Standardised job submission and control in cluster and grid environments

Peter Tröger*

Blekinge Institute of Technology,
372 25 Ronneby, Sweden
E-mail: peter@troeger.eu
*Corresponding author

Hrabri Rajic

Intel Corporation Champaign,
IL 61820, USA
E-mail: hrabri.rajic@intel.com

Andreas Haas

Sun Microsystems GmbH 93049 Regensburg,
Germany
E-mail: andreas.haas@sun.com

Piotr Domagalski

Poznan Supercomputing and Networking Center,
61-704 Poznan, Poland
E-mail: piotr.domagalski@man.poznan.pl

Abstract: Cluster and Grid environments mostly required the use of product-specific Application Programming Interface (APIs) to submit, control and monitor computational jobs in the past. The *Open Grid Forum (OGF)* standardisation body therefore has developed several specifications to fill the gap and enable developers to code to few standardised APIs.

This article discusses the details of one of these specifications, the *Distributed Resource Management Application API* Grid recommendation. We compare the basic concepts of the finalised API to other specifications from the same area and explain issues and findings uncovered during the standardisation of such unified interface.

Keywords: job submission; job monitoring; job control; cluster system; array job; IDL; interface definition language; API; application programming interface; standardisation; open grid forum; DRM system; DRMS; distributed resource management system; DRMAA; distributed resource management application API

Reference to this paper should be made as follows: *Tröger, P., Rajic, H., Haas, A. and Domagalski, P.* (XXXX) 'Standardised job submission and control in cluster and grid environments', *Int. J. of Grid and Utility Computing*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Peter Tröger works as Adjunct Lecturer and Researcher at the Blekinge Institute of Technology. His research interests are middleware-based distributed systems, with a focus on Grid Computing and service-oriented environments. He is a co-chair in the OGF DRMAA working group and collaborates with different industry partners in the area of dynamic resource usage for service environments.

Hrabri Rajic received his PhD degree in Nuclear Engineering in 1988. He is a founding member and co-chair of OGF DRMAA working group. His current interests include distributed systems, generic programming, cross platform graphical user interfaces and parallel tools. He is a member of Intel's Performance, Analysis and Threading Lab.

Andreas Haas joined Sun Microsystems in 2000 with the acquisition of Gridware Inc. and works as Senior Staff Engineer on Sun Grid Engine. He is also a founding member and co-chair of OGF DRMAA working group.

Piotr Domagalski joined Poznan Supercomputing and Networking Center in 2006 and has been actively involved in various research activities in the scope of EU funded IST projects. He is interested in OGF standardisation work and helps to bring various grid standards on the market under the FedStage open source initiative.

1 Introduction

The compelling advantages of the integrated job processing has lead enterprises to introduce *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 DRM software becomes available, the use of non-standard interfaces makes updating the integration an expensive and time-consuming effort. The advances of Grid Computing in the last years have even more increased the need for standardised interfaces to DRM systems in order to connect virtual organisation infrastructures using Grid middleware frameworks.

The *Open Grid Forum* (OGF) (former Global Grid Forum) standardisation body is a collaborative effort of representatives from academia and industry to foster interoperability in Grid systems. The work in OGF is separated into different areas, each containing multiple working and research groups. A working group tackles a particular standardisation topic and publishes its work in OGF documents. The publication of these specifications is organised after a document process (Catlett, 2001), which distinguishes between recommendation and informational documents, experience reports and community practice reports. Software interoperability specifications are submitted first as *proposed recommendations*, which require favourable reviews from both the public and the OGF Steering Committee. If such proposed recommendations are shown to be mature, e.g., by having interoperable implementations in real-world settings, they evolve to full *Grid recommendation* status. Other document types in OGF have fewer restrictions in order to be published as OGF documents.

In the following article, we will give details about the results of the *Distributed Resource Management Application API* (DRMAA) specification work. DRMAA is a software standard developed in the OGF standardisation 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 reached the final stage of a *Grid recommendation* in 2007. Today, DRMAA implementations are available for popular DRM systems, such as *Sun's N1 Grid Engine* (N1GE), Condor from the University of Wisconsin (Litzkow et al., 1988), Torque from Cluster Resources Inc., Globus (via Grid Way) from the Globus consortium (Foster and Kesselman, 1997) and Platform's Load Sharing Facility (LSF).

DRMAA defines a unified interface for job submission, monitoring and control in heterogeneous distributed systems. It therefore 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 system that has implemented DRMAA specification.

Besides DRMAA, there are other OGF specifications from the same area of job submission and control:

JSDL: The OGF *Job Submission Description Language* (JSDL) provides an abstract type system for job submission parameters, including job attributes, their relationships and value ranges (Anjomshoaa et al., 2005). The abstract description of types is mapped to a normative XML schema definition. In contrast to DRMAA, questions of job submission, scheduling and monitoring are declared out of scope. Multiple research projects have already adopted parts of the specification.

SAGA: The *Simple API for Grid Applications* (SAGA) provides a high-level Application Programming Interface (API) for Grid application programming (Goodale et al., 2007). It is derived from the *Grid Application Toolkit (GAT)* library (Allen et al., 2003) and covers logical and physical file handling, information system management and job submission, monitoring and distributed communication. Even though GAT is a stable toolkit with multiple implementations, the SAGA standardisation is still under work, close to becoming a proposed recommendation.

