Standardization of an API for Distributed Resource Management Systems

Peter Tröger
Hasso-Plattner-Institute
14482 Potsdam, Germany
peter@troeger.eu

Hrabri Rajic
Intel Americas Inc.
Champaign, IL 61820
hrabri.rajic@intel.com

Andreas Haas
Sun Microsystems GmbH
93049 Regensburg, Germany
andreas.haas@sun.com

Piotr Domagalski
Poznan Supercomputing and Networking Center
61-704 Poznan, Poland
piotr.domagalski@man.poznan.pl

Abstract

Today's cluster and grid environments demand the usage of product-specific APIs and tools for developing distributed applications. We give an overview of the Distributed Resource Management Application API (DRMAA) specification, which defines a common interface for job submission, control, and monitoring. The DRMAA specification was developed by the authors at the Open Grid Forum standardization body, and has meanwhile significant adoption in academic and commercial cluster systems. Within this paper, we describe the basic concepts of the finalized API, and explain issues and findings with the standardization of such an unified interface.

1 Introduction

The compelling advantage of integrated job processing has lead enterprises to integrate *Distributed Resource Management* (DRM) solutions into their IT environments. Traditionally, this integration has been based on DRM-specific, non-standard interfaces, but each time a newer, more capable release of the DRM software becomes available, the use of non-standard interfaces makes updating the integration an expensive, time-consuming effort for the enterprise.

The *Distributed Resource Management Application API* (DRMAA) specification is a software standard developed in the *Open Grid Forum* (OGF) (former Global Grid Forum) standardization body. Starting in 2001, the specification was developed by numerous contributors from both academia and industry. As a first major milestone, the *proposed recommendation* document was published in June 2004. The updated version of the specification was recently submitted to OGF to reach the final stage of a *full recommendation*. Today DRMAA implementations that adopt the latest specification version are either already available (*Sun's N1 Grid Engine* (N1GE), Condor, Torque, GridWay) or planned (XGrid).

DRMAA defines a unified interface for job submission, monitoring, and control in heterogeneous distributed systems. It provides a programming model for tight interaction with an underlying *Distributed Resource Management System* (DRMS). The specification is designed to encourage both application builders and DRMS vendors to adopt and use the interface in their products. An application using the standard DRMAA interface can be run on any DRM software that has adopted the DRMAA specification.

Within this paper, we will describe the basic concepts of the finalized DRMAA 1.0 specification, as well as some of the outcomes regarding the feature semantics, future issues, and the standardization process itself. Our article can be seen as experience report about the standardization work, and explanation of some of the non-obvious design decisions.

We start with design principles and basic concepts of the DRMAA API, continue with a collection of standardization experiences, describe our efforts for a generalized object-oriented DRMAA API, and conclude the paper with a list of some related work.

2 Design Principles

The DRMAA specification was designed in order to define a simple, lightweight, portable, and modular interface for today's cluster and grid systems. DRMAA provides a fundamental set of operations allowing programmatic access to capabilities common to typical DRM systems. The API follows some basic design principles, which ensures the applicability of the results for both DRMAA implementation providers and API users:

Keep it simple - The standardization of an unified API is always a balancing act between supporting all demanded features, keeping compatibility to existing heterogeneous systems, and having a simple and easy to understand API definition. The DRMAA working group concentrated on the best possible interoperability to major DRM systems, while keeping the amount of functions and concepts as low as possible. This restriction leaded to several features intentionally left out, since their semantic either differs between different DRM systems (e.g. job list operation), or because they are not supported by some of the systems (e.g. workflow identifier). This restriction also keeps the entrance barrier low for implementers, which in turn supports the adoption of the specification.

Language independence - DRMAA is intended to be adoptable to multiple programming languages. The specification defines the API in an abstract syntax, as set of procedural functions with input and output parameters. Existing binding specifications adopt these structural elements to languages like C, Java, Python, Perl, Ruby and C#.

Pluggability - It should be possible to combine different DRMAA implementations for one submission host, in order to allow the parallel usage of multiple DRM systems by one portal implementation or end-user application. This demands a proper definition of late binding issues for the application. It must be possible to identify and choose one of the available implementations at runtime with the DRMAA API.

