GWD-R
DRMAA-WG
drmaa-wg@ogf.org

Peter Tröger, Hasso-Plattner-Institute
(Corresponding Author)
Roger Brobst, Cadence Design Systems
Daniel Gruber, Univa
Mariusz Mamoński, PSNC
Daniel Templeton, Cloudera

June 2011

Distributed Resource Management Application API Version 2 (DRMAA) - Draft 7

Status of This Document

Group Working Draft Recommendation (GWD-R)

(See footnote)[1]

Obsoletes

This document obsoletes GFD-R.022 [7], GFD-R-P.130 [9], and GWD-R.133 [8].

Copyright Notice

Trademark

All company, product or service names referenced in this document are used for identification purposes only and may be trademarks of their respective owners.

Abstract

This document describes the *Distributed Resource Management Application API Version 2 (DRMAA)*, which provides a generalized API to *Distributed Resource Management (DRM)* systems in order to facilitate the development of portable application programs and high-level libraries for such systems. DRMAA defines interfaces for a tightly coupled, but still portable access by abstracting the fundamental functions available in the majority of DRM systems. The scope is limited to job submission, job control, reservation management, and retrieval of job and machine monitoring information.

This document acts as root specification for the abstract API concepts and the behavioral rules that must be fulfilled by a DRMAA-compliant implementation. The programming language representation of the abstract API concepts must be formulated by a separate *language binding specification* derived from this document.

The intended audience for this specification are DRMAA language binding designers, DRM system vendors, high-level API designers and meta-scheduler architects. End users are expected to rely on product-specific documentation for the DRMAA API implementation in their particular programming language.

[1] This is the non-normative annotated version of the specification with line numbers. It includes historical information concerning the content and why features were included or discarded by the working group. It also emphasizes the consequences of some aspects that may not be immediately apparent. This document in only intended for internal working group discussions.

Contents

1 Introduction

This document describes the *Distributed Resource Management Application API Version 2 (DRMAA)* interface semantics in a generalized way by using the *OMG Interface Definition Language (IDL)* [4] syntax for a language-agnostic description. Based on this abstract specification, *language binding* standards have to be designed that map the described concepts into a library interface for a particular programming language (e.g. C, Java, Python). While this document has the responsibility to ensure consistent API semantics over all possible DRMAA implementations, the language binding has the responsibility to ensure source-code portability for DRMAA applications on different DRM systems.

An effort has been made to choose an API layout that is not unique to a particular language. However, in some cases, various languages disagree over some points. In those cases, the most meritous approach was taken, irrespective of language.

There are other relevant OGF standards in the area of job submission and monitoring. An in-depth comparison and positioning of the obsoleted first version of the DRMAA [8] specification was provided by another publication [10]. This document was created in close collaboration with the OGF SAGA and the OGF OCCI working group.

The DRMAA specification is based on the following stakeholders:

- *Distributed resource management system / DRM system / DRMS*: Any system that supports the concept of distributing computational jobs on execution resources through the help of a central scheduling entity. Examples are multi-processor systems controlled by a operating system scheduler, cluster systems with multiple machines controlled by a central scheduler software, grid systems, or cloud systems with a job concept.
- *DRMAA implementation, DRMAA library*: The implementation of a DRMAA language binding specification with the functional semantics described in this document. The resulting artifact is expected to be a library that is deployed together with the DRM system that is wrapped by the particular implementation.
- *(DRMAA-based) application*: Software that utilizes the DRMAA implementation for gaining access to one or multiple DRM systems in a standardized way.
- *Submission host*: An execution resource in the DRM system that runs the DRMAA-based application. A submission host MAY also be able to act as execution host.
- *Execution host*: An execution resource in the DRM system that can run a job submitted through the DRMAA implementation.

Provide mapping to GLUE (GFD.147)

1.1 Notational Conventions

In this document, IDL language elements and definitions are represented in a `fixed-width` font.

The key words “MUST” “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” are to be interpreted as described in RFC 2119 [1].

Memory quantities are expressed in *kilobyte (KB)*. 1 kilobyte equals 1024 bytes.

Parts of the specification which are normative for derived language binding specifications only are graphically marked as shaded box.

(See footnote)[2] .

1.2 Language Bindings

A language binding specification derived from this document MUST define a mapping between the IDL constructs and programming language constructs, with focus on source code portability for the resulting DRMAA-based applications.

A language binding SHOULD NOT rely completely on the OMG language mapping standards available for many programming languages, since they have a huge overhead of irrelevant CORBA-related mapping rules. Therefore, language binding authors must carefully decide if a binding decision reflects a natural and simple mapping of the intended purpose for the DRMAA interfaces. The binding SHOULD reuse OMG value type mappings (e.g. IDL `long long` to Java `long`), and SHOULD define custom mappings for the other types. The language binding MUST use the described concept mapping in a consistent manner for its overal API layout.

Due to the usage of IDL, all method groups for a particular purpose (e.g. job control) are described in terms of interfaces, and not classes. The mapping to a class concept depends on the specific language-mapping rules.

It may be the case that IDL constructs do not map directly to any language construct. In this case it MUST be ensured that the chosen mapping retains the intended semantic of the DRMAA interface definition.

Access to scalar attributes (`string`, `boolean`, `long`) MUST operate in a pass-by-value mode. A language binding must ensure that this behavior is always fulfilled. For non-scalar attributes, the language binding MUST specify a consistent access strategy for all these attributes – either pass-by-value or pass-by-reference – according to the use cases of language binding implementations.

This specification tries to consider the possibility of a Remote Procedure Call (RPC) scenario in a DRMAA-conformant language mapping. It SHOULD therefore be ensured that the programming language type for an IDL `struct` definition supports the serialization and comparison of instances. These capabilities should be accomplished through whatever mechanism is most natural for the programming language.

A language binding MUST define a way to declare an invalid value (`UNSET`). In case, a definition per data type needs to be provided. Evaluating an `UNSET` boolean value MUST result in a negative result, e.g. for `JobTemplate::emailOnStarted`.

(See footnote)[3]

[2] The usage of kikibyte as memory quantity unit, as well as the usage of bytes as in JSDL, was rejected by the group (conf call Apr. 13th 2011)

[3] The concept of a UNSET value was decided on a conf call (Aug 25th 2010). Boolean in C can use custom enumeration (TRUE, FALSE, INVALID) or pointer to static values. A numerical UNSET in C should use a magic number, since all long attributes are unsigned, it could be MIN_INT. With Python, just use `None`. For Java, Dan has an idea.

1.3 Slots and Queues

DRMAA supports the notion of slots and queues as resources of a DRM system. A DRMAA application can request them in advance reservation and job submission. However, slots and queues SHALL be opaque concepts from the viewpoint of a DRMAA implementation, meaning that the requirements given by the application are just passed through to the DRM system. This is reasoned by the large variation in interpreting that concepts in the different DRM systems, which makes it impossible to define a common understanding on the level of the DRMAA API.

(See footnote)[4]

1.4 Job Categories

DRMAA facilitates writing DRM-enabled applications even though the deployment properties, in particular the configuration of the DRMS, cannot be known in advance. This is realized by a set of standardized attributes that can be specified for job submission or advance reservation.

One of these attributes is the job category, which allows to give an indication about the nature of the job at execution time. Examples are parallel MPI jobs, OpenMP jobs, jobs targeting specific accelerator hardware, or jobs demanding managed runtime environments (e.g. Java). For bulk job submissions, the category is expected to be valid for each of the jobs created.

Job categories typically map to site-specific reservation or submission options. Each category expresses a particular type of job execution that demands site-specific configuration such as example path settings, environment variables, or application starters such as MPIRUN. This mapping SHOULD take place at submission time of the job or advance reservation.

A non-normative recommendation of category names is maintained at:

`http://www.drmaa.org/jobcategories/`

Implementations SHOULD use the recommended names, if applicable. In case the name is not taken from the non-normative recommendation, it should be self-explanatory for the user so that she can understand the implications on job execution.

Implementations MAY provide a library configuration facility, which allows a site administrator to link job category names with specific product- and site-specific configuration options, such as submission wrapper shell scripts.

The order of precedence between the job category and other attributes is implementation-specific. It is RECOMMENDED to overrule job / reservation settings with a conflicting `jobCategory` setting.

(See footnote)[5]

1.5 Multithreading

High-level APIs such as SAGA [3] are expected to utilize DRMAA for asynchronous operations, based on the assumption that re-entrancy is supported by DRMAA implementations. For this reason, implementations

[4] As one example, queues can be either treated as representation of execution hosts (Sun Grid Engine) or as central waiting line located at the scheduler (LSF).

[5] There was a discussion on supporting the specification of multiple categories at the same time. Since this would put more burden on the implementation in terms of conflict resolving, we avoided that intentionally. This allows to map categories simply to some additional job submission command line arguments, similar to the old nativeSpecification thing.

SHOULD ensure the proper functioning of the library in case of re-entrant library calls. A DRMAA library SHOULD allow a multithreaded application to use DRMAA interfaces without any explicit synchronization among the application threads. DRMAA implementers should document their work as thread safe if they meet the above criteria. Providers of non-thread-safe DRMAA implementations should document all the interfaces that are thread unsafe and provide a list of interfaces and their dependencies on external thread unsafe routines.

2 Namespace

The DRMAA interfaces and structures are encapsulated by a naming scope, which avoids conflicts with other APIs used in the same application.

```
module DRMAA2 {
```

> Language binding authors MUST map the IDL module encapsulation to an according package or namespace concept and MAY change the module name according to programming language conventions.

(See footnote)[6]

3 Common Type Definitions

The DRMAA specification defines some custom types to express special value semantics not expressible in IDL.

```
  typedef sequence<string> OrderedStringList;
  typedef sequence<string> StringList;
  typedef sequence<Job> JobList;
  typedef sequence<QueueInfo> QueueInfoList;
  typedef sequence<MachineInfo> MachineInfoList;
  typedef sequence<SlotInfo> SlotInfoList;
  typedef sequence<Reservation> ReservationList;
  typedef sequence< sequence<string,2> > Dictionary;
  typedef string AbsoluteTime;
  typedef long long TimeAmount;
  native ZERO_TIME;
  native INFINITE_TIME;
  native NOW;
```

OrderedStringList: An unbounded list of strings, which supports element insertion, element deletion, and iteration over elements while keeping an element order.

StringList: An unbounded list of strings, without any demand on element order.

JobList: An unbounded list of `Job` instances, without any demand on element order.

JobArrayList: An unbounded list of `JobArray` instances, without any demand on element order.

[6] Comparison to DRMAA v1.0: The IDL module name was change to DRMAA2, in order to intentionally break backward compatibility of the interface.

QueueInfoList: An unbounded list of `QueueInfo` instances, without any demand on element order.