OGSA-BES: The *OGSA Basic Execution Service* (OGSA-BES) specification uses JSDL to define a SOAP-based job submission interface (Grimshaw et al., 2006). The specification is embedded in the overall OGF *Open Grid Services Architecture* (OGSA) work. OGSA-BES relies on the availability of an execution container and is mainly intended for SOAP-based job submission in a distributed Grid environment.

Different adoptions exist in Grid middleware for all these specifications, provided by both research and industry. GridSAM project provides an extensible implementation of the JSDL and OGSA-BES interface specifications (Lee et al., 2004). Job submission plugins are available for Globus and the POSIX fork() interface, as well as for DRMAA 1.0 implementations. Unicore uses DRMAA internally to access existing DRM systems, and provides JSDL and OGSA-BES interfaces to the outside (Erwin and Snelling, 2001). Gridway uses DRMAA as a job submission API and can parse JSDL descriptions (Herrera et al., 2004).

There also exist other job submission and control APIs which are developed outside of the OGF standardisation body. The most prominent example is the Commodity Grid Kit (CoG), which provides an end-user API for job submission, Grid security, task graphs and file transfer (von Laszewski et al., 2001). Even though the CoG is developed as an independent project, there is a tight interconnection to features and protocols from the Globus toolkit. CoG is used in many specialised Grid portal projects, since it offers one the most powerful, albeit specialised Java-based API implementation for Globus. Similar to GridSAM, CoG can use DRMAA for vendor-independent interfacing to a DRM system.

In the context of this article, we will describe the basic concepts of the final DRMAA 1.0 Grid recommendation. We will discuss some of the relevant features semantics, possible future extensions, and the standardisation process itself. Our article can be seen as experience report about OGF job submission APIs and the standardisation work itself. We start with design principles and basic concepts of the DRMAA API, continue with our standardisation work experience and finish with explanation of our efforts for a generalised DRMAA API description in the context of both object-oriented and procedural languages.

2 Design principles

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 as follows, which ensures the applicability of the results for both DRMAA implementation providers and API users:

Keep it simple – The standardisation 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. The 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 facilities 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, to allow a parallel use of multiple DRM systems by one portal implementation or an end-user application. This demands a proper definition of late binding issues for the application. It must also be possible to identify and choose one of the available implementations at runtime.

No user handling – The specification does not consider any security aspect of cluster and Grid 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 resulting 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 library implementers, it eases up the development of applications using DRMAA.

Site-specific policies – The 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 behaviours not covered by the specification.

The intentional ignorance of security issues turned out to be beneficial during the unforeseeable long time of standardisation and adoption of DRMAA. The continuous research and according development of different security standards for the Grid – like with WS-Security, X.509 proxy certificates (Tuecke et al., 2004) or Shibboleth (Sinnott et al., 2006) – needs to be considered in all API standards referring to a particular security mechanism. For DRMAA, this would conflict with the overall goal of easy and broad adoption of the specification.

Thread safety as relevant factor for a standardised API was identified in the later stages of the DRMAA development process, mainly by implementation and user experiences in Sun N1GE. The according guidelines and rules were therefore added when the document evolved from *proposed* to *Grid recommendation* status.

Job categories on the other hand were considered from the very beginning, but were never widely used in the implementations. One identified root cause is the lack of standardised policy semantics. Adopters are free to define their own job categories, according to the usage scenarios of the underlying DRM system. These implementation-specific categories are not useful in a portable DRMAA-based application, since they always relate to a particular product. One example is the Condor DRMAA implementation where the job category expresses the *universe* to be used by the job. This particular attribute is only valid for the Condor execution environment, and would therefore not trigger the same kind of behaviour in another DRM system.

3 DRMAA API

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. An application using the DRMAA interface therefore

- gets the list of supported DRM systems by calling drmaa_get_contact()
- initialises a session by calling drmaa_init() with one of the resulting contact strings
- submits one or more single or bulk jobs
- performs control and monitor operations for a single job or for all jobs submitted in the session so far
- closes the session with **drmaa_exit** ().

Session-based interaction was mainly introduced in DRMAA for performance reasons. Typical control programs, like shell scripts or front-end applications, let the user submit and monitor a possibly large amount of jobs. The session concept allows the front-end application to perform synchronisation and monitoring on all jobs submitted by the particular application instance. It also introduces an explicit finalisation phase, which gives the DRMAA library a chance to perform cleanup operations. This was especially needed by some of the object-oriented implementations on top of DRMAA C libraries, like the Java DRMAA implementation in Sun N1GE, which relies on JNI calls to the C DRMAA library of the same product.

Besides the classical front-end applications, DRMAA charter also states portal applications as another relevant application class. Most portal environments, like Java servlet containers, already contain a session mechanisms on both the client and the application scope. The DRMAA session concept therefore does not conflict with these execution environments.

Based on initial implementation experiences, multiple open sessions as well as nested sessions were intentionally left out of the API. Both concepts demand fine-grained synchronisation and monitoring rules, in order to avoid mutual exclusion conditions for control activities on jobs.

On the other side, DRMAA job control routines are meanwhile free to accept job IDs from previous DRMAA sessions. The addition of the rule was another part of the latest specification update, based on practical experiences with long-running job workflows, which maintain there own list of valid job handles. Even though the workflow application may shutdown and restart, the workflow status containing the DRMAA job identifiers is constantly persisted. An underlying DRMAA implementation needs to support this application behaviour. Therefore, DRMAA libraries are meanwhile free to accept job identifiers in the control routines that were not created during the active session. The bulk control mechanisms still act on the jobs of the current session only.