No user handling - DRMAA does not consider any security aspect of DRM systems, since this would demand a choice for platform- or middleware-specific security concept (e.g. Unix UID, X.509, Kerberos). Such a choice would be in contrast to the overall goal of platform-independence, portability and simplicity. For this reason, DRMAA relies on the security context provided with the user running the DRMAA application.

Thread safety - DRMAA is explicitly designed for supporting multi-threaded applications. The API specification therefore describes the potential impact for multi-threaded usage in all relevant API functions. Even though this requirement puts a higher burden on the DRMAA library implementers, it eases up the development of applications using DRMAA.

Site-specific policies - The DRMAA specification is intended to abstract from DRMS-specific functionalities. Anyway, there might be a particular need for site-specific policies per user. These policies typically concern site-specific attributes, such as resources to be used by the job or the job scheduling in relation to other jobs. DRMAA therefore defines the notion of *job categories*, which describe cross-site behaviors not covered by the specification.

2.1 API Basics

The DRMAA specification consists of a set of functions, which are grouped into init and exit routines, job template routines, job submission and monitoring routines, job control routines, and auxiliary routines. The API is based on a session concept, where all submitted jobs are grouped in a library-instance specific session. This allows the application developer to perform synchronization and monitoring on all jobs submitted by the particular application instance. The session concept also establishes an explicit cleanup phase, needed by some of the object-oriented implementations on top of DRMAA C libraries. IN addition, the application can expect the DRMAA library to no longer influence the set of jobs in the DRM.

Based on practical implementation experiences, multiple open sessions as well as nested sessions are intentionally left out. Both concepts would demand more fine-grained synchronization and monitoring primitives. The DRMAA job control routines are free to accept job ids from previous DRMAA sessions. This rule arose from practical experiences with unstable applications, which need to continue there job-related operations after a restart. Another example are long-running job workflows, which maintain there own list of valid job handles.

2.2 Job Template Routines

The description of a job to be submitted to the DRMS is encapsulated in a *job template*. The job template is defined as set of key-value pairs, containing mandatory, optional and implementation-specific attributes. Examples for mandatory attributes are the executable name, the working directory or the output stream file. Examples for optional attributes are the absolute job termination time or a maximum wall clock time limit. Most of these parameters arose from an early comparison of DRM submission parameters, and from the initial DRMAA implementation in Sun N1GE. The working group constantly re-evaluates mandatory and optional attributes in the specification. So far, most existing implementations for cluster and grid systems ignore the set of optional DRMAA attributes and provide full support only for the mandatory attributes.

DRMAA supports the identification of all supported attributes during runtime. Job templates are not bound to a particular job execution, and therefore can be reused for multiple submissions. The specification defines the template to be evaluated at submission time; therefore all setter functions only consider errors like incorrect attribute name, invalid value format, or conflicting setting.

2.3 Job Submission and Monitoring Routines

A job can be submitted with DRMAA either as single job (`drmaa_run_job()`) or set of bulk jobs (`drmaa_run_bulk_jobs()`). For bulk jobs, a beginning index, ending index and loop increment can be specified. Template attributes can contain a placeholder string for the current parametric job index during submission.

The `drmaa_job_ps()` function allows to query for the status of a job (see figure 1). A queued job can either be ready for execution or in a hold state. A job on hold can be triggered by an explicit `drmaa_control()` call, or by a submission as hold job, which is specified with one of the mandatory job template attributes. Both cases are represented with the 'user on hold' job state. A held job can also be triggered by the DRM system itself or by a combination of both. Held jobs are explicitly released with another `drmaa_control()` call.

A job in the status class 'running' can either be actively executed or in a suspend state. The suspend state might be explicitely triggered by the user through `drmaa_control()`, which leads to `DRMAA_PS_USER_SUSPENDED`, or by the system itself, which leads to (`DRMAA_PS_SYSTEM_SUSPENDED`).

