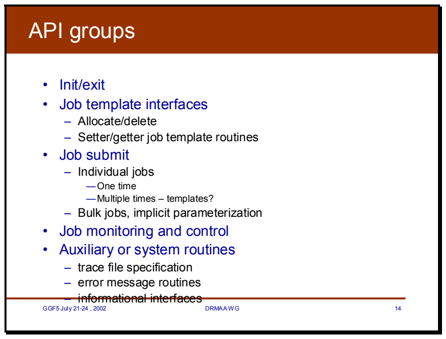


Implementation characteristics

- C-API library interface - no protocol
  - Simplifies utilization by ISV's
- Shared library binding
  - Prerequisite to allow end user to select DRM technology of their choice
- Library supports only one DRM system per implementation
  - Simultaneous support of different DRM systems is beyond the scope of our project

GGF5 July 21-24 , 2002 DRMAA WG 13

Slide 14

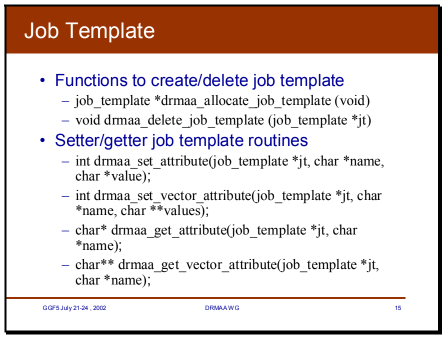


API groups

- Init/exit
- Job template interfaces
  - Allocate/delete
  - Setter/getter job template routines
- Job submit
  - Individual jobs
    - One time
    - Multiple times – templates?
  - Bulk jobs, implicit parameterization
- Job monitoring and control
- Auxiliary or system routines
  - trace file specification
  - error message routines
  - informational interfaces

GGF5 July 21-24 , 2002 DRMAA WG 14

Slide 15

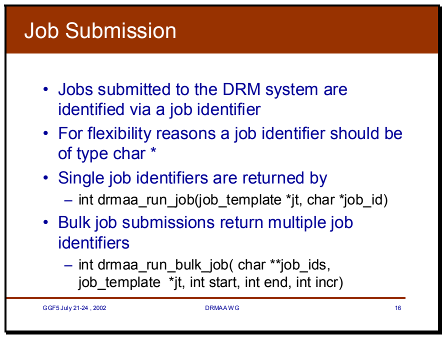


Job Template

- Functions to create/delete job template
  - job_template *drmaa_allocate_job_template (void)
  - void drmaa_delete_job_template (job_template *jt)
- Setter/getter job template routines
  - int drmaa_set_attribute(job_template *jt, char *name, char *value);
  - int drmaa_set_vector_attribute(job_template *jt, char *name, char **values);
  - char* drmaa_get_attribute(job_template *jt, char *name);
  - char** drmaa_get_vector_attribute(job_template *jt, char *name);

GGF5 July 21-24 , 2002 DRMAA WG 15

Slide 16

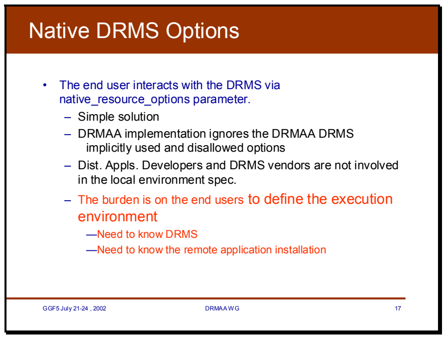


Job Submission

- Jobs submitted to the DRM system are identified via a job identifier
- For flexibility reasons a job identifier should be of type char *
- Single job identifiers are returned by
  - int drmaa_run_job(job_template *jt, char *job_id)
- Bulk job submissions return multiple job identifiers
  - int drmaa_run_bulk_job(char **job_ids, job_template *jt, int start, int end, int incr)

GGF5 July 21-24 , 2002 DRMAA WG 16

Slide 17

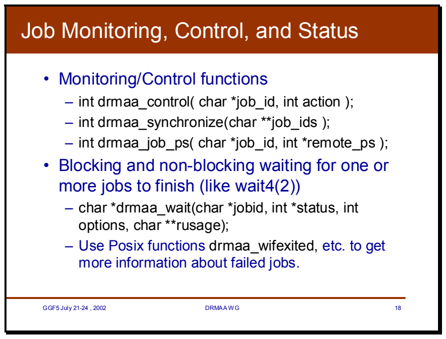


Native DRMS Options

- The end user interacts with the DRMS via native_resource_options parameter.
  - Simple solution
  - DRMAA implementation ignores the DRMAA DRMS implicitly used and disallowed options
  - Dist. Appls. Developers and DRMS vendors are not involved in the local environment spec.
  - The burden is on the end users to define the execution environment
    - —Need to know DRMS
    - —Need to know the remote application installation

GGF5 July 21-24 , 2002 DRMAA WG 17

Slide 18

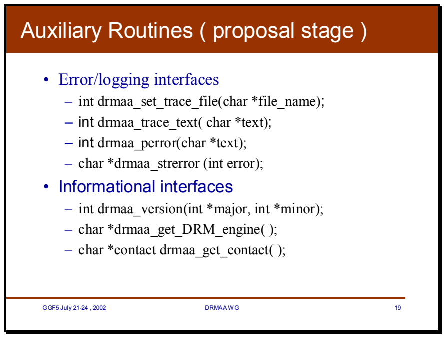


Job Monitoring, Control, and Status

- Monitoring/Control functions
  - int drmaa_control(char *job_id, int action);
  - int drmaa_synchronize(char **job_ids);
  - int drmaa_job_ps(char *job_id, int *remote_ps);
- Blocking and non-blocking waiting for one or more jobs to finish (like wait4(2))
  - char *drmaa_wait(char *jobid, int *status, int options, char **rusage);
  - Use Posix functions drmaa_wifexited, etc. to get more information about failed jobs.

GGF5 July 21-24 , 2002 DRMAA WG 18

Slide 19



Auxiliary Routines (proposal stage)

- Error/logging interfaces
  - int drmaa_set_trace_file(char *file_name);
  - int drmaa_trace_text(char *text);
  - int drmaa_perror(char *text);
  - char *drmaa_strerror (int error);
- Informational interfaces
  - int drmaa_version(int *major, int *minor);
  - char *drmaa_get_DRM_engine( );
  - char *contact drmaa_get_contact( );

GGF5 July 21-24 , 2002 DRMAA WG 19