MachineInfoList: An unbounded list of `MachineInfo` instances, without any demand on element order.

SlotInfoList: An unbounded list of `SlotInfo` instances, without any demand on element order.

ReservationList: An unbounded list of `Reservation` instances, without any demand on element order.

Dictionary: An unbounded dictionary type for storing key-value pairs, without any demand on element order.

AbsoluteTime: Expression of a point in time, with a resolution at least to seconds.

TimeAmount: Expression of an amount of time, with a resolution at least to seconds.

ZERO_TIME: A constant value of type `TimeAmount` that expresses a zero amount of time.

INFINITE_TIME: A constant value of type `TimeAmount` that expresses an infinite amount of time.

NOW: A constant value of type `AbsoluteTime` that stands for the point in time at which it is evaluated by some function.

A language binding MUST replace these type definitions with semantically equal reference or value types in the according language. This may include the creation of new complex language types for one or more of the above concepts. The language binding MUST define a consistent mapping on module level, and a mechanism for obtaining the RFC822 string representation from a given `AbsoluteTime` or `TimeAmount` instance.

(See footnote)[7]

4 Enumerations

Language bindings SHOULD define numerical values for all DRMAA constants and enumeration members, in order to foster binary portability of DRMAA-based applications.

4.1 OperatingSystem enumeration

DRMAA supports the identification of an operating system installation on execution resources in the DRM system. The `OperatingSystem` enumeration is used as data type both in the advance reservation and the DRM system monitoring functionalities. It defines a set of standardized identifiers for operating system types. The list is a shortened version of the according CIM Schema [6]. It includes only operating systems that are supported by the majority of DRM systems available at the time of writing:

```
enum OperatingSystem {
  AIX, BSD, LINUX, HPUX, IRIX, MACOS, SUNOS, TRUE64, UNIXWARE, WIN,
  WINNT, OTHER_OS};
```

AIX: AIX Unix by IBM.

BSD: All operating system distributions based on the BSD kernel.

[7] The PartialTimestamp functionality from DRMAA 1.0 was completely removed. Absolute date and time values are now expressed as RFC822 conformant data items with stringification support (conf. call Mar 31st 2009). String list for job identifiers are replaced by Job object lists (F2F meeting July 2009)

LINUX: All operating system distributions based on the Linux kernel.

HPUX: HP-UX Unix by Hewlett-Packard.

IRIX: The IRIX operating system by SGI.

MACOS: The MAC OS X operating system by Apple.

SUNOS: SunOS or Solaris operating system by Sun / Oracle.

TRUE64: True64 Unix by Hewlett-Packard, or DEC Digital Unix, or DEC OSF/1 AXP.

UNIXWARE: UnixWare system by SCO group.

WIN: Windows 95, Windows 98, Windows ME.

WINNT: Microsoft Windows operating systems based on the NT kernel

OTHER_OS: An operating system type not specified in this list.

Implementations SHOULD NOT add new operating system identifiers to this enumeration, even if they are supported by the underlying DRM system.

The operating system information is only useful in conjunction with version information (see Section 10.1), which is also the reporting approach taken in most DRM systems. Examples:

- The Apple MacOS X operating system commonly denoted as “Snow Leopard” would be reported as “MACOS” with the version structure [“10”,“6”]
- The Microsoft Windows 7 operating system would be reported as “WINNT” with the version information [“6”,“1”], which is the internal version number reported by the Windows API.
- All Linux distributions would be reported as operating system type “LINUX” with the major revision of the kernel, such as [“2”,“6”].
- The Solaris operating system is reported as “SUNOS”, together with the internal version number, e.g. [“5”,“10”] for Solaris 10.

The DRMAA `OperatingSystem` enumeration can be mapped to other high-level APIs. Table 1 gives a non-normative set of examples.

DRMAA `OperatingSystem` value	JSDL `jsdl:OperatingSystemTypeEnumeration` value
HPUX	HPUX
LINUX	LINUX
IRIX	IRIX
TRUE64	Tru64_UNIX, OSF
MACOS	MACOS
SUNOS	SunOS, SOLARIS
WIN	WIN95, WIN98, Windows_R_Me
WINNT	WINNT, Windows_2000, Windows_XP
AIX	AIX
UNIXWARE	SCO_UnixWare, SCO_OpenServer
BSD	BSDUNIX, FreeBSD, NetBSD, OpenBSD
OTHER_OS	Other

Table 1: Mapping example for the DRMAA `OperatingSystem` enumeration

4.2 CpuArchitecture enumeration

DRMAA supports identifying the processor instruction set architecture on execution resources in the DRM system. The `CpuArchitecture` enumeration is used as data type in job submission, advance reservation and system monitoring. It defines a set of standardized identifiers for processor architecture families. The list is a shortened version of the according CIM Schema [6], It includes only processor families that are supported by the majority of DRM systems available at the time of writing:

```
enum CpuArchitecture {
  ALPHA, ARM, CELL, PARISC, X86, X64, IA64, MIPS, PPC, PPC64,
  SPARC, SPARC64, OTHER_CPU};
```

ALPHA: The DEC Alpha / Alpha AXP processor architecture.

ARM: The ARM processor architecture.

CELL: The Cell processor architecture.

PARISC: The PA-RISC processor architecture.

X86: The IA-32 line of the X86 processor architecture family, with 32bit support only.

X64: The X86-64 line of the X86 processor architecture family, with 64bit support.

IA64: The Itanium processor architecture.

MIPS: The MIPS processor architecture.

PPC: The PowerPC processor architecture, all models with 32bit support only.

PPC64: The PowerPC processor architecture, all models with 64bit support.

SPARC: The SPARC processor architecture, all models with 32bit support only.

SPARC64: The SPARC processor architecture, all models with 64bit support.

OTHER_CPU: A processor architecture not specified in this list.

The DRMAA `CpuArchitecture` enumeration can be mapped to other high-level APIs. Table 2 gives a non-normative set of examples.

The reporting and job configuration for processor architectures SHOULD operate on a “as-is” base, if supported by the DRM system. This means that the reported architecture should reflect the current operation mode of the processor with the running operating system. For example, X64 processors executing a 32-bit operating system typically report themself as X86 processor.

4.3 ResourceLimitType enumeration

Modern DRM systems expose resource constraint capabilities from the operating system for jobs on the execution host. The `ResourceLimitType` enumeration represents the typical *setrlimit* parameters [5] supported for jobs in different DRM systems. Resource limitations MUST work on the level of jobs. If a job gets more than one slot, the interpretation of limits is implementation-specific.

DRMAA `CpuArchitecture` value	JSDL `jsdl:ProcessorArchitectureEnumeration` value
ALPHA	other
ARM	arm
CELL	other
PARISC	parisc
X86	x86_32
X64	x86_64
IA64	ia64
MIPS	mips
PPC	powerpc
PPC64	powerpc
SPARC	sparc
SPARC64	sparc
OTHER	other

Table 2: Mapping example for DRMAA `CpuArchitecture` enumeration

(See footnote)[8]

```
  enum ResourceLimitType {
    CORE_FILE_SIZE, CPU_TIME, DATA_SEG_SIZE, FILE_SIZE, OPEN_FILES,
    STACK_SIZE, VIRTUAL_MEMORY, WALLCLOCK_TIME };
```

CORE_FILE_SIZE: The maximum size of the core dump file created on fatal errors of the job, in kilobyte. Setting this value to zero SHOULD disable the creation of core dump files on the execution host.

CPU_TIME: The maximum accumulated time in seconds the job is allowed to perform computations. This value includes only time the job is spending in `JobState::RUNNING` (see Section 8.1).

DATA_SEG_SIZE: The maximum amount of memory the job can allocate on the heap e.g. for object creation, in kilobyte.

FILE_SIZE: The maximum file size the job can generate, in kilobyte.

OPEN_FILES: The maximum number of file descriptors the job is allowed to have open at the same time.

STACK_SIZE: The maximum amount of memory the job can allocate on the stack, e.g. for local variables, in kilobyte.

VIRTUAL_MEMORY: The maximum amount of memory the job is allowed to allocate, in kilobyte.

WALLCLOCK_TIME: The maximum wall clock time in seconds the job is allowed to exist. The time amount MUST include the time spent in RUNNING state, and MAY also include the time spent in SUSPENDED state (see Section 8.1). The limit value MAY also be used for job scheduling decisions in the DRM system.

Clarify CPU time and wallclock time for jobs with multiple processes

[8] The June 2011 face-to-face meeting had hard discussion on the relation between operating system processes, jobs, and slots. It was decided that slot is a truly opaque concept, which means that you cannot do resource contraints on something that is implementation-specific. Therefore, the spec semantics must focus on jobs only, and leave the interpretation to the DRM system / DRMAA implementation.This leads to some intentional fuzzying of descriptions for ResourceLimitType members.

(See footnote)[9]

4.4 JobTemplatePlaceholder enumeration

The `JobTemplatePlaceholder` enumeration defines constant macros to be used in string attributes of a `JobTemplate` instance.

```
  enum JobTemplatePlaceholder {
    HOME_DIRECTORY,WORKING_DIRECTORY,PARAMETRIC_INDEX };
```

A `HOME_DIRECTORY` placeholder SHOULD be only allowed at the beginning of a `JobTemplate` attribute value. It denotes the remaining portion as a directory / file path resolved relative to the job users home directory at the execution host.

A `WORKING_DIRECTORY` placeholder SHOULD be only allowed at the beginning of a `JobTemplate` attribute value. It denotes the remaining portion as a directory / file path resolved relative to the jobs working directory at the execution host.

The `PARAMETRIC_INDEX` placeholder SHOULD be usable at any position within an attribute value that supports place holders. It SHALL be substituted by the parametric job index in a `JobSession::runBulkJobs` call (see Section 8.2.7). If the job template is used for a `JobSession:runJob` call, `PARAMETRIC_INDEX` SHOULD be substituted with a constant implementation-specific value.

(See footnote)[10]

4.5 DrmaaCapability

The `DrmaaCapability` enumeration expresses DRMAA features and data attributes that may or may not be supported by a particular implementation. Applications are expected to check the availability of optional capabilities through the `SessionManager::supports` method (see Section 7.1).

```
  enum DrmaaCapability {
    ADVANCE_RESERVATION, RESERVE_SLOTS, CALLBACK,
    BULK_JOBS_MAXPARALLEL,
     JT_EMAIL, JT_STAGING, JT_DEADLINE, JT_MAXSLOTS,
    JT_ACCOUNTINGID, RT_STARTNOW,
    RT_DURATION, RT_MACHINEOS, RT_MACHINEARCH
  };
```

ADVANCE_RESERVATION: Indicates that the advance reservation interfaces (`ReservationSession`, `Reservation`) are functional in this implementation.