For finished jobs, `drmaa_job_ps()` returns `DRMAA_PS_DONE` in case of a successful execution, or `DRMAA_PS_FAILED` when the job ended unexpectedly. A monitoring call might also lead to `DRMAA_PS_UNDETERMINED`, which reflects a problem with the status determination in the underlying DRMS. In this situation, DRMAA applications are free to perform additional calls to `drmaa_job_ps()`. The implementation experiences showed that this is desperately needed for temporally effects in idle-time or wide-area grid environments.

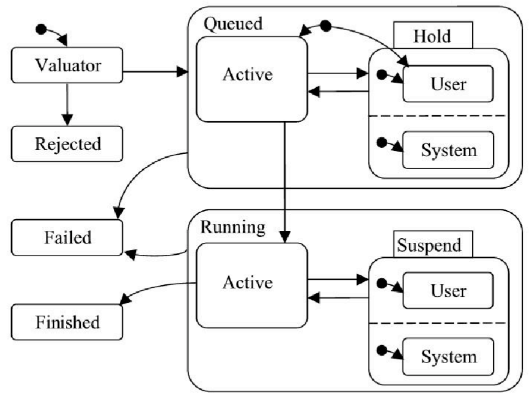

Figure 1. DRMAA Job State Transition [2]

2.4 Job Control Routines

The state of a submitted job can be changed through the `drmaa_control()` function. Different control command constants allow suspending, resuming, holding, releasing and terminating a job. The routine also supports control actions on all submitted jobs in the current DRMAA session (`DRMAA_JOB_IDS_SESSION_ALL`).

An application can synchronize the finishing of a set of jobs with `drmaa_synchronize()`. Input arguments are the list of job identifiers, a timeout specification, and a dispose flag. This routine also can act on all jobs in the current session by using `DRMAA_JOB_IDS_SESSION_ALL` as job ID parameter. The timeout parameter restricts the blocking time of the operation, from zero to indefinite.

The dispose parameter specifies how to treat reaping of the remote job's system resources consumption and other statistics. If dispose is set to false, the job's information remains available and can be retrieved through `drmaa_wait()`. If dispose is set to true, the job's information is not retained.

drmaa_wait() allows to wait for the finishing of a particular job, and returns the termination status information. Input arguments are the job identifier and the timeout value, output arguments are the job ID of the finished job, an opaque status code and resource usage information. The routine reaps jobs and their status information on a successful call. The resource usage information is provided as set of key-value-pairs, which contain implementation-specific resource indicators.

The status code resulting from drmaa_wait() is used in a series of functions, in order to provide more detailed information about job termination. The overall semantic of this approach is modeled after POSIX wait() and the related preprocessor macros (e.g. WIFEXITED).

2.5 Auxiliary Routines

drmaa_get_contact() provides the list of supported contact strings before drmaa_init(), and returns the current contact string for an open session. The drmaa_version() routine provides the version number of the supported DRMAA specification, and drmaa_get_DRM_system() respectively drmaa_get_DRMAA_implementation() provide information about the specific library implementation. All functions are independent from the DRMAA session handling, and can therefore be used before a drmaa_init() or after a drmaa_exit() call.

3 Standardization Experiences

The following section describes a list of experiences and important design decisions, as additional source of information for DRMAA adopters and for maybe future standardization efforts in job management API's. The presented issues mostly caused heavy discussions or increased implementation efforts, and where either resolved or delayed to a later update of the DRMAA specification.

3.1 Implementation Complexity

DRMAA was designed with the overall goal of having a small and easy to understand API definition, which fits for most application developer needs. Even though the set of available implementations shows the success of the approach, there are still challenging aspects, such as multithreading issues.

The specification demands the library to be threadsafe. The demand for supporting multiple application threads, together with the goal of satisfactory performance leads to significant implementation efforts for library developers. The specification also needs to be very precise regarding the functional descriptions, in order to provide enough information about the intended behavior in these cases.

One example: Since session information can be changed during a blocking call (e.g. job termination with drmaa_control() during drmaa_wait() operation), both functions must define their behavior and the according error codes for such multithreading cases. The DRMAA group decided that the session state is reevaluated after each such parallel operation. An application thread waiting for a session job therefore must return if another activity (e.g. explicit termination of the job) provokes a relevant session state change. This must be ensured by the DRMAA library implementation for all possible cases.