Compared to the API semantics of DRMAA, SAGA defines an explicit session concept, which supports multiple separated operational scopes at the same time. Different running sessions are promised to be shielded from each other, in terms of security and job objects. SAGA provides in this part a richer functionality than DRMAA, but also puts a higher burden on the adopters. The support for different security models and multiple sessions mainly arose from the resource access scenarios for a Grid environment, where heterogeneous resources must be accessible by a central API at the same time. In contrast, DRMAA arose from the background of cluster systems and local installations, where the parallel access to multiple resources is no major issue.

OGSA-BES relies on the session concept already provided by the OGSA (Banks, 2005) *Web Service Resource Framework* (WSRF) technology. OGSA-BES therefore can assume the existence of a user-specific session, including application of authentication steps. The exhaustive modularity of the OGSA-related specifications shows its advantage here, since it simplifies the layout of every single specification. On the other hand, OGSA-BES cannot be seen as self-contained standard for job submission. Applications need to consider the particular implementation of the interface in order to achieve interoperability. The *OGSA High-Performance Computing Profile* (OGSA-HPCP) working group therefore defined a basic composition of standards, which form a common base for interoperable job submission based on Web service interfaces.

3.1 Job templates

The description of a job to be submitted to a DRMS is typically done via a *job template.* In DRMAA, it 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. They are guaranteed to be always supported, ensuring portability. 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. So far, most existing implementations for cluster and Grid systems ignored 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 also not bound to a particular job execution and can therefore be reused for multiple submissions. The specification defines the template to be evaluated at submission time; therefore all template attribute setter functions only consider errors like incorrect attribute name, invalid value format or conflicting setting.

In direct comparison, JSDL supports a much broader range of attributes than the DRMAA job template, for example, for the definition of resource requirements or data-staging rules. Due to this exhaustive consideration of possible job properties in an XML format, all OGSA-related standards meanwhile rely on the JSDL data model for job descriptions. Even the SAGA specification derives its *job-description* structure from the JSDL specification.

The reusability and portability of one particular job description is a major driver in the JSDL template layout.

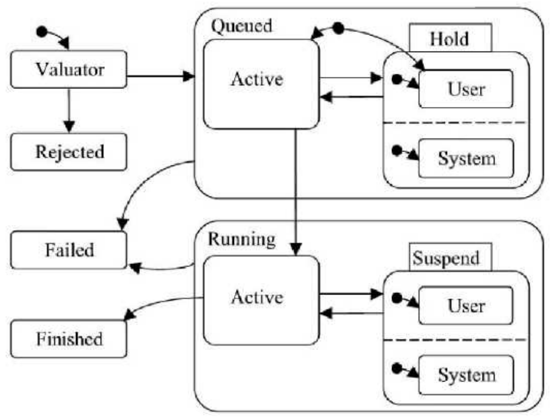
For this reason, submission-specific information like job start time, latest job termination time and initial status at submission are not represented in these documents. This particular design decision of the JSDL specification is one of the major reasons for DRMAA to stick with an own job description model. An additional argument is that the repeated investigation of DRMAA job descriptions in real applications leads to only a few suggested extensions. When compared, the typical DRM systems still only share a small set of common job description properties. However, non-technical demands from the user community meanwhile lead to the decision of additionally supporting JSDL in an upcoming version of the API. Implementations like the GridWay framework already realised the according mapping of JSDL elements to DRMAA job templates, mainly by ignoring non-transferable attribute values of the original job description. It remains an open research question if such an approach alters some of the job descriptions in a non-acceptable way.

3.2 Job submission and monitoring

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. Job 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 created by an explicit drmaa_control() call, or by a submission as hold job. Both cases are represented by the 'user on hold' job state. A held job can further be caused by the DRM system itself or by a combination of both factors, which is explicitly considered in the state model. All types of held jobs are explicitly released with another drmaa_control() call.

Figure 1 DRMAA job state transition



An active job in the status class 'Running' can either be done executing or it can be put in a suspended state. The latter might be explicitly triggered by the use of drmaa_control() or by the system itself.

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 temporal effects in idle-time environments, like Condor, or wide-area Grid environments, like GridWay.

In comparison, SAGA and OGSA-BES provide simpler job state models, but with the possibility to let the implementation define extended states. DRMAA specification provides a fixed set of possible states. Not all states are mandated to be used by a DRMAA implementation for a particular DRM system.

SAGA and OGSA-BES define a *pending* state, a *running* state, a *finished* state, a *terminated* state and a *failed* state. Those basic states can be specialised in OGSA by profiles, allowing more complex clients that are still interoperable with different service implementations.

JSDL only handles the description of jobs to be submitted and has therefore no job state model as part of the specification.

3.3 Job control

The state of a submitted job in DRMAA 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 synchronise a set of jobs, waiting for all of them to finish before continuing further with drmaa_synchronise(). 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 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. This feature is very useful for a large number of jobs to reduce memory requirements.

drmaa_wait 0 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.