RESERVE_SLOTS: Indicates that the advance reservation support is targeting slots. If this capability is not given, the advance reservation is targeting whole machines as granularity level.

[9] "Pipe size" was not added, since there is no use case in DRM systems with a job concept. "Max user processes" was omitted because it operates on the notion of users, which is not an explicit concept in DRMAA.

The understanding of wall clock time was decided in the Apr 6th and 13th 2011 conf call. Condor and Grid Engine also add the SUSPEND time, but LSF does not.

[10] Placeholders for other job template attributes were rejected, in order to avoid circular dependencies (Conf. call Oct 20th 2010). Any extended semantic of placeholders in comparison to DRMAA1 was rejected, since the support in the DRM system didn't change. (conf call Apr. 20th 2011)

CALLBACK: Indicates that the implementation supports event notification through a `DrmaaCallback` interface in the application.

BULK_JOBS_MAXPARALLEL: Indicates that the `maxParallel` parameter in the `JobSession::runBulkJobs` method is considered and supported by the implementation.

JT_EMAIL: Indicates that the optional `email`, `emailOnStarted`, and `emailOnTerminated` attributes in a job template are supported by the implementation.

JT_STAGING: Indicates that the optional `JobTemplate::stageInFiles` and `JobTemplate::stageOutFiles` attributes are supported by the implementation.

JT_DEADLINE: Indicates that the optional `JobTemplate::deadlineTime` attribute is supported by the implementation.

JT_MAXSLOTS: Indicates that the optional `JobTemplate::maxSlots` attribute is supported by the implementation.

JT_ACCOUNTINGID: Indicates that the optional `JobTemplate::accountingId` attribute is supported by the implementation.

RT_STARTNOW: Indicates that the `ReservationTemplate::startTime` attribute accepts the `NOW` value.

RT_DURATION: Indicates that the optional `ReservationTemplate::duration` attribute is supported by the implementation.

RT_MACHINEOS: Indicates that the optional `ReservationTemplate::machineOS` attribute is supported by the implementation.

RT_MACHINEARCH: Indicates that the optional `ReservationTemplate::machineArch` attribute is supported by the implementation.

5 Extensible Data Structures

DRMAA defines a set of data structures commonly used by different interfaces to express information for and from the DRM system. A DRMAA implementation is allowed to extend these structures with *implementation-specific attributes* in all cases. Behavioral aspects of such extended attributes are out of scope for DRMAA. The interpretation is implementation-specific, implementations MAY even ignore such attribute values.

Implementations SHALL only extend data structures in the way specified by the language binding. The introspection about supported implementation-specific attributes is supported by the `DrmaaReflective` interface (see Section 5.9). Implementations SHOULD also support native introspection functionalities if defined by the language binding.

A language binding MUST define a consistent mechanism to realize implementation-specific structure extension, without breaking the portability of DRMAA-based applications that relies on the original version of the structure. Object oriented languages MAY use inheritance mechanisms for this purpose. Instances of these structures SHALL be treated in a "call-by-value" fashion, meaning that the collection of struct member values is handed over as one to the called interface method.

Language bindings MAY define how native introspection capabilities of the language or it's runtime environment can also be used to work with implementation-specific attributes. These mechanisms MUST work in parallel to the `DrmaaReflective` interface.

(See footnote)[11]

5.1 QueueInfo structure

A queue is an opaque concept from the perspective of the DRMAA application (see Section 1.3). The `QueueInfo` struct contains read-only information, which can be extended by the implementation as described in Section 5.

```
struct QueueInfo {
  string name;
};
```

5.1.1 name

This attribute contains the name of the queue as reported by the DRM system. The format of the queue name is implementation-specific. The naming scheme SHOULD be consistent for all strings returned.

5.2 Version structure

The `Version` structure denotes versioning information for an operating system, DRM system, or DRMAA implementation.

```
struct Version {
  string major;
  string minor;
};
```

Both the `major` and the `minor` part are expressed as strings, in order to allow extensions with character combinations such as "rev". Original version strings containing a dot, e.g. Linux "2.6", SHOULD be interpreted as having the major part before the dot, and the minor part after the dot. The dot character SHOULD NOT be added to the `Version` attributes.

Implementations SHOULD NOT extend this structure with implementation-specific attributes.

[11] Comparison to DRMAA 1.0: The binding of job template attribute names and exception names to strings was removed. Language bindings have to define their own mapping, if needed.

One example for native language introspection support could be attributes.

There was a discussion to remove the attribute ignorance possibility for implementations, in order to have a defined error when unknown attributes are used. This was rejected on the Apr. 13th conf call, since applications do not need the error as indication for missing attribute support. Instead, they should use the given introspection capabilities.

(See footnote)[12]

5.3 MachineInfo structure

The MachineInfo structure describes the properties of a particular execution host in the DRM system. It contains read-only information. An implementation or its DRM system MAY restrict jobs in their resource utilization even below the limits described in the MachineInfo structure. The limits given here MAY be imposed by the hardware configuration, or MAY be be imposed by DRM system policies.

```
struct MachineInfo {
  string name;
  boolean available;
  long sockets;
  long coresPerSocket;
  long threadsPerCore;
  double load;
  long physMemory;
  long virtMemory;
  OperatingSystem machineOS;
  Version machineOSVersion;
  CpuArchitecture machineArch;
};
```

5.3.1 name

This attribute describes the name of the machine as reported by the DRM system. The format of the machine name is implementation-specific, but MAY be a DNS host name. The naming scheme SHOULD be consistent among all machine struct instances.

5.3.2 available

This attribute expresses the usability of the machine for job execution at the time of querying. The value of this attribute SHALL NOT influence the validity of job template instances containing a candidateHosts setting, since the availability of machines is expected to change at any point in time. DRM systems may allow to submit jobs for unavailable machines, where these jobs are queued until the machine becomes available again.

5.3.3 sockets

This attribute describes the number of processor sockets (CPUs) usable for jobs on the machine from operating system perspective. The attribute value MUST be greater than 0. In the case where the correct value is unknown to the implementation, the value MUST be set to 1.

5.3.4 coresPerSocket

This attribute describes the number of cores per socket usable for jobs on the machine from operating system perspective. The attribute value MUST be greater than 0. In case where the correct value is unknown to

[12] We could see no use case in doing implementation-specific extensions here, so this structure is not considered in DrmaaReflective.

the implementation, the value MUST be set to 1.

5.3.5 threadsPerCore

This attribute describes the number of threads that can be executed in parallel by a job's process on one core in the machine. The attribute value MUST be greater than 0. In case where the correct value is unknown to the implementation, the value MUST be set to 1.

5.3.6 load

This attributes describes the 1-minute average load on the given machine, similar to the Unix *uptime* command. The value has only informative character, and should not be utilized by end user applications for job scheduling purposes. An implementation MAY provide delayed or averaged data here, if necessary due to implementation issues. The implementation strategy on non-Unix systems is undefined.

(See footnote)[13]

5.3.7 physMemory

This attribute describes the amount of physical memory in kilobyte available on the machine.

5.3.8 virtMemory

This attribute describes the amount of virtual memory in kilobyte available for a job executing on this machine. The virtual memory amount is defined as the sum of physical memory installed plus the configured swap space for the operating system. The value is expected to be used as indicator whether or not an application is able to get its memory allocation needs fulfilled on a particular machine. Implementations SHOULD derive this value directly from operating system information, without further consideration of additional memory allocation restrictions such as address space range or already running processes.

5.3.9 machineOS

This attribute describes the operating system installed on the described machine, with semantics as specified in Section 4.1.

5.3.10 machineOSVersion

This attribute describes the operating system version of the machine, with semantics as specified in Section 4.1.

5.3.11 machineArch

This attribute describes the instruction set architecture of the machine, with semantics as specified in Section 4.2.

[13] In July 2011, there was a short debate on the list if this value should be normalized by the library to ¡0,1¿. It was rejected, since DRMAA should just forward given information from the DRM / OS, for which the maximum value is typically not known.

5.4 SlotInfo structure

The SlotInfo structure describes the amount of reserved slots on a machine, resulting from an advance reservation operation (see also Section 1.3).

Implementations SHOULD NOT extend this structure with implementation-specific attributes.

(See footnote)[14]

```
struct SlotInfo {
  string machineName;
  string slots;
};
```

5.4.1 machineName

The name of the machine. Strings returned here SHOULD be equal to the MachineInfo::name attribute in the matching MachineInfo instance.

5.4.2 slots

The number of slots reserved on the given machine. Depending on the intepretation of slots in the implementation, this value MAY be always one.

5.5 JobInfo structure

The JobInfo structure describes job information that is available for the DRMAA-based application.

```
struct JobInfo {
  string jobId;
  long exitStatus;
  string terminatingSignal;
  string annotation;
  JobState jobState;
  any jobSubState;
  OrderedStringList allocatedMachines;
  string submissionMachine;
  string jobOwner;
  long slots;
  string queueName;
  TimeAmount wallclockTime;
  long cpuTime;
  AbsoluteTime submissionTime;
  AbsoluteTime dispatchTime;
  AbsoluteTime finishTime;
};
```

[14] We could see no use case in realizing implementation-specific extensions here, so this structure is not considered in DrmaaReflective.

The structure is used in two occasions - first for the expression of information about a single job, and second as filter expression when retrieving a list of jobs from the DRMAA implementation.

In both usage scenarios, the structure information has to be understood as snapshot of the live DRM system. Multiple values being set in one structure instance should be interpreted as "occurring at the same time". In real implementations, some granularity limits must be assumed - for example, the `wallclockTime` and the `cpuTime` attributes might hold values that were measured with a very small delay one after each other.

DRMAA makes no assumption on the `JobInfo` availability for jobs in a a "Terminated" state (see Section 8.1). Implementations SHOULD allow to fetch information about such jobs, complete or incomplete, for a reasonable amount of time. For such terminated jobs, implementations MAY also decide to return only partially filled `JobInfo` instances due to performance restrictions in the communication with the DRM system.

For additional DRMS-specific information, the `JobInfo` structure MAY be extended by the DRMAA implementation (see Section 5).

(See footnote)[15]

5.5.1 jobId

For monitoring: Returns the stringified job identifier assigned to the job by the DRM system.

For filtering: Returns the job with the chosen job identifier.

5.5.2 exitStatus

For monitoring: The process exit status of the job, as reported by the operating system. If the job is not in one of the terminated states, the value should be `UNSET`.

For filtering: Return the jobs with the given `exitStatus` value. Jobs without exit status information should be filtered out by asking for the appropriate states.

5.5.3 terminatingSignal

For monitoring: This attribute specifies the UNIX signal that reasoned the end of the job. Implementations should document the extent to which they can gather such information in the particular DRM system (e.g. with Windows hosts).