High-level language binding implementations such as Java DRMAA are currently based on the according C libraries, and benefit from the already implemented multithreading support. Mostly, the performance advantage of the C implementation outweighs the additional effort for 'native' calls in non-native environments like Java.

3.2 Reference Implementation

From the very beginning, the development of DRMAA API was assisted by a continuous implementation in the Sun N1GE 6 product line. Today DRMAA defines the default API for programming of the N1GE6 system. The product-quality reference implementation helped to identify real-world issues with the specification, and acted as major driver for the API finalization.

In the later phases of the standardization process, the existence of preliminary implementations provided a high barrier for major concept changes. While this can be seen as an advantage in terms of a stable interface definition, it still avoids the fast adoption of new or improved concepts (e.g. for monitoring support).

The existence of a reference implementation also indirectly leaded to a DRMAA compliance test suite. In 2005, the N1GE product DRMAA test suite was donated by Sun Microsystems to the DRMAA group as a base for the official DRMAA test suite. The test suite was a mandatory precondition for demonstrating the practical interoperability of the existing implementations for the OGF. It currently contains of around 4000 lines of C code with 13 complex tests, especially for the multithreading access rules of the DRMAA specification. Due to the informal description of operational semantics in DRMAA, it is not possible to prove

DRMAA-compliance of an implementation with a successful test suite run. This problem relates to the well-known problem domain of automated software testing. The test suite now mainly provides a valuable source of information for DRMAA library developers. Multiple specification improvements arose from issues identified through test suite errors with different DRMAA implementations.

3.3 Client-API or Remote-API ?

Due to the increasing orientation of OGF towards SOAP-based interfaces [4], the group received multiple request regarding a WSDL-mapping of the DRMAA interface. An analysis showed that the initial focus on a client-side API prevents the direct mapping of DRMAA concepts to a server-side API. One example are blocking calls in the `drmaa_wait()` and `drmaa_control()` functions, which can take an unknown amount of time and therefore are inappropriate for a client-server scenario.

The session concept of the DRMAA API also prevents an easy migration to remote call scenarios: The specification demands multiple library calls to perform a job submission - session initialization, job template creation, parameter change, and job startup. Service-oriented grid interfaces rather would try to combine all information in one document (e.g. as JSDL document), followed by a submission of this information in one round trip.

The group finally decided to declare these issues as out-of-scope for the current specification. However, other groups at OGF such as OGSA-BES (Open Grid Services Architecture - Basic Execution Service) currently develop appropriate server-side job submission specifications in collaboration with DRMAA group members. Future DRMAA versions based on the IDL specification (see section 4) will consider SOAP-based RPC scenarios with an according *Web Service Resource Framework* (WSRF) binding specification.

3.4 Re-use of POSIX concepts

During the development of the specification, it was attempted to apply existing concepts and wording from POSIX (1003.1 and 1003.2d) to the specification text, in order to ease up adoption by application developers. As one example, DRMAA relies on the semantics of POSIX `wait()` and the according C macros (e.g. `WIFEXITED`) for the further processing of termination information.

The resulting set of DRMAA status functions must be called in a specific order, e.g. first asking if the job was signaled and then querying for the core dump availability. Multiple language binding authors, as well as several DRMAA users, expressed there wish to favor a single status query call over multiple POSIX-like functions for the same purpose, similar to the `drmaa_control()` routine. The upcoming next version of the specification therefore will support such a concept of a job status data structure.

3.5 Time Definitions

DRMAA supports the notion of partial timestamps, which allow the incomplete specification of time information (e.g. "Start job at 8 o'clock."). Partial timestamps are expressed as string variable in a special syntax, and are evaluated for complete time information at job submission time. This enables the re-usage of incomplete time stamp definitions for multiple job runs.

Standard time formats (like ISO 8601) also allow an incomplete specification of time stamp information, but define specification-time completion rules in order to create unambiguous time stamp information. DRMAA extends this model by explicitly supporting the formulation of relative timing information for job submission attributes.