Listing 1 DRMAA C Example

```
#include <stdio.h>
#include "drmaa.h"

int main(int argc, char *argv[]) {
  drmaa_job_template_t *jt;
  drmaa_attr_values_t  *rusage;
  drmaa_job_ids_t *ids;
  char actid[DRMAA_ATTR_BUFFER];
  const char *args[2]={"-i", NULL};
  int result, stat, exited;

  drmaa_init(NULL, NULL, 0);
  drmaa_allocate_job_template(&jt, NULL, 0);
  drmaa_set_attribute(jt, DRMAA_REMOTE_COMMAND,
    "/bin/hostname", NULL, 0);
  drmaa_set_vector_attribute(jt, DRMAA_V_ARGV,
    args, NULL, 0);
  /* Start bulk jobs */
  drmaa_run_bulk_jobs(&ids, jt, 1, 4, 1, NULL, 0);
  do {
    result=drmaa_get_next_job_id(ids, actid,
      DRMAA_JOBNAME_BUFFER);
    if (result==DRMAA_ERRNO_SUCCESS) {
      drmaa_wait(actid, NULL,
        DRMAA_JOBNAME_BUFFER, &stat,
        DRMAA_TIMEOUT_WAIT_FOREVER, &rusage,
        NULL, 0);
      drmaa_wifexited(&exited, stat, NULL, 0);
      if (exited) {
        drmaa_wexitstatus(&stat,stat,NULL,0);
        printf("Job␣%s␣-␣exit␣status␣%i\n",
               actid, stat);
    }}
  } while (result!=DRMAA_ERRNO_NO_MORE_ELEMENTS);
  /* ... cleanup functions ... */
  return 0;
}
```

The status code resulting from drmaa_wait 0 is used in a series of functions, in order to provide more detailed information about job termination. The overall semantic of this approach is modelled after POSIX wait() and the related pre-processor macros (e.g. WIFEXITED).

Listing 1 shows an example application for the C language mapping of the DRMAA API. After initialising the session with the DRM system through the drmaa_init() call, a job template is allocated by a C-specific DRMAA function. The application binary and the command line argument attributes of the job template are modified, before the job template is used to submit a bulk job. Timing and queuing issues for the bulk job are part of the product-specific DRM system configuration, and cannot be influenced by the DRMAA abstraction layer. With the resulting set of job identifiers (drmaa_job_ids_t *ids) and another C-binding specific helper function, the program can now iterate over the submitted jobs, wait for their ending sequentially and check their exit status. The relevant cleanup functions for job template deallocation and session closing are omitted in the example.

In direct comparison of the job control functionalities, SAGA explicitly incorporates the job control semantics from the DRMAA specification and has therefore the same set of functions. It allows the cancellation, suspending or signalling of jobs. OGSA-BES relies on the representation of jobs WSRF endpoints and allows control activities through the according resource properties.

3.4 *Auxiliary routines*

Few routines in the DRMAA API provide necessary functions for establishing a session with a DRMAA library. drmaa_get_contact() provides the list of supported contact strings, when called before drmaa_init(). This method therefore supports the implementation of one DRMAA library for multiple DRM systems. The application can use one of the offered contact strings in the drmaa_init() call, or the default provider is used. The concept is comparable to the ODBC provider model (Geiger, 1995). After the drmaa_init() call, the same routine call produces the currently used connection string.

The drmaa_version() routine provides the version number of the supported DRMAA specification, which enables applications to check their compatibility to the loaded DRMAA library. The drmaa_get_DRM_system() and drmaa_get_DRMAA_implementation() methods provide information about the specific library implementation, which is mainly needed for the portable usage of implementation-specific job template attributes.

All auxiliary functions are independent from the DRMAA session handling, and can therefore be used before a drmaa_init() or after a drmaa_exit() call.

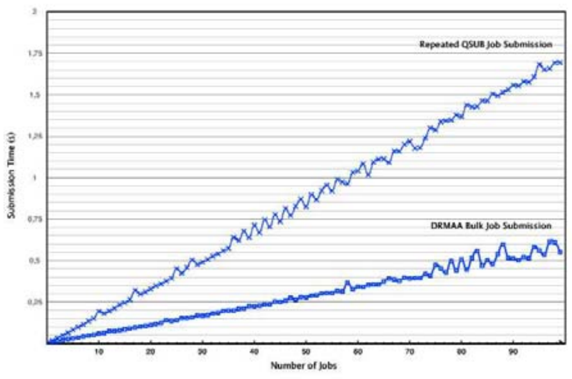
In comparison, SAGA offers a much richer amount of other functions needed for an interaction with DRM systems. This includes file and replica management, data streaming and detailed monitoring operations. SAGA can therefore be seen as the 'one-fits-all' solution for applications accessing a DRM system. There are existing implementations of SAGA relying on DRMAA for the job submission and monitoring part. Different APIs must cover other parts of the SAGA functionality.

OGSA-BES concentrates, like DRMAA, exclusively on the job submission and control aspect. It could therefore be seen as the 'WSRF-version' of the DRMAA interface, but with a strict focus on the overall OGSA concepts. Due to the usage of Web service technology, the specification does not consider the local usage of the API, which implies questions of language bindings, performance optimisation and concurrent access. Both specifications therefore have their unique position in the OGF standards landscape.