For filtering: Returns the jobs with the given `terminatingSignal` value.

[15] In comparison to DRMAA 1.0, the JobInfo value type was heavily extended for providing more information (solves issue #2827). JobInfo::hasCoreDump is no longer supported, since the information is useless without according core file staging support, which is not implementable in a portable way. (conf. call Jun 9th 2010) resourceUsage is no longer supported, since this should be modelled with implementation-specific attributes (conf call Apr 13th 2011).

Some DRM systems (SGE / Condor at least) support the automated modification of job template attributes after submission, and therefore allow to fetch the true job template attributes at run-time from the job. The monitoring for such data was intentionally not included in DRMAA (mailing list July 2010).

A comment attribute was rejected (conf call May 11th).

Several conf. calls in 2011 ended up in the conclusion that data reaping cannot be clarified by DRMAA. There are too many completely different use cases in local and distributed systems.

5.5.4 annotation

For monitoring: Gives a human-readable annotation describing why the job is in its current state or sub-state. Implementations MAY decide to offer such description only in specific cases.

For filtering: This attribute is ignored for filtering.

5.5.5 jobState

For monitoring: This attribute specifies the jobs current state according to the DRMAA job state model (see Section 8.1).

For filtering: Returns all jobs in the specified state. If the given state is simulated by the implementation (see Section 8.1), the implementation SHOULD raise an `InvalidArgumentException` explaining that this filter can never match.

5.5.6 jobSubState

For monitoring: This attribute specifies the jobs current DRMAA implementation specific sub-state (see Section 8.1).

For filtering: Returns all jobs in the specified sub-state. If the given sub-state is not supported by the implementation (see Section 8.1), the implementation SHOULD raise an `InvalidArgumentException` explaining that this filter can never match.

As the JobSubState is an opaque object then passing -sub-state is not suported by the impl..-may simply lead to SEG FAULT ;-) so filtering using sub-state should be permitted if one known which implementation is used.

5.5.7 allocatedMachines

This attribute expresses the set of machines that are utilized for job execution. Implementations MAY decide to give the ordering of machine names a particular meaning, for example putting the master node in a parallel job at first position. This decision should be documented for the user. For performance reasons, only the machine names are returned, and SHOULD be equal to the according `MachineInfo::name` attribute in monitoring data.

For monitoring: This attribute lists the set of names of the machines to which this job has been assigned.

For filtering: Returns the list of jobs which have a set of assigned machines that is a superset of the given set of machines.

5.5.8 submissionMachine

This attribute provides the machine name of the submission host for this job. For performance reasons, only the machine name is returned, and SHOULD be equal to the according `MachineInfo::name` attribute in monitoring data.

For monitoring: This attribute specifies the machine from which this job was submitted.

For filtering: Returns the set of jobs that were submitted from the specified machine.

5.5.9 jobOwner

For monitoring: This attribute specifies the job owner as reported by the DRM system.

For filtering: Returns all jobs owned by the specified user.

5.5.10 slots

For monitoring: This attribute reports the number slots that were allocated for the job. The value SHOULD be in between `JobTemplate::minSlots` and `JobTemplate::maxSlots`.

For filtering: Return all jobs with the specified number of reserved slots.

5.5.11 queueName

For monitoring: This attribute specifies the name of the queue in which the job was queued or started (see Section 1.3).

For filtering: Returns all jobs that were queued or started in the queue with the specified name.

5.5.12 wallclockTime

For monitoring: The accumulated wall clock time, with the semantics as defined in Section 4.3.

For filtering: Returns all jobs that have consumed at least the specified amount of wall clock time.

5.5.13 cpuTime

For monitoring: The accumulated CPU time, with the semantics as defined in Section 4.3.

For filtering: Returns all jobs that have consumed at least the specified amount of CPU time.

5.5.14 submissionTime

For monitoring: This attribute specifies the time at which the job was submitted. Implementations SHOULD use the submission time recorded by the DRM system, if available.

For filtering: Returns all jobs that were submitted at or after the specified submission time.

5.5.15 dispatchTime

For monitoring: The time the job first entered a “Started” state (see Section 8.1). On job restart or re-scheduling, this value does not change.

For filtering: Returns all jobs that entered a “Started” state at, or after the specified dispatch time.

5.5.16 finishTime

For monitoring: The time the job first entered a “Terminated” state (see Section 8.1).

For filtering: Returns all jobs that entered a “Terminated” state at or after the specified finish time.

5.6 ReservationInfo structure

The ReservationInfo structure describes reservation information that is available for the DRMAA-based application.

```
  struct ReservationInfo {
    string reservationId;
    string reservationName;
    AbsoluteTime reservedStartTime;
    AbsoluteTime reservedEndTime;
    StringList usersACL;
    long reservedSlots;
    SlotInfoList reservedMachines;
  };
```

The structure is used for the expression of information about a single advance reservation. Information provided in this structure SHOULD NOT change over the reservation lifetime. However, implementations MAY reflect the altering of advance reservations outside of DRMAA sessions.

For additional DRMS-specific information, the ReservationInfo structure MAY be extended by the DRMAA implementation (see Section 5).

5.6.1 reservationId

Returns the string version of the identifier assigned to the advance reservation by the DRM system.

5.6.2 reservationName

This attribute describes the reservation name that was stored by the implementation or DRM system, derived from the original reservationName attribute given in the ReservationTemplate.

5.6.3 reservedStartTime

This attribute describes the start time for the reservation. If the value is UNSET, it expresses an unrestricted start time (i.e., *minus infinity*) for this reservation.

5.6.4 reservedEndTime

This attribute describes the end time for the reservation. If the value is UNSET, the behavior is implementation-specific.

(See footnote)[16]

5.6.5 usersACL

The list of the users that are permitted to submit jobs to the reservation.

[16] Mai 18th 2011 conf call rejected to treat UNSET as unrestricted end time (i.e. “plus infinity”) here.

5.6.6 reservedSlots

This attribute describes the number of slots reserved by the DRM system. The value SHOULD range in between `ReservationTemplate::minSlots` and `ReservationTemplate::maxSlots`.

5.6.7 reservedMachines

This attribute describes the set of machines that were reserved under the conditions described in the according reservation template. Each `SlotInfo` instance in the result describes the reservation of a particular machine, and of a set of slots related to this machine. The sum of all slot counts in the sequence SHOULD be equal to `ReservationInfo::reservedSlots`.

5.7 JobTemplate structure

In order to define the attributes associated with a job, a DRMAA application uses the `JobTemplate` structure. It specifies any required job parameters and is passed to the DRMAA `JobSession` instance when job execution is requested.

```
  struct JobTemplate {
    string remoteCommand;
    OrderedStringList args;
    boolean submitAsHold;
    boolean rerunnable;
    Dictionary jobEnvironment;
    string workingDirectory;
    string jobCategory;
    StringList email;
    boolean emailOnStarted;
    boolean emailOnTerminated;
    string jobName;
    string inputPath;
    string outputPath;
    string errorPath;
    boolean joinFiles;
    string reservationId;
    string queueName;
    long minSlots;
    long maxSlots;
    long priority;
    OrderedStringList candidateMachines;
    long minPhysMemory;
    OperatingSystem machineOS;
    CpuArchitecture machineArch;
    AbsoluteTime startTime;
    AbsoluteTime deadlineTime;
    Dictionary stageInFiles;
    Dictionary stageOutFiles;
    Dictionary resourceLimits;
    string accountingId;
  };
```

The DRMAA job template concept makes a distinction between *mandatory* and *optional* attributes. Mandatory attributes MUST be supported by the implementation in the sense that they are evaluated on job submission. Optional attributes MAY be evaluated on job submission, but MUST be provided as part of the `JobTemplate` structure in the implementation. If an unsupported optional attribute has a value different to `UNSET`, the job submission MUST fail with a `UnsupportedAttributeException`. DRMAA applications are expected to check for the availability of optional attributes before using them (see Section 4.5).

Implementations MUST set all attribute values to `UNSET` on struct allocation. This ensures that both the DRMAA application and the library implementation can determine untouched attribute members. If not described differently in the following sections, all attributes SHOULD be allowed to have the `UNSET` value on job submission.

An implementation MAY support `JobTemplatePlaceholder` macros in more occasions than defined in this specification.

A language binding specification SHOULD define how a `JobTemplate` instance is convertible to a string for printing, through whatever mechanism is most natural for the implementation language. The resulting string MUST contain the values of all set properties.

The initialization to `UNSET` SHOULD be realized without additional methods in the DRMAA interface, if possible. The according approach MUST be specified by the language binding.

(See footnote)[17]

5.7.1 remoteCommand

This attribute describes the command to be executed on the remote host. In case this parameter contains path information, it MUST be seen as relative to the execution host file system and is therefore evaluated there. The implementation SHOULD NOT use the value of this attribute to trigger file staging activities. Instead, the file staging should be performed by the application explicitly.

The behavior with an `UNSET` value is implementation-specific.

The support for this attribute is mandatory.

5.7.2 args

This attribute contains the list of command-line arguments for the job(s) to be executed.

The support for this attribute is mandatory.

[17] Comparison to DRMAA 1.0: JobTemplate is now a value type, meaning that it maps to a struct in C. This removes the need for DRMAA-defined methods for construction and destruction of job templates. An eventual RPC scenario for DRMAA gets easier with this approach, since it is closer to the JSDL concept of a job description document.

Supported string placeholders for job template attributes are now listed in the JobTemplatePlaceholder enumeration, and must be filled with values by the language binding. Invalid job template settings are now only detected on job submission, not when the attribute is set.

DRMAA1 supported the utilization of new DRM features through an old DRMAA implementation, based on the `nativeSpecification` field. A conf call (Jul 14th 2010) voted for dropping this intentionally. Implementations should use according implementation-specific attributes for this.

GridEngine does not support to request a number of slots per machine - of course in a default installation, since you can do everything in GridEngine ... This is the reason for not having such an attribute.

5.7.3 submitAsHold

This attribute defines if the job(s) should be submitted as QUEUED or QUEUED_HELD (see Section 8.1). Since the boolean UNSET value defaults to False, jobs are submitted as non-held if this attribute is not set.

The support for this attribute is mandatory.

5.7.4 rerunnable

This flag indicates if the submitted job(s) can safely be restarted by the DRM system, for example on a node failure or some other re-scheduling event. Since the boolean UNSET value defaults to False, jobs are submitted as not rerunnable if this attribute is not set. This attribute SHOULD NOT be used by the implementation to let the application denote the checkpointability of a job.

The support for this attribute is mandatory.

(See footnote)[18]

5.7.5 jobEnvironment

This attribute holds the environment variable key-value pairs for the execution machine(s). The values SHOULD override the execution host environment values if there is a collision.