Even though the concept of partial timestamps is helpful for DRMAA users and portal developers, it turned out to be a difficulty for some language bindings. The C language binding simply adopts the formulation of partial timestamps as string variable. In contrast, modern object-oriented development in Java and C# need a mapping of partial time stamp information to class library types. Since the mapping is influenced by the capabilities of the existing type, the IDL specification defines partial time stamps only as 'native' data type, together with a detailed description of intended functionalities. This enables appropriate custom mappings for the language-binding authors, instead of providing an own representation of date and time information.

3.6 Monitoring Features

Several DRMAA users criticized the lack of support for resource-related monitoring parameters, like cluster load information, hardware status or operating system type. As one major problem, the provisioning of such status information in a standardized API would demand a common information model for the monitoring parameters. For example, the specifications of *CPU type* or *disk quota* parameters need to consider heterogeneous DRM products, hardware platforms and operating systems. Even though some standards (like CIM

[8] and *Job Submission Description Language* (JSDL)) provide such unified information model, it is still relevant to check the applicability of the model on today's cluster and grid environments. Facing this specific problem, the JSDL group meanwhile provides mapping information of their information model to different batch systems. Upcoming versions of the DRMAA specification need to consider this recent work for the extension of the API.

3.7 Job State Semantics

DRMAA so far does not provide an extensive description of job state semantics, but uses only keywords ('suspended', 'running', 'queued') for its definition of possible job state transitions. With the development of multiple DRMAA implementations, the group figured out that some of the states were hard to map for particular DRM systems.

One example is the *suspended* job state. The current DRMAA implementation for the Condor system [5] works in the *vanilla universe*, which does not support the suspension and resuming of particular jobs. The Condor DRMAA implementation therefore uses the `condor_hold()` command to implement the mandatory `drmaa_control()` suspend operation. Since this Condor command kills and requeues the job in vanilla universe, the job might restart on another machine. According to the specification, this behavior does not break DRMAA compliance, since the job state 'SUSPENDED' is not further described. For the end user, the visible effect is a re-scheduling and not the expected suspending of the job on the original machine.

There are two possible solutions for bringing more clarity to the users and developers. The specification could either provide explicit descriptions of the job state semantics, or could leave out semantically interpretations completely. The first choice would lead to non-implementable features for particular DRM systems, while the second choice leads to different behaviors with different DRM systems. So far, the DRMAA group favors the second approach. Recent discussions identified a third approach, which would mark the critical job states as optional. Implementations then would be free to not support some of the DRMAA job states.

4 Object-Oriented Bindings

Even though DRMAA started with the idea of a language-independent specification, it mainly was written with a procedural C-language slant. With the ongoing development and adoption effort, several users demanded a mapping of DRMAA to object-oriented programming languages. The first mapping was performed with the Java language binding specification, which was also implemented for the Sun N1GE product. In parallel, other working group members developed a language binding specification for the .NET environment. Based on two independently developed object-oriented language bindings, the in-depth comparison identified general issues for an object-oriented API mapping. For this reason, both binding authors started the alignment of these mapping issues. Simple examples are variable and method naming, as well as error information structuring. Complex cases are the realization of partial time stamps and job templates in appropriate language constructs.

To keep and maintain the newly derived DRMAA semantics for OO-environments, the group decided to rely future versions of the language-independent DRMAA specification on the OMG Interface Definition Language (IDL) [6]. Future language binding documents should only need to define a mapping between DRMAA IDL constructs and their specific language constructs, instead of creating own syntactical rules for the interface. This approach is also used in other popular specifications, for example the W3C Document Object Model. IDL ensures consistent API semantics in all language bindings, whereby the binding author only needs to concentrate on language-specific issues. The general representation of DRMAA is described in abstract IDL concepts - interfaces, classes, value types, reference types and properties. The re-use of OMG IDL language mapping specifications is omitted, since these mapping rules rely on CORBA runtime environment mechanisms. Instead, each DRMAA language binding defines own rules which are most appropriate to the according environment.

Job templates are represented as interfaces with according member attributes. As one interesting aspect, optional attributes also needed to be reflected in the OO-API design. They must be represented in the job template interface layout for a portable API, even though they might not be usable for a particular implementation. The current IDL specification therefore introduces a new exception type for the job template attributes, which is thrown when an unsupported attribute is about to be changed.