3.5 *Performance*

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. Figure 2 shows one example for the job submission performance in Sun N1GE. The test environment was a cluster of five PIII nodes with 1 GB of RAM and Sun N1GE version 6.0 installed. An increasing number of jobs were submitted to the local Sun N1GE scheduler with the drmaa_run_bulk_job function on the one hand and a repeated qsub command call on the other hand. For each number of jobs, the submission experiment was repeated 10 times, with a removal of the submitted jobs before each cycle. Figure 2 compares the measured average time for each number of jobs. The figure clearly shows that the direct interaction of Sun N1GE's DRMAA library with the scheduler outperforms the usual repeated usage of command line tools in shell scripts. This is mainly reasoned by the careful reuse of existing network connections to scheduling and queuing servers in the library implementation.

Figure 2 Job submission in Sun Grid Engine (see online version for colours)



It must be noted that this example result is only valid for the particular DRMAA implementation in Sun N1GE. Other implementations of DRMAA (e.g. Condor) rely on reusing the command line utilities of the DRM system due to non-available wire format specifications for the according system. These systems naturally show a similar performance to traditional batch submission scripts.

4 Standardisation experiences

The following section describes the experiences and design decisions of the DRMAA working group. The presented issues have mostly caused lengthy discussions or increased implementation efforts for the adopters, and should therefore be considered by other standardisation efforts in the future. All of them were either resolved or delayed to a later update of the DRMAA specification.

It must be noted that the technical background of the working group members mainly influenced the presented DRMAA design principles and solution strategies. Most DRMAA group members and contributors have an industrial background, leading to standardisation decisions mainly driven by applicability and acceptance for users and vendors. On the negative side, this basically practice-oriented view sometimes hinders the creation of comprehensive complete solutions for a problem domain.

One counter example is the JSDL specification, where an academia-driven working group spent great efforts on building an exhaustive type model. Adoptions in the field meanwhile show that – as intended – most implementations do not cover the whole specification, but instead rely on the free choice of JSDL attributes to be implemented. Therefore, both 'standardisation styles' have been shown to work in the past.

4.1 Implementation complexity

DRMAA was designed with the overall goal of having a small and easy to understand API, which accounts for most application developer needs. Even though the set of available implementations shows the success of this approach, there are still challenging aspects, one example being multithreading.

The specification basically demands the API implementation to be thread-safe. Together with the goal of satisfactory performance this 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 behaviour in these cases. Most updates for the *Grid recommendation* version of DRMAA were related to such issues.

One example: Since session information could change during a blocking call (e.g. job termination with drmaa_control() in parallel with drmaa_wait() operation), both functions must define how the multithreading case is handled. Here, 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.

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

4.2 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 is the default API for programming in this system. The product-quality reference implementation helped to identify real-world issues with the specification and acted as major driver for the API finalisation.

Nevertheless, it must be noted that the existence of preliminary implementations provided a high barrier for major conception changes later. While this can be seen as an advantage in terms of a stable interface definition, it hinders adoption of new or improved concepts, such as monitoring support.

The existence of a reference implementation also indirectly leads 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. It helped to demonstrate the practical interoperability of the existing implementations for the OGF. The test suite currently contains around 4000 lines of C code with 13 complex tests, especially for the DRMAA specification multithreading access rules.

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 domain of automated software testing. It remains an open research question if behavioural descriptions for software functionality can be formulated in a way to allow automated API compliance testing.

DRMAA test suite now mainly provides a valuable source of information for DRMAA library developers. Multiple specification improvements arose from issues identified through test results with different DRMAA implementations.

4.3 Reuse of existing standards

During the development of the specification, it was attempted to apply the existing concepts and wording from POSIX (1003.1 and 1003.2d), 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 processing of termination information.

The resulting set of DRMAA status functions now must be called in a specific order, e.g. first asking if the job was signalled and then querying for the core dump availability. Multiple language binding authors, as well as several DRMAA users, expressed their wish to favour 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. The POSIX semantics therefore have not shown to be a helpful solution in terms of user acceptance.

Beside the POSIX example, the continuous discussion of integrating DRMAA and JSDL shows a similar outcome. In theory, DRMAA job template structures could be replaced by JSDL-compliant descriptions. As already discussed, this was not driven by missing attributes in the DRMAA job template format, but was driven by the wish of applying existing standardisation results for the same kind of problem. A closer look on the intended usage scenarios for both specifications clearly shows the missing benefit from such a merger. While JSDL is completely XML-based and therefore intended as document format or SOAP message payload, DRMAA job template structures are part of an API definition, and therefore must be represented in the according programming language.

4.4 Time definitions

DRMAA supports the notion of partial timestamps, which allow the incomplete specification of the 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 is intended to enable the reusage of incomplete timestamp definitions for multiple job runs.

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

Even though the concept of partial timestamps might be helpful for DRMAA users and portal developers, it turned out to be a difficulty for several language bindings. The C language binding simply adopted the formulation of the partial timestamps as string variable. In contrast, modern object-oriented development in Java and C# needed a mapping of partial timestamp information to class library types.

4.5 Monitoring features

Several DRMAA users criticised the lack of support for resource-related monitoring data, like cluster load information, hardware status or operating system type. As one major problem, the provisioning of such status information in a standardised API would demand a common information model for the monitoring parameters. For example, the specification of the *CPU type* or *disk quota* parameters would need to consider heterogeneous DRM products, hardware platforms and operating systems. Even though some standards (like Common Information Model; CIM (Bumpus et al., 2000) and JSDL) provide such unified information model, it would be 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 therefore need to consider this recent work for the extension of the API.