The support for this attribute is mandatory.

5.7.6 workingDirectory

This attribute specifies the directory where the job or the bulk jobs are executed. If the attribute value is UNSET, the behavior is implementation dependent. Otherwise, the attribute value MUST be evaluated relative to the file system on the execution host. The attribute value MUST be allowed to contain either the JobTemplatePlaceholder::HOME_DIRECTORY or the JobTemplatePlaceholder::PARAMETRIC_INDEX placeholder (see Section 4.4).

The workingDirectory attribute should be specified by the application in a syntax that is common at the host where the job is executed. Implementations MAY perform according validity checks on job submission. If the attribute is set and no placeholder is used, an absolute directory specification is expected. If the attribute is set and the job was submitted successfully and the directory does not exist on the execution host, the job MUST enter the state JobState::FAILED.

The support for this attribute is mandatory.

5.7.7 jobCategory

This attribute defines the job category to be used (see Section 1.4). A valid input SHOULD be one of the strings in MonitoringSession::drmsJobCategoryNames (see Section 10.1), otherwise an InvalidArgumentException SHOULD be raised.

The support for this attribute is mandatory.

[18] The differentiation between rerunnable and checkpointable was decided on a conf call (Aug 25th 2010). Checkpointability indication was intentionally left out, since there is no common understanding in the DRM systems (conf call Apr. 27th, 2011).

5.7.8 email

This attribute holds a list of email addresses that should be used to report DRM information. Content and formatting of the emails are defined by the implementation or the DRM system. If the attribute value is `UNSET`, no emails SHOULD be sent to the user running the job(s), even if the DRM system default behavior is to send emails on some event.

The support for this attribute is optional, expressed by the `DrmaaCapability::JT_EMAIL` flag. If an implementation cannot configure the email notification functionality of the DRM system, or if the DRM system has no such functionality, the attribute SHOULD NOT be supported in the implementation.

(See footnote)[19]

5.7.9 emailOnStarted / emailOnTerminated

The `emailOnStarted` flag indicates if the given email address(es) SHOULD get a notification when the job (or any of the bulk jobs) entered one of the “Started” states. `emailOnTerminated` fulfills the same purpose for the ”Terminated” states. Since the boolean `UNSET` value defaults to `False`, the notification about state changes SHOULD NOT be sent if the attribute is not set.

The support for these attributes is optional, expressed by the `DrmaaCapability::JT_EMAIL` flag.

5.7.10 jobName

The job name attributes allows the specification of an additional non-unique string identifier for the job(s). The implementation MAY truncate any client-provided job name to an implementation-defined length.

The support for this attribute is mandatory.

5.7.11 inputPath / outputPath / errorPath

This attribute specifies standard input / output / error stream of the job as a path to a file. If the attribute value is `UNSET`, the behavior is implementation dependent. Otherwise, the attribute value MUST be evaluated relative to the file system of the execution host in a syntax that is common at the host. Implementations MAY perform according validity checks on job submission. The attribute value MUST be allowed to contain any of the `JobTemplatePlaceholder` placeholders (see Section 4.4). If the attribute is set and no placeholder is used, an absolute file path specification is expected.

If the `outputPath` or `errorPath` file does not exist at the time the job is about to be executed, the file SHALL first be created. An existing `outputPath` or `errorPath` file SHALL be opened in append mode.

If the attribute is set and the job was submitted successfully and the file cannot be created / read / written on the execution host, the job MUST enter the state `JobState::FAILED`.

The support for this attribute is mandatory.

5.7.12 joinFiles

Specifies whether the error stream should be intermixed with the output stream. Since the boolean `UNSET` value defaults to `False`, intermixing SHALL NOT happen if the attribute is not set.

[19] The blockEmail attribute in the JobTemplate was replaced by the UNSET semantic for the email addresses. (conf. call July 28th 2010). This became an optional attribute, since we mandate the 'switch off' semantic in case of `UNSET`.

If this attribute is set to `True`, the implementation SHALL ignore the value of the `errorPath` attribute and intermix the standard error stream with the standard output stream as specified by the `outputPath`.

The support for this attribute is mandatory.

5.7.13 stageInFiles / stageOutFiles

Specifies what files should be transferred (staged) as part of the job execution. The data staging operation MUST be a copy operation between the submission host and the execution host(s) (see also Section 1 for host types). File transfers between execution hosts are not covered by DRMAA.

The attribute value is formulated as dictionary. For each key-value pair in the dictionary, the key defines the source path of one file or directory, and the value defines the destination path of one file or directory for the copy operation. For `stageInFiles`, the submission host acts as source, and the execution host(s) act as destination. For `stageOutFiles`, the execution host(s) acts as source, and the submission host act as destination.

All values MUST be evaluated relative to the file system on the host in a syntax that is common at that host. Implementations MAY perform according validity checks on job submission. Paths on the execution host(s) MUST be allowed to contain any of the `JobTemplatePlaceholder` placeholders. Paths on the submission host MUST be allowed to contain the `JobTemplatePlaceholder::PARAMETRIC_INDEX` placeholder (see Section 4.4). If no placeholder is used in the values, an absolute path specification on the particular host SHOULD be assumed by the implementation.

Relative path specifications for the submission host should be interpreted starting from the current working directory of the DRMAA application at the time of job submission. The behavior for relative path specifications on the execution is implementation-specific. Implementations MAY use *JobTemplate::workingDirectory* as starting point on the execution host in this case, if given by the application.

Jobs SHOULD NOT enter `JobState::DONE` unless all staging operations are finished. The behavior in case of missing files is implementation-specific. The support for wildcard operators in path specifications is implementation-specific. Any kind of recursive or non-recursive copying behavior is implementation-specific.

If the job category (see Section 1.4) implies a parallel job (e.g., MPI), the copy operation SHOULD target the parallel job master host as destination. A job category MAY also trigger file distribution to other hosts participating in the job execution.

The support for this attribute is optional, expressed by the `DrmaaCapability::JT_STAGING` flag.

(See footnote)[20]

5.7.14 reservationId

Specifies the identifier of the advance reservation associated with the job(s). The application is expected to create an advance reservation through the `ReservationSession` interface, the resulting `reservationId` (see Section 9.2) then acts as valid input for this job template attribute. Implementations MAY support a reservation identifier from non-DRMAA information sources as valid input.

The support for this attribute is mandatory.

[20] Comparison to DRMAA 1.0: New job template attributes for file transfers were introduced. They allow to express a set of file staging activities, similar to the approach in LSF and SAGA. They replace the old transferFiles attribute, the according FileTransferMode data structure and the special host definition syntax in inputPath / outputPath / errorPath (different conf. calls, SAGA F2F meeting, solves issue #5876)

5.7.15 queueName

This attribute specifies the name of the queue the job(s) should be submitted to. In case this attribute value is `UNSET`, and `MonitoringSession::getAllQueues` returns a list with a minimum length of 1, the implementation SHOULD use the DRM systems default queue.

The `MonitoringSession::getAllQueues` method (see 10.1) supports the determination of valid queue names. Implementations SHOULD allow these queue names to be used in the `queueName` attribute. Implementations MAY also support queue names from other non-DRMAA information sources as valid input. If no default queue is defined or if the given queue name is not valid, the job submission MUST lead to an `InvalidArgumentException`.

If `MonitoringSession::getAllQueues` returns an empty list, this attribute MUST be only accepted with the value `UNSET`.

Since the meaning of "queues" is implementation-specific, there is no implication on the effects in the DRM system when using this attribute. As one example, requesting a number of slots for a job in one queue has no implication on the number of utilized machines at run-time. Implementations therefore SHOULD document the effects of this attribute accordingly.

The support for this attribute is mandatory.

5.7.16 minSlots

This attribute expresses the minimum number of slots requested per job (see also Section 1.3). If the value of `minSlots` is `UNSET`, it SHOULD default to 1.

Implementations MAY interpret the slot count as number of concurrent processes being allowed to run. If this interpretation is taken, and `minSlots` is greater than 1, than the `jobCategory` SHOULD also be demanded on job submission, in order to express the nature of the intended parallel job execution.

The support for this attribute is mandatory.

(See footnote)[21]

5.7.17 maxSlots

This attribute expresses the maximum number of slots requested per job (see also Section 1.3). If the value of `maxSlots` is `UNSET`, it SHOULD default to the value of `minSlots`.

Implementations MAY interpret the slot count as number of concurrent processes being allowed to run. If this interpretation is taken, and `maxSlots` is greater than 1, than the `jobCategory` SHOULD also be demanded on job submission, in order to express the nature of the intended parallel job execution.

The support for this attribute is optional, as indicated by the `DrmaaCapability::JT_MAXSLOTS` flag.

(See footnote)[22] .

[21] The hint regarding number of concurrent processes intentionally does not speak about processes per host - this would create semantics for our opaque slot concept.

[22] Torque does not support maxSlots on job submission, conf call on May 11th decided to keep it as optional feature. Expected use cases are billing limitations and parallel job scalability considerations

5.7.18 jobCategory

This attribute defines the job category to be used (see Section 1.4). A valid input SHOULD be one of the strings in `MonitoringSession::drmsJobCategoryNames` (see Section 10.1), otherwise an `InvalidArgumentExc` SHOULD be raised.

The support for this attribute is mandatory.

New, needs group approval. Long explanation is now in Section 1.4

5.7.19 priority

This attribute specifies the scheduling priority for the job. The interpretation of the given value incl. an `UNSET` value is implementation-specific.

The support for this attribute is mandatory.

5.7.20 candidateMachines

Requests that the job(s) should run on any subset (with minimum size of 1), or all of the given machines. If the attribute value is `UNSET`, it should default to the result of the `MonitoringSession::getAllMachines` method. If this resource demand cannot be fulfilled, an `InvalidArgumentException` SHOULD be raised on job submission time. If the problem can only be detected after job submission, the job should enter `JobState::FAILED`.

The support for this attribute is mandatory.

5.7.21 minPhysMemory

This attribute denotes the minimum amount of physical memory in kilobyte that should be available for the job. If the job gets more than one slot, the interpretation of this value is implementation-specific. If this resource demand cannot be fulfilled, an `InvalidArgumentException` SHOULD be raised at job submission time. If the problem can only be detected after job submission, the job SHOULD enter `JobState::FAILED` accordingly.

The support for this attribute is mandatory.

5.7.22 machineOS

This attribute denotes the expected operating system type on the / all execution host(s). If this resource demand cannot be fulfilled, an `InvalidArgumentException` SHOULD be raised on job submission time. If the problem can only be detected after job submission, the job SHOULD enter `JobState::FAILED` accordingly.

The support for this attribute is mandatory.