Modern OO-languages use exception hierarchies, in order to be able to react on a set of exceptional situations with one catch block. As future DRMAA specifications need to consider this demand, the IDL-document already defines groups of error classes. In case, this grouping can be mapped to according language exception hierarchies.

DRMAA originally relies on two special data types: an unbounded unsorted list of strings (for job or argument lists), and an unbounded dictionary of key-value pairs (for resource usage information and environment variables). Since both constructs have different representations in OO-languages, the IDL spec defines two abstract data types, which must be mapped by the language binding specification to a matching construct.

Even with the support for garbage collection mechanisms in modern object-oriented languages, the `delete()` operation for job templates remains an explicit function. This design decision reflects the missing support for guaranteed object finalizer execution in some of the object-oriented languages, which is needed for maybe mandatory cleanup operations in the DRMS.

5 DRMAA Usage

DRMAA is meanwhile used as job submission and monitoring API in several applications, like meta-schedulers or compute-intensive commercial frameworks, also for distributed software compilation or as integrated data grid solution.

As one example, the OpenDSP project[1] from the Poznan Supercomputing and Networking Center developed a Web Service based interface for multi-user access to DRMAA-enabled DRM systems. The usability and efficiency of OpenDSP has been proved in several productive deployments, like the Faculty of Civil and Geodetic Engineering at University of Ljubljana (earthquake simulation) or the InteliGrid project (remote access to engineering and construction applications).

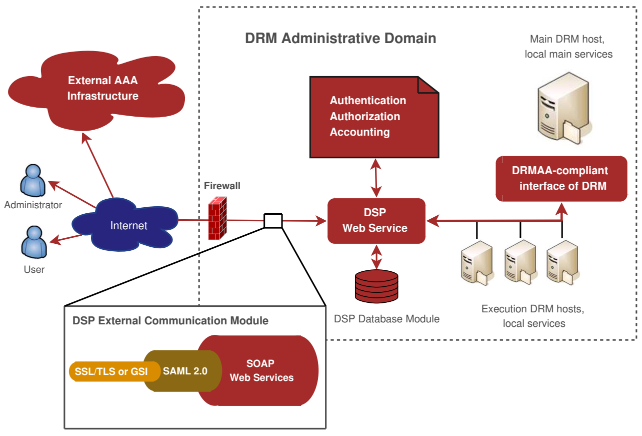
DRM Administrative Domain
Main DRM host, local main services
External AAA Infrastructure
Authentication Authorization Accounting
DRMAA-compliant interface of DRM
Firewall
Administrator
Internet
User
DSP Web Service
DSP Database Module
Execution DRM hosts, local services
DSP External Communication Module
SSL/TLS or GSI
SAML 2.0
SOAP Web Services

Figure 2. OpenDSP architecture

There are several other examples for successful DRMAA-based applications and frameworks. The Target System Interface Framework for UNICORE is now using is DRMAA as front-end to systems such as Condor, Sun Grid Engine, Globus or Torque [7]. EGEE relies on DRMAA for integration issues, and the MOAB scheduler can interface DRMS systems with a DRMAA interface. The latest list of use cases and DRMAA implementations is maintained on the DRMAA home page[2].

[1]http://sourceforge.net/projects/opendsp/

6 Related Work

The OGF JSDL working group provides an abstract definition of a type system for job submission parameters - attributes, their relationship, and value ranges. The abstract syntax description is mapped to a normative XML schema definition. In contrast to DRMAA, questions of job submission, scheduling, workflow management and monitoring are declared as out of scope. The reusability of one JSDL document is a major driver in the template layout - job-specific information like start time, end time or submission status are not represented. JSDL supports a broader but different range of attributes as DRMAA, for example for the definition of resource requirements or data staging issues. Multiple research projects already adopted parts of the specification.

The OGF OGSA-BES working group uses JSDL to define a SOAP-based job submission interface. The specifications from this group are embedded in the overall OGF work for the design of the Open Grid Services Architecture (OGSA). OGSA-BES relies on a very specific execution-container approach, and is still work in progress. In contrast to DRMAA, it does not consider the specific issues of using the API with a local library implementation.