4.6 Job state semantics

DRMAA so far does not provide an extensive description of the 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 a particular DRM system.

One example is the *suspended* job state. The current DRMAA implementation for the Condor system (Litzkow et al., 1988) works in the *vanilla universe*, which does not support the suspension and resuming of running 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 behaviour does not break DRMAA compliance, since the job state ‘SUSPENDED’ is not further described. But for the end user, the visible effect is a rescheduling of the job 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 with this particular problem. 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 behaviours with different DRM systems. So far, the DRMAA group favours the second approach. Recent discussions identified a third approach, which would mark the critical job states as optional. Implementations would then be free to not support some of the DRMAA job states. This can be seen as a similar strategy to the extensibility model of the OGSA-BES job states.

4.7 *Keeping the discussion history*

A significant amount of time was spent by the DRMAA working group on discussions about non-obvious design decisions from the past. Due to the long lifetime of the group, some of the initial API design needed to be constantly requestioned and reevaluated. The typical sources of the discussion history, like phone conference meeting minutes and mailing list archives, turned out to being of less use than expected. Since OGF does not dictate mandatory document histories or rationale sections, users and adopters sometimes need to clarify API semantics by communication with the document authors.

Future improvements of the OGF standardisation regulations should consider this issue as an important factor for both specification adopters and users. A popular example for such an approach is the work in the Open Group consortium for the Single Unix specification. This API always refers in its description to a *Rationale (XRAT)* section for explaining the different design decisions.

4.8 *Conclusions*

The presented excerpt of standardisation experiences underlines the need for some well-known best practices of the software development work, but also shows non-obvious aspects of the OGF standardisation work.

From our experience, the working groups heading towards the API definition should consider a concurrent application access to the interface from the beginning, by specifying the implications on the functional behaviour. Existing modelling techniques for concurrent processes could be applied to the API definition, in order to check for possible mutual exclusion situations or race conditions. So far, the DRMAA group gained such experiences only from flaws in the different implementations. Future work should try to identify such conditions in a more formal way.

Distributed systems standardisation work needs to clearly distinguish between the remote and local API standards, since this influences the design of session and fault-handling mechanisms. Also, the reapplication of the existing standardisation results does not automatically provide better results for the specification to be developed.

Multiple implementations by different organisations, as already demanded by the OGF regulations for *Grid recommendations*, have shown to be a major information source regarding portability issues. Even with a continuous review process over many years and a small amount of fundamental changes, new issues showed up all the time. The portability problems increased even more with the provisioning of multiple language bindings for the DRMAA API. The working group therefore created an Interface Definition Language (IDL)-based specification of the original API, in order to solve portability issues regardless of the destination language. The following section describes this new specification.

5 IDL-based specification

Even though DRMAA efforts started with the idea of a language-independent specification, the *Grid recommendation* document was mainly written with a procedural C-language slant. With the ongoing development and adoption efforts, 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 these two independently developed object-oriented language bindings, an in-depth comparison identified general issues for object-oriented mappings of the API. 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 realisation of partial timestamps and job templates with the available language constructs.

To keep and maintain the newly derived DRMAA semantics for OO-environments, the group decided to base future versions of the language-independent DRMAA specification on the *CORBA Interface Definition Language* (Object Management Group, 2004). Language-binding documents should now only need to define a mapping between DRMAA IDL constructs and their specific language constructs, instead of creating own syntactical rules for the interface. An example for such mappings is shown in Table 1. The Java mapping was developed by Daniel Templeton (Templeton et al., 2006) is already adopted by multiple implementations and is about to be standardised as an OGF document during 2008. The .NET binding, a newer development, should also be standardised in 2008. The WSDL binding is also a new proposal, mainly to demonstrate that the flexibility of the IDL description approach not only holds for pure programming languages, but also for other interface description languages. The standardisation of this mapping is under discussion at the time of writing.

The chosen API standardisation approach is also used in other popular specifications, for example in the W3C Document Object Model. The IDL ensures a consistent description of API signatures and provides the functional semantics of DRMAA operations, whereby the binding author only needs to concentrate on their language-specific mapping issues. The reuse of OMG IDL language mapping specifications is omitted, since these mapping rules rely on CORBA runtime environment mechanisms. Instead, each DRMAA language binding should define its own rules, which are most appropriate to the according environment.

Table 1 Examples for DRMAA IDL language binding

IDL keyword	*Java*	*WSDL*	*.NET*
module	package	<wsdl:definitions>	namespace
interface	ublic interface	<wsdl:portType>	public interface
enum	int constants	<xsd: simpleType> + <xsd:restriction>	public enum
valuetype	public class	<xsd:complexType>	public class
attribute	Java Bean attribute	<xsd:sequence>	get / set operations
exception	java.lang.Exception derivation	<wsdl:message>	ApplicationException derivation
long	int	<xsd:int>	int
boolean	boolean	<xsd:boolean>	bool
string	java.lang.String	<xsd:string>	string
Const	public static final	not in XML schema	public const
valuetype Dictionary	j ava.ut il.Map	<xsd:sequence>	IDictionary
valuetype StringList	java.util.Set	<xsd:list>	IList
valuetype OrderedStringList	java.util.List	<xsd:sequence>	IList
valuetype TimeAmount	long	<xsd:int>	long
native PartialTimestamp	class PartialTimestamp	<xsd:string>	class PartialTimestamp

In the IDL specification, job templates are represented as interfaces with according member attributes. As one interesting challenge, optional attributes needed to be reflected in the IDL-API design. All attributes must be available in the job template interface layout, even though they might not be usable for a particular implementation. This ensures the source code portability of a DRMAA-based application. 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 also 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 this case, the groupings can be mapped to the respective language exception hierarchies. Java is one of the languages that utilise this approach.