(See footnote)[23]

[23] Requesting a specific operating system version beyond the type is not supported by the majority of DRM systems (conf call Jul 28th 2010)

5.7.23 machineArch

This attribute denotes the expected machine architecture on the / all execution host(s). If this resource demand cannot be fulfilled, an `InvalidArgumentException` SHOULD be raised on job submission time. If the problem can only be detected after job submission, the job should enter `JobState::FAILED`.

The support for this attribute is mandatory.

5.7.24 startTime

This attribute specifies the earliest time when the job may be eligible to be run.

The support for this attribute is mandatory.

5.7.25 deadlineTime

Specifies a deadline after which the implementation or the DRM system SHOULD change the job state to any of the "Terminated" states (see Section 8.1).

The support for this attribute is optional, as expressed by the `DrmaaCapability::JT_DEADLINE`.

5.7.26 resourceLimits

This attribute specifies the limits on resource utilization of the job(s) on the execution host(s). The valid dictionary keys and their value semantics are defined in Section 4.3.

The following resource restrictions should operate as soft limit, meaning that exceeding the limit SHOULD NOT influence the job state from a DRMAA perspective:

- `CORE_FILE_SIZE`
- `DATA_SEG_SIZE`
- `FILE_SIZE`
- `OPEN_FILES`
- `STACK_SIZE`
- `VIRTUAL_MEMORY`

The following resource restrictions should operate as hard limit, meaning that exceeding the limit MAY terminate the job. The termination could be performed by the DRM system, or by the job itself if it reacts on a signal from the DRM system or the execution host operating system:

- `CPU_TIME`
- `WALLCLOCK_TIME`

The support for this attribute is mandatory. If only a subset of the attributes from `ResourceLimitType` is supported by the implementation, and some of the unsupported attributes are used, the job submission SHOULD raise an `InvalidArgumentException` expressing the fact that resource limits are supported in general.

Conflicts of these attribute values with any other job template attribute or with referenced advance reservations are handled in an implementation-specific manner. Implementations SHOULD try to delegate the decision about parameter combination validity to the DRM system, in order to ensure similar semantics in different DRMAA implementations for this system.

(See footnote)[24]

5.7.27 accountingId

This attribute denotes a string that can be used by the DRM system for job accounting purposes. Implementations SHOULD NOT utilize this information as authentication token, but only as identification information in addition to the implementation-specific authentication (see Section 12).

The support for this attribute is optional, as described by the `DrmaaCapability::JT_ACCOUNTINGID` flag.

5.8 ReservationTemplate structure

In order to define the attributes associated with an advance reservation, the DRMAA application creates an `ReservationTemplate` instance and requests the fulfillment through the `ReservationSession` methods in the DRM system.

```
struct ReservationTemplate {
  string reservationName;
  AbsoluteTime startTime;
  AbsoluteTime endTime;
  TimeAmount duration;
  long minSlots;
  long maxSlots;
  string jobCategory;
  StringList usersACL;
  OrderedStringList candidateMachines;
  long minPhysMemory;
  OperatingSystem machineOS;
  CpuArchitecture machineArch;
};
```

Similar to the `JobTemplate` concept (see Section 5.7), there is a distinction between *mandatory* and *optional* attributes. Mandatory attributes MUST be supported by the implementation in the sense that they are evaluated in a `ReservationSession::requestReservation` call. Optional attributes MAY NOT be evaluated by the particular implementation, but MUST be provided as part of the `ReservationTemplate` structure in the implementation. If an optional attribute is not evaluated by the particular implementation, but has a value different to `UNSET`, the call to `ReservationSession::requestReservation` MUST fail with an `UnsupportedAttributeException`.

[24] In comparison to DRMAA 1.0, resource usage limitations can now be expressed by two dictionaries and an according standardized set of valid dictionary keys (LimitType). The idea is to allow a direct mapping to ulimit(3) semantics, which are supported by the majority of DRM system today. A separate run duration limit is no longer needed, since this is covered by the new CPU_TIME limit parameter. (conf. call Jun 9th 2010).

This distinguishing between different reactions on limit violation was restricted to the job entering, or not entering, the FAILED state. All further effects (e.g., no more *open()* calls possible) are out of scope for DRMAA, since they relate to operating system behavior on execution host (conf call May 4th 2011).

The attribute is mandatory, since the missing general support for resource limits can be simply expressed by raising InvalidArgumentException for all types.

Implementations MUST set all attribute values to `UNSET` on struct allocation. This ensures that both the DRMAA application and the library implementation can determine untouched attribute members.

A language binding specification SHOULD model the `ReservationTemplate` representation the same way as the `JobTemplate` interface (see Section 5.7), and therefore MUST define the realization of implementation-specific attributes, printing, and the initialization of attribute values.

5.8.1 reservationName

A human-readable reservation name. The implementation MAY truncate or alter any application-provided job name in order to adjust it to the DRMS specific constraints. The name of the reservation SHALL be automatically defined by the implementation if this application provides no value on its own.

The support for this attribute is mandatory.

5.8.2 startTime / endTime / duration

The time frame in which resources should be reserved. Table 3 explains the different possible parameter combinations and their semantic.

`startTime`	`endTime`	`duration`	Description
`UNSET`	`UNSET`	`UNSET`	Invalid, SHALL leave to a `InvalidArgumentException` on the reservation attempt.
Set	`UNSET`	`UNSET`	Invalid, SHALL leave to a `InvalidArgumentException` on the reservation attempt.
`UNSET`	Set	`UNSET`	Invalid, SHALL leave to a `InvalidArgumentException` on the reservation attempt.
Set	Set	`UNSET`	Perform reservation attempt to get resources in the specified time frame.
`UNSET`	`UNSET`	Set	Perform reservation attempt the get resources at least for the time amount given in `duration`.
Set	`UNSET`	Set	Implies `endTime = startTime + duration`
`UNSET`	Set	Set	Implies `startTime = endTime - duration`
Set	Set	Set	If `endTime - startTime` is larger than `duration`, perform a reservation attempt where the demanded `duration` is fulfilled at the earliest point in time after `startTime`, and without extending `endTime`. If `endTime - startTime` is smaller than `duration`, the reservation attempt SHALL leave to a `InvalidArgumentException`. If `endTime - startTime` and `duration` are equal, `duration` SHALL be ignored.

Table 3: Parameter combinations for the advance reservation time frame. If `duration` is not supported, it should be treated as `UNSET`.

The support for `startTime` and `endTime` is mandatory. The support for `duration` is optional, as described by the `DrmaaCapability::RT_DURATION` flag. Implementations that do not support the described ”sliding window” approach for the SET / SET / SET case SHOULD express this by NOT supporting the *duration* attribute.

Implementations MAY support `startTime` to have the constant value `NOW` (see Section 3), which expresses that the reservation should start at the time of reservation template approval in the DRM system. The support for this feature is declared by the `DrmaaCapability::RT_STARTNOW` flag.

5.8.3 minSlots

The minimum number of requested slots (see also Section 1.3). If the attribute value is `UNSET`, it should default to 1.

The support for this attribute is mandatory.

5.8.4 maxSlots

The maximum number of requested slots (see also Section 1.3). If the attribute value is `UNSET`, it should default to the value of `minSlots`.

The support for this attribute is mandatory.

5.8.5 usersACL

The list of the users that would be permitted to submit jobs to the created reservation. If the attribute value is `UNSET`, it should default to the user running the application.

The support for this attribute is mandatory.

5.8.6 candidateMachines

Requests that the reservation SHALL be created for exactly the given set of machines. Implementations and their DRM systems MAY decide to reserve only a subset of the given machines. If this attribute is not specified, it should default to the result of `MonitoringSession::getAllMachines` (see Section 10.1).

The support for this attribute is mandatory.

(See footnote)[25]

5.8.7 minPhysMemory

Requests that the reservation SHALL be created with machines that have at least the given amount of physical memory in kilobyte. Implementations MAY interpret this attribute value as filter for candidate machines, or as memory reservation demand on a shared execution resource.

The support for this attribute is mandatory.

(See footnote)[26]

5.8.8 machineOS

Requests that the reservation must be created with machines that have the given type of operating system, regardless of its version, with semantics as specified in Section 4.1.

[25] May 18th 2011 conf call identified the subset reservation feature to be only available in some of the systems, so it is no promise here.

[26] May 18th 2011 conf call identified the different understandings of memory reservation.

The support for this attribute is optional, as described by the `DrmaaCapability::RT_MACHINEOS` flag.

(See footnote)[27]

5.8.9 machineArch

Requests that the reservation must be created with machines that have the given instruction set architecture, with semantics as specified in Section 4.2.

The support for this attribute is optional, as described by the `DrmaaCapability::RT_MACHINEARCH` flag.

(See footnote)[28]

5.9 DrmaaReflective Interface

The `DrmaaReflective` interface allows an application to determine the set of supported implementation-specific attributes in the DRMAA structures (see also Section 5). It also standardizes the read / write access to such attributes when their existence is determined at run-time by the application.

Applications are expected to determine the supported optional attributes with the `SessionManager::supports` method (see Section 7.1).

```
  interface DrmaaReflective {
    readonly attribute StringList jobTemplateImplSpec;
    readonly attribute StringList jobInfoImplSpec;
    readonly attribute StringList reservationTemplateImplSpec;
    readonly attribute StringList reservationInfoImplSpec;
    readonly attribute StringList queueInfoImplSpec;
    readonly attribute StringList machineInfoImplSpec;
    readonly attribute StringList notificationImplSpec;

    string getInstanceValue(in any instance, in string name);
    void setInstanceValue(in any instance, in string name, in string value);
    string describeAttribute(in any instance, in string name);
  };
```

5.9.1 jobTemplateImplSpec

This attribute provides the list of supported implementation-specific `JobTemplate` attributes.

5.9.2 jobInfoImplSpec

This attribute provides the list of supported implementation-specific `JobInfo` attributes.

5.9.3 reservationTemplateImplSpec

This attribute provides the list of supported implementation-specific `ReservationTemplate` attributes.

[27] May 18th 2011 conf call identified support in DRM systems to be mainly given by additional configuration only.

[28] May 18th 2011 conf call identified support in DRM systems to be mainly given by additional configuration only.

5.9.4 reservationInfoImplSpec

This attribute provides the list of supported implementation-specific `ReservationInfo` attributes.

5.9.5 queueInfoImplSpec

This attribute provides the list of supported implementation-specific `QueueInfo` attributes.

5.9.6 machineInfoImplSpec

This attribute provides the list of supported implementation-specific `MachineInfo` attributes.

5.9.7 notificationImplSpec

This attribute provides the list of supported implementation-specific `DrmaaNotification` attributes.

5.9.8 getInstanceValue

This method allows to retrieve the attribute value for `name` from the structure instance given in the `instance` parameter. The return value is the stringified current attribute value.