The GridSAM project provides an extensible implementation of the JSDL and OGSA-BES interface specifications. Job submission plug-in?s are available for Globus and the POSIX fork() interface, as well as for DRMAA 1.0 implementations.

The Commodity Grid Kit (CoG) [1] is developed by the Argonne National Laboratory, and provides an end-user API for job submission, grid security, task graphs and file transfer. Even though CoG is developed as independent project, there is a tight inter-connection to features and protocols from the Globus toolkit [3]. CoG is used in many specialized grid portal projects, since it offers the most powerful, but specialized Java-based API implementation for Globus. Similar to GridSAM, CoG can use DRMAA for interfacing a DRM system in a vendor-independent manner.

[2]http://www.drmaa.org/

The *Simple API for Grid Applications* (SAGA) provides a high-level API for grid application programming. SAGA is derived from the Grid Application Toolkit (GAT) library, and covers logical and physical file handling, information system management, and job submission, monitoring and distributed communication. Even though GAT is a stabilized toolkit with multiple backend's for grid- and cluster-systems, the SAGA standardization is in an early phase. Existing implementations of GAT are able to interface DRMAA library implementations for job submission.

7 Conclusion

DRMAA is an approved specification for a small common DRM system API, which enables easy transition from cluster-based end user applications to grid-based end user applications. DRMAA has multiple interoperable implementations for relevant cluster and grid systems (Sun N1GE, Torque, Condor, GridWay), and is increasingly supported in various programming languages. Several industrial customers, especially for the Sun N1GE system, use DRMAA in their real-world productive environments.

Regarding the performance characteristics of DRMAA, practical measurements in Sun N1GE showed interactions with the DRMAA library to be comparatively faster than interactions via the command line. Experiments showed that job submission rates can easily be doubled just by using DRMAA due to lower job submission overhead. This is mainly reasoned by the careful re-use of existing network connections to scheduling and queuing servers in the library implementation. Other implementations of DRMAA (e.g. Condor) rely on re-using the command line utilities of the DRM system, due to non-available wire format specifications for the particular system. These systems naturally show a similar performance to traditional batch submission scripts.

In our standardization work, we experienced several interesting issues. Independent parallel development of different language bindings ('n-version standardization') can identify general issues with a root specification. Even with an comparatively lightweight API, the definition of implementation behavior in a concurrent and non-reliable grid scenario still demands continuous in-depth analysis and practical implementation experience. The combination of academia and industry people in one working group ensured both theoretical completeness and practical applicability. A reference implementation by an industrial partner clearly acts as major driver for community acceptance and additional implementations of a standard.

The future versions of the specification will provide an improved support for object-oriented programming languages. The DRMAA working group will also tackle the extension of the API with respects to most demanded features, like job workflow management, improved file staging support or JSDL-based job template descriptions. Contributions and discussions are welcome on the DRMAA mailing list[3].

References

[1] Gregor von Laszewski and Ian Foster and Jarek Gawor and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[2] Hrabri Rajic and Roger Brobst and Waiman Chan and Fritz Ferstl and Jeff Gardiner and Andreas Haas and Bill Nitzberg and John Tollefsrud. Distributed Resource Management Application API Specification 1.0. http://www.drmaa.org/, 2004.

[3] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[4] Karl Czajkowski and Don Ferguson and Ian Foster and Jeff Frey and Steve Graham and Tom Maguire and David Snelling and Steve Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution, 05 2004.

[5] M.J. Litzkow and M. Livny and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[6] Object Management Group, Inc. *Common Object Request Broker Architecture: Core Specification.* Prentice Hall Professional Technical Reference, march 2004.

[7] Morris Riedel, Roger Menday, Achim Streit, and Piotr Bala. A drmaa-based target system interface framework for unicore. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 133–138, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Winston Bumpus and John W. Schweitzer and Patrick Thompson. *Common Information Model.* John Wiley & Sons Inc, 2000.

[3]drmaa-wg@ogf.org