The original DRMAA API definition 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 the modern object-oriented languages, the delete() operation for job templates remains an explicit function in the IDL definition. This design decision reflects the missing support for guaranteed object finaliser execution in some of the object-oriented languages, which is needed for maybe mandatory cleanup operations in the DRMS.

Listing 2 shows the Java version of the C example program discussed in Section 3.3. The utilised Java binding was derived from the IDL definition as described in the table (Templeton et al., 2006). In the main() function, an object for the Session interface is indirectly obtained from a factory object. This is a typical Java approach for supporting the separation of API implementation and Service Provider Interface (SPI) implementation. The API implementation contains the generic DRMAA functionality, such as partial timestamp handling, job template parsing or job state management. The SPI implementation is added by each DRM vendor and interacts with the underlying cluster or Grid system. In the case of Sun N1GE, this interaction is based on native calls to the DRMAA C implementation.

As shown in the example program, job template objects are created by a function of the Session interface. This expresses the tight binding of session and job template in DRMAA and enables easy cleanup operations (even with an underlying C library) in the Session implementation.

The job identifiers for the created jobs are expressed as one collection object, containing an iteratable list of string identifiers. This design decision – against a list of job objects – was mainly driven by the close relation of DRMAA to DRM control commands, where jobs are always represented as strings. Job objects would also introduce unnecessary remote calls in a remote procedure call scenario, as with a DRMAA WSDL binding. The IDL specification therefore sticks with the job identification by DRM-specific strings, and this design decision is therefore also reflected in the Java binding.

As described earlier, the job information retrieval was adopted to a typical object-oriented approach. Each session.wait() call results in a JobInfo object, which has the equivalent status checking functions to the C version of the DRMAA specification.

6 DRMAA usage

DRMAA is, at the current time, used as job submission and monitoring API in several cluster and Grid applications. Examples are meta-schedulers or integrated Data Grid solutions. Many of the known applications, which are driven by DRMAA API provisioning, primarily use Sun N1GE.

Listing 2 DRMAA Java example

```
package drmaa;
import java.util.*;
import org.ggf.drmaa.*;

public class JDrmaa {
  public static void main(String[] args)
    throws DrmaaException {
    Session session=
      SessionFactory.getFactory().getSession();
    session.init("");
    JobTemplate jt=session.createJobTemplate();
    jt.setRemoteCommand("/bin/hostname");
    jt.setArgs(new String[]{"-i"});
    Iterator ids=
      session.runBulkJobs(jt, 1, 4, 1).iterator();
    while (ids.hasNext()) {
      JobInfo info=session.wait((String)ids.next()
                    Session.TIMEOUT_WAIT_FOREVER);
      if (info.hasExited()) {
        System.out.println("Job␣"+info.getJobId()+
                           "-␣exit␣status␣"+
                           info.getExitStatus());
    }}
    /* ... cleanup functions ... */
}}
```

As one example, the OpenDSP project[1] from the Poznan Supercomputing and Networking Center developed a Web Service-based interface for multiuser access to DRMAA-enabled DRM systems. The system combines the low-level DRMAA API with OGSA-BES SOAP interface adding an infrastructure for security, accounting and administrator control. JSDL is used as job description language and currently a few of its elements are supported (including data staging with Web Services Attachments) by mapping to the corresponding DRMAA attributes. The upcoming version is going to extend the remote interface by adding remote advance reservation capabilities. 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), InteliGrid project (remote access to engineering and construction applications) or European BREIN project.

Another example is the *Target System Interface Framework* for UNICORE (Erwin and Snelling, 2001), which is using DRMAA as front-end to DRM systems (Riedel et al., 2006). EGEE[AQ1] relies on DRMAA for integration issues. The MOAB[AQ1] scheduler can control DRM systems that implement a DRMAA interface. Wolfram Research offers a cluster integration package for Mathematica, which also relies on DRMAA for job submission. Latest research on a Grid operating system in the XtreemOS project uses DRMAA as basic API for execution resource access. Since most usage scenarios have not yet been described in scientific publications, the latest list of the use cases and DRMAA implementations is maintained on the DRMAA home page.[2]

AQ1: Based on the internet source. website addresses have been added in references 'Object Management Group (2004), Templeton (2006) and Tuecke (2004)'. Please check for the correctness.

7 Conclusion

DRMAA is an approved specification for a small unified 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 and is increasingly supported in various programming languages. Several industrial customers and research projects use DRMAA in their real-world productive environments.

In comparison to other related specifications from OGF, the DRMAA specification has shown to be the least common denominator for interoperable job submission. It lacks several relevant features, such as data-staging features or improved job monitoring, in order to keep interoperability with a maximum set of DRM systems. The SAGA specification has shown to be the richer API, but still lacks the matureness and adoption rate of DRMAA in commercial products. One promising approach for future work is the implementation of SAGA libraries based on DRMAA, which was already demonstrated by several research projects.