5.9.9 setInstanceValue

This method allows to set the attribute `name` to `value` in the structure instance given in the `instance` parameter. In case the conversion from string input into the native attribute type leads to an error, `InvalidArgumentException` SHALL be thrown.

5.9.10 describeAttribute

This method returns a human-readable description of an attributes purpose, for the attribute described by `name` in the structure instance referenced by `instance`. The content and language of the return value is implementation-specific, but should consider the use case of portal applications.

6 Common Exceptions

The exception model specifies error information that can be returned by a DRMAA implementation on method calls.

```
  exception DeniedByDrmsException {string message;};
  exception DrmCommunicationException {string message;};
  exception TryLaterException {string message;};
  exception SessionManagementException {string message;};
  exception TimeoutException {string message;};
  exception InternalException {string message;};
  exception InvalidArgumentException {string message;};
  exception InvalidSessionException {string message;};
  exception InvalidStateException {string message;};
  exception OutOfResourceException {string message;};
  exception UnsupportedAttributeException {string message;};
  exception UnsupportedOperationException {string message;};
```

If not defined otherwise, the exceptions have the following meaning:

DeniedByDrmsException: The DRM system rejected the operation due to security issues.

DrmCommunicationException: The DRMAA implementation could not contact the DRM system. The problem source is unknown to the implementation, so it is unknown if the problem is transient or not.

TryLaterException: The DRMAA implementation detected a transient problem with performing the operation, for example due to excessive load. The application is recommended to retry the call.

TimeoutException: The timeout given in one the waiting functions was reached without successfully finishing the waiting attempt.

InternalException: An unexpected or internal error occurred in the DRMAA library, for example a system call failure. It is unknown if the problem is transient or not.

InvalidArgumentException: From the viewpoint of the DRMAA library, a function parameter is invalid or inappropriate for the particular function call. If the parameter is a structure, the exception description SHOULD contain the name(s) of the problematic attribute(s).

InvalidSessionException: The session used for the function is not valid, for example since it was closed before.

InvalidStateException: The function call is not allowed in the current state of the job.

OutOfResourceException: This exception can be thrown by any method at any time when the DRMAA implementation has run out of operating system resources, such as buffer, main memory, or disk space.

UnsupportedAttributeException: The optional attribute is not supported by the DRMAA implementation.

UnsupportedOperationException: The function is not supported by the DRMAA implementation. One example is the registration of an event callback function.

The DRMAA specification assumes that programming languages targeted by language bindings typically support the concept of exceptions. If a destination language does not support them (like ANSI C), the language binding specification SHOULD map error conditions to an appropriate consistent concept.

A language binding MAY chose to model exceptions as numeric error codes. In this case, the language binding specification SHOULD specify numeric values for all DRMAA error constants.

The representation of exceptions in the language binding MUST support a possibility to express an exception cause as textual description. This is intended as specialization of the general error information. Implementations MAY use this text also to express DRMS-specific error conditions that are outside of the DRMAA scope.

Object-oriented language bindings MAY decide to derive all exceptions from one or multiple exception base classes, in order to support generic catch clauses. Whenever it is appropriate, language bindings SHOULD replace a DRMAA exception by some semantically equivalent native exception from the application runtime environment.

Language bindings MAY decide to introduce a hierarchical ordering of the DRMAA exceptions through class derivation. In this case, any new exceptions added for aggregation purposes SHOULD be prevented from being thrown, for example by marking them as abstract.

The `UnsupportedAttributeException` may either be raised by the setter function for the attribute or by the job submission function. A consistent decision for either one or the other approach MUST be made by the language binding specification.

(See footnote)[29]

7 The DRMAA Session Concept

DRMAA relies on an overall session concept, which supports the persistency of job and advance reservation information over multiple application runs. This supports short-lived applications that need to work with DRM system state spanning multiple application runs. Typical examples are job submission portals or command-line tools. The session concept is also intended to allow implementations to perform DRM system attach / detach operations at dedicated points in the application control flow.

7.1 SessionManager Interface

```
interface SessionManager{
  readonly attribute string drmsName;
  readonly attribute Version drmsVersion;
  readonly attribute string drmaaName;
  readonly attribute Version drmaaVersion;
  boolean supports(in DrmaaCapability capability);
  JobSession createJobSession(in string sessionName,
                              in string contactString);
  ReservationSession createReservationSession(in string sessionName,
```

[29] Comparison to DRMAA 1.0: The InconsistentStateException was removed, since it is semantically equal to the InvalidStateException (conf. call Jan 7th 2010) The former HoldInconsistentStateException, ReleaseInconsistentStateException, ResumeInconsistentStateException, and SuspendInconsistentStateException from DRMAA v1.0 are now expressed as single InvalidStateException with different meaning per raising method. (F2F meeting July 2009)

```
                                              in string contactString);
  MonitoringSession createMonitoringSession (in string contactString);
  JobSession openJobSession(in string sessionName);
  ReservationSession openReservationSession(in string sessionName);
  void closeJobSession(in JobSession s);
  void closeReservationSession(in ReservationSession s);
  void closeMonitoringSession(in MonitoringSession s);
  void destroyJobSession(in string sessionName);
  void destroyReservationSession(in string sessionName);
  StringList getJobSessions();
  StringList getReservationSessions();
  void registerEventNotification(in DrmaaCallback callback);
};
```

The `SessionManager` interface is the main interface for establishing communication with a given DRM system. By the help of this interface, sessions for job management, monitoring, and/or reservation management can be maintained.

Job and reservation sessions maintain persistent state information (about jobs and reservations created) between application runs. State data SHOULD be persisted by the library implementation or the DRMS itself (if supported). The data SHOULD be written to stable storage when the session is closed by the according method in the `SessionManager` interface.

The re-opening of a session MUST work on the machine where the session was originally created. Implementations MAY also offer to re-open the session on another machine, if the state information is accessible.

The state information SHOULD be kept until the job or reservation session is explicitly reaped by the according destroy method in the `SessionManager` interface. If an implementation runs out of resources for storing the session information, the closing function SHOULD throw an `OutOfResourceException`. If an application ends without closing the session properly, the behavior is unspecified.

An implementation MUST allow the application to have multiple sessions of the same or different types instantiated at the same time. This includes the proper coordination of parallel calls to session methods that share state information.

A `SessionManager` instance SHALL be available as singleton at DRMAA application start. Language bindings MAY realize this by mapping the session manager operations to global functions.

(See footnote)[30]

7.1.1 drmsName

A system identifier denoting a specific type of DRM system, e.g., “LSF” or “GridWay”. Implementations SHOULD NOT make versioning information of the particular DRM system a part of this attribute value.

[30] Comparison to DRMAA 1.0: The concept of a factory from GFD.130 was removed (solves issue #6276). Version 2.0 of DRMAA supports restartable sessions by the newly introduced SessionManager interface. It allows creating multiple concurrent sessions for job submission (solves issue #2821), which can be restarted by their generated session name (solves issue #2820). Session.init() and Session.exit() functionalities are moved to the according session creation and closing routines. The descriptions were fixed accordingly (solves issue #2822). The AlreadyActiveSession error was removed. (F2F meeting July 2009) The drmaaImplementation attribute from DRMAA 1.0 was removed, since it was redundant to the drmsInfo attribute. This one is now available in the new SessionManager interface. (F2F meeting July 2009).

The value should only be utilized for informative output to the end user.

7.1.2 drmsVersion

This attribute provides the DRM-system specific version information.

The value should only be utilized for informative output to the end user.

7.1.3 drmaaName

This attribute contains a string identifying the vendor of the DRMAA implementation.

The value should only be utilized for informative output to the end user.

New, needs group approval

7.1.4 drmaaVersion

A combination of minor / major version number information for the DRMAA implementation. The major version number MUST be the constant value "2", the minor version number SHOULD be used by the DRMAA implementation for expressing its own versioning information.

The minor version number should only be utilized for informative output to the end user.

7.1.5 createJobSession / createReservationSession / createMonitoringSession

The method creates a new session instance of the particular type for the application. On successful completion of this method, the necessary initialization for making the session usable MUST be completed. Examples are the connection establishment from the DRMAA library to the DRM system, or the prefetching of information from non-thread-safe operating system calls, such as `getHostByName`.

The `contactString` parameter is an implementation-dependent string that SHALL allow the application to specify which DRM system instance to use. A contact string represents a specific installation of a specific DRM system, e.g., a Condor central manager machine at a given IP address, or a Grid Engine 'root' and 'cell'. Contact strings are always implementation dependent and therefore opaque to the application. If `contactString` has the value `UNSET`, a default DRM system SHOULD be contacted. The manual configuration or automated detection of a default contact is implementation-specific.

The `sessionName` parameter denotes a unique name to be used for the new session. If a session with such a name was created before, the method MUST throw an `InvalidArgumentException`. In all other cases, including if the provided name has the value `UNSET`, a new session MUST be created with a unique name generated by the implementation.

What means -before-

Should we state that is enough that session names must be unique for tuple (DRMS,user) ?

A `MonitoringSession` instance has no persistent state, and therefore does not support the name concept.

If the DRM system does not support advance reservation, than `createReservationSession` SHALL throw an `UnsupportedOperationException`.

If MonitoringSession is a singleton, we can get rid of the creation function at all. Currently, it is confusing that there is no destruction method. We might also rename it to open().

7.1.6 openJobSession / openReservationSession

The method is used to open a persisted `JobSession` or `ReservationSession` instance that has previously been created under the given `sessionName`. The implementation MUST support the case that the session have been created by the same application or by a different application running on the same machine. The implementation MAY support the case that the session was created or updated on a different machine. If no session with the given `sessionName` exists, an `InvalidArgumentException` MUST be raised.

If the session described by `sessionName` was already opened before, implementations MAY return the same job or reservation session instance.

If the DRM system does not support advance reservation, `openReservationSession` SHALL throw an `UnsupportedOperationException`.

7.1.7 closeJobSession / closeReservationSession / closeMonitoringSession

The method MUST perform the necessary action to disengage from the DRM system. It SHOULD be callable only once, by only one of the application threads. This SHOULD be ensured by the library implementation. Additional calls beyond the first SHOULD lead to a `NoActiveSessionException` error notification.

For `JobSession` or `ReservationSession` instances, the according state information MUST be saved to some stable storage before the method returns. This method SHALL NOT affect any jobs or reservations in the session (e.g., queued and running jobs remain queued and running).

If the DRM system does not support advance reservation, `closeReservationSession` SHALL throw an `UnsupportedOperationException`.

Allow the language binding to implicitly call close on session object destruction, or to add a close method to the according session objects.