OGSA-BES confirms to the technological and architectural framework provided by the OGF OGSA standardisation effort. An increasing number of vendors support OGSA-compliant interfaces in their products. With the current performance and stability status of SOAP-based interfaces, DRMAA can still fulfil the role of a small, efficient and truly interoperable interface API to multiple DRM systems. With the increasing maturity of SOAP technologies in the future, this role might also be fulfilled by XML-based specifications.

In our standardisation work, we experienced several non-obvious issues, which should be considered by other researchers in the standardisation area. The independent parallel development of different language bindings (‘n-version standardisation’) can identify general issues with a root specification. Even with a comparatively lightweight API, the definition of implementation behaviour for concurrent and non-reliable Grid environments still demands continuous in-depth analysis and practical implementation experience. The combination of academia and industry people in one working group has shown to be beneficial for both theoretical completeness and practical applicability. A reference implementation by an industrial partner clearly acts as a major driver for the community acceptance and additional implementations of a standard.

The future versions of the DRMAA specification will tackle necessary 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

Allen, G., Davis, K., Dolkas, K., Doulamis, N., Goodale, T., Kielmann, T., Merzky, A., Nabrzyski, J., Pukacki, J., Radke, T., Russell, M., Seidel, E., Shalf, J. and Taylor, I. (2003) ‘Enabling applications on the grid: a gridlab overview’, *International Journal of High Performance Computing Applications*, Vol. 4, No. 17 pp.449–466.

Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, S., Pulsipher, D. and Savva, A. (2005) *Job submission description language* (JSDL) Specification v1.0 (GFD.56).

Banks, T. (2005) *Web Services Resource Framework (WSRF) – Primer*, Committee Draft 01, OASIS Open.

Bumpus, W., Schweitzer, J.W. and Thompson, P. (2000) *Common Information Model*, John Wiley & Sons Inc.

Catlett, C. (2001). *Global Grid Forum Documents and Recommendations: Process and Requirements (GFD-C.1)*.

Erwin, D.W. and Snelling, D.F. (2001) ‘UNICORE: a grid computing environment’, *Lecture Notes in Computer Science*, Vol. 2150, p.825.

Foster, I. and Kesselman, C. (1997) ‘Globus: a metacom-puting infrastructure toolkit’, *The International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11, No. 2, pp.115–128.

Geiger, K. (1995) *Inside ODBC*, Microsoft Press Redmond, WA, USA.

Goodale, T., Jha, S., Kaiser, H., Kielmann, T., Kleijer, P., Merzky, A., Shalf, J. and Smith, C. (2007) *A Simple API for Grid Applications (Saga) Version 1.0*. Available online at: http://saga.cct.lsu.edu/

Grimshaw, A., Newhouse, S., Pulsipher, D. and Morgan, M. (2006) *OGSA basic execution service version 1.0*. Available online at: https://forge.gridforum.org/projects/ogsa-bes-wg/

Herrera, J., Montero, R., Huedo, E. and Llorente, I. (2004) DRMAA Implementation within the GridWay Framework, *Application Developers and Users Research Group Meeting, Global Grid Forum 12, Brussels, Belgium.*

Lee, W., McGough, S., Newhouse, S. and Darlington, J. (2004) ‘A standard based approach to job submission through web services’, *Proceedings of the UK e-Science All Hands Meeting*, UK EPSRC, Nottingham, UK, pp.901–905.

Litzkow, M., Livny, M. and Mutka, M. (1988) ‘Condor – A hunter of idle workstations’, *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp.104–111.

Object Management Group (2004) *Common Object Request Broker Architecture: Core Specification*, Revision 3.0.3 (CORBA v3.0.3), Prentice Hall Professional Technical Reference. Available online at: www.omg.org/technology/documents/formalcorba_iiop.htm

Riedel, M., Menday, R., Streit, A. and Bala, P. (2006) ‘A DRMAA-based target system interface framework for UNICORE’, *Proceedings of the 12th International Conference on Parallel and Distributed Systems (IC-PADS)*, IEEE Computer Society, Washington, DC, USA, pp.133–138.

Sinnott, R., Jiang, J., Watt, J. and Ajayi, O. (2006) ‘Shibboleth-based access to and usage of grid resources’, *7th IEEE/ACM International Conference on Grid Computing*, pp.136–143.

Templeton, D., Tröger, P., Brobst, R., Haas, A., Rajic, H. and Tollefsrud, J. (2006) *Distributed Resource Management Application API Java(TM) Language Bindings 1.0. Open Grid Forum*. Available online at: http://forge.gridforum.org/sf/docman/do/downloadDocument/projects.drmaa-wg/docman.root.recommendations/doc14447;jsessionid=C3C9F73BADFAA728D78611F440E4D5EE

Tuecke, S., Welch, V., Engert, D., Pearlman, L. and Thompson, M. (2004) *Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile*, RFC 3820 (Proposed Standard). Available online at: http://tools.ietf.org/html/rfc3820

von Laszewski, G., Foster, I., Gawor, J. and Lane, P. (2001) ‘A Java commodity grid kit’, *Concurrency and Computation: Practice and Experience*, Vol. 13, Nos. 8–9, pp.643–662.

Notes

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

2 http://www. drmaa.org/

3 drmaa-wg@ogf.org