7.1.8 destroyJobSession / destroyReservationSession

The method MUST do whatever work is required to reap persistent session state and cached job state information for the given session name. It is intended to be used when no session instance with this particular name is open. If session instances for the given name exist, they MUST become invalid after this method was finished successfully. Invalid sessions MUST throw `InvalidSessionException` on every attempt of utilization. This method SHALL NOT affect any jobs or reservations in the session in their operation, e.g., queued and running jobs remain queued and running.

If the DRM system does not support advance reservation, `destroyReservationSession` SHALL throw an `UnsupportedOperationException`.

7.1.9 getJobSessions / getReservationSessions

This method returns a list of `JobSession` or `ReservationSession` names that are valid input for a `openJobSess` or `openReservationSession` call.

If the DRM system does not support advance reservation, `getReservationSessions` SHALL throw an `UnsupportedOperationException`.

All getXYZ methods in the API return XYZ, apart from these two which return the name of XYZ. IMHO, they should either return XYZ, or should be called listXXX

.

7.1.10 registerEventNotification

This method is used to register a `DrmaaCallback` interface (see Section 8.3) implemented by the DRMAA-based application. If the callback functionality is not supported by the DRMAA implementation, the method SHALL raise an `UnsupportedOperationException`, and the capability `DrmaaCapability::CALLBACK` MUST NOT be indicated (see Section 4.5). Implementations with callback support SHOULD allow to perform multiple registration calls, which updates the callback target function.

If the argument of the method call is `UNSET`, the currently registered callback MUST be unregistered. After this method call returned, no more events SHALL be delivered to the application. If no callback target is registered, the method should return immediately.

> A language binding specification MUST define how the reference to an interface-compliant method can be given as argument to this method. It MUST also clarify how to pass an `UNSET` callback method reference.

8 Working with Jobs

A DRMAA job represents a single computational activity that is executed by the DRM system on one or more execution hosts, as one or more operating system processes. The `JobSession` interface represents all control and monitoring functions commonly available in DRM systems for such jobs as a whole, while the `Job` interface represents the common functionality for single jobs. Sets of jobs resulting from a bulk submission are separately represented by the `JobArray` interface. `JobTemplate` instances allow to formulate conditions and requirements for the job execution by the DRM system.

8.1 The DRMAA State Model

DRMAA defines the following job states:

```
enum JobState {
  UNDETERMINED, QUEUED, QUEUED_HELD, RUNNING, SUSPENDED, REQUEUED,
  REQUEUED_HELD, DONE, FAILED};
```

UNDETERMINED: The job status cannot be determined. This is a permanent issue, not being solvable by querying again for the job state.

QUEUED: The job is queued for being scheduled and executed.

QUEUED_HELD: The job has been placed on hold by the system, the administrator, or the submitting user.

RUNNING: The job is running on an execution host.

SUSPENDED: The job has been suspended by the user, the system or the administrator.

REQUEUED: The job was re-queued by the DRM system, and is eligible to run.

REQUEUED_HELD: The job was re-queued by the DRM system, and is currently placed on hold.

DONE: The job finished without an error.

FAILED: The job exited abnormally before finishing.

If a DRMAA job state has no representation in the underlying DRMS, the DRMAA implementation MAY never report that job state value. However, all DRMAA implementations MUST provide the `JobState` enumeration as given here. An implementation SHOULD NOT return any job state value other than those defined in the `JobState` enumeration.

The status values relate to the DRMAA job state transition model, as shown in Figure 1.

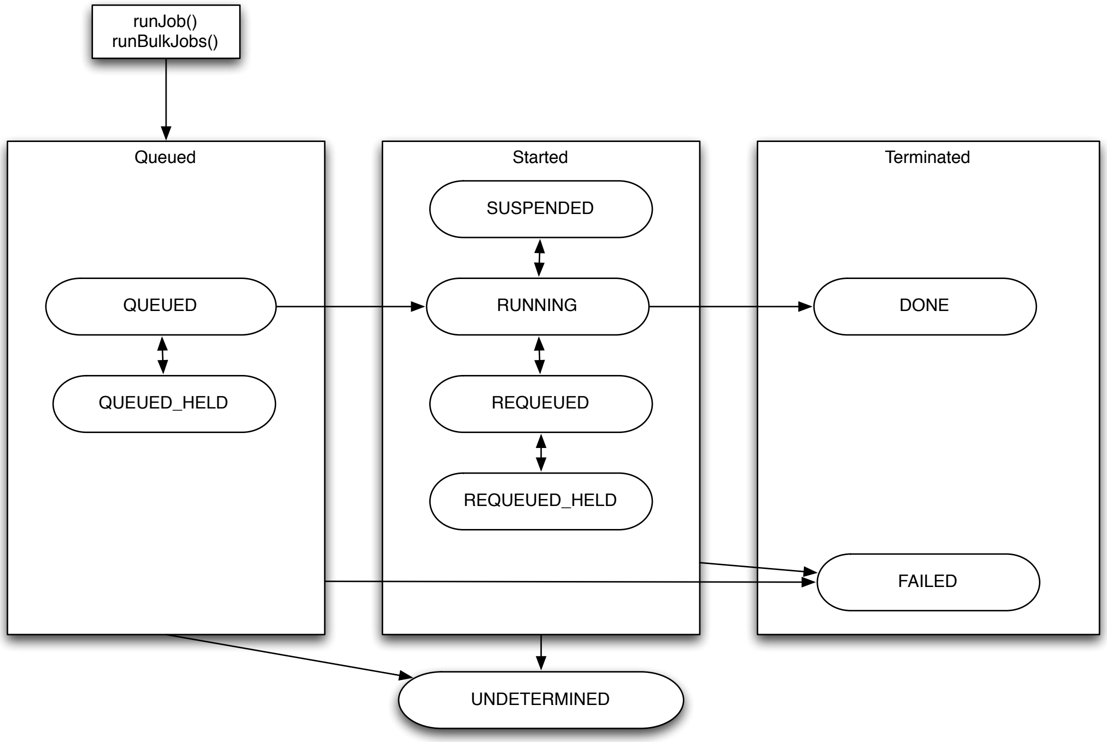

Figure 1: DRMAA Job State Transition Model

The transition diagram in Figure 1 expresses the classification of possible job states into “Queued”, “Started”, and “Terminated”. This is relevant for the job waiting functions (see Section 8.2 and Section 8.4), which operate on job state classes only. The “Terminated” class of states is final, meaning that further state transition is not allowed.

Implementations SHALL NOT introduce other job transitions (e.g., from `RUNNING` to `QUEUED`) beside the ones stated in Figure 1, even if they might happen in the underlying DRM system. In this case, implementations MAY emulate the necessary intermediate steps for the DRMAA-based application.

When an application requests job state information, the implementation SHOULD also provide the `subState` value to explain DRM-specific information about the job state. The possible values of this attribute are implementation-specific, but should be documented properly. Examples are extra states for staging phases or details on the hold reason. Implementations SHOULD define a DRMS-specific data structure for the sub-state information that can be converted to / from the data type defined by the language binding.

The IDL definition declares the sub state attributes as type `any`, expressing the fact that the language binding MUST map the data type to a generic language type (e.g., *void**, *Object*) that maintains source code portability across DRMAA implementations and still accepts an `UNSET` value.

The DRMAA job state model can be mapped to other high-level API state models. Table 4 gives a non-normative set of examples.

DRMAA JobState	SAGA JobState [3]	OGSA-BES Job State [2]
UNDETERMINED	N/A	N/A
QUEUED	Running	Pending (Queued)
QUEUED_HELD	Running	Pending (Queued)
RUNNING	Running	Running (Executing)
SUSPENDED	Suspended	Running (Suspended)
REQUEUED	Running	Running (Queued)
REQUEUED_HELD	Running	Running (Queued)
DONE	Done	Finished
FAILED	Cancelled, Failed	Cancelled, Failed

Table 4: Example Mapping of DRMAA Job States

(See footnote)[31]

8.2 JobSession Interface

A job session instance acts as container for job instances controlled through the DRMAA API. The session methods support the submission of new jobs, the monitoring and the control of existing jobs. The relationship between jobs and their session MUST be persisted, as described in Section 7.1.

```
interface JobSession {
  readonly attribute string contact;
  readonly attribute string sessionName;
  readonly attribute StringList jobCategories;
  JobList getJobs(in JobInfo filter);
  JobArray getJobArray(in string jobArrayId);
  Job runJob(in JobTemplate jobTemplate);
  JobArray runBulkJobs(
```

[31] Comparison to DRMAA 1.0:

The differentiation between the system hold, user hold, and system / user hold job states was removed (conf. call Jan 20th 2009). There is only one hold state now. A job can now change its state from one of the SUSPENDED states to the QUEUED_ACTIVE state (conf. call Jan 20th 2009, solves issue #2788). The job state UNDETERMINED is now clearer defined. It expressed a permanent issue, meaning that the job state will not change by just waiting. Temporary problems in the detection of the job state are now expressed by the TryLaterException (conf. call Feb 5th 2009, solves issue #2783). The description of the FAILED state was extended to support a more specific differentiation between different job failure reasons. The new subState feature allows the DRMAA implementation to provide better information, if available. There was no portable way of standardizing extended failure information in a better way. (conf. call May 12th 2009, solves issue #5875) The different suspend job states from DRMAA1 (user suspended, system suspended, user / system suspended) are now combined into one suspend state. DRM systems with the need to express the different suspend reasons can use the new sub-state feature (conf. call Mar 5th 2010).

REQUEUED and REQUEUED_HELD maps to RUNNING in BES, since BES does not allow a transition between Running and Pending (mailing list, APr. 2011)

```
        in JobTemplate jobTemplate,
        in long beginIndex,
        in long endIndex,
        in long step,
        in long maxParallel);
    Job waitAnyStarted(in JobList jobs, in TimeAmount timeout);
    Job waitAnyTerminated(in JobList jobs, in TimeAmount timeout);
  };
```

(See footnote)[32]

8.2.1 contact

This attribute contains the `contact` value that was used in the `SessionManager::createJobSession` call for this instance (see Section 7.1). If no value was originally provided, the default contact string from the implementation MUST be returned. This attribute is read-only.

8.2.2 sessionName

This attribute contains the `sessionName` value that resulted from the `SessionManager::createJobSession` or `SessionManager::openJobSession` call for this instance (see Section 7.1). This attribute is read-only.

8.2.3 jobCategories

This method provides the list of valid job category names which can be used for the `jobCategory` attribute in a `JobTemplate` instance. Further details about job categories are described in Section 1.4.

8.2.4 getJobs

This method returns a sequence of jobs that belong to the job session. The `filter` parameter allows one to choose a subset of the session jobs as return value. The semantics of the `filter` argument are explained in Section 5.5. If no job matches or the session has no jobs attached, the method MUST return an empty sequence instance. If `filter` is `UNSET`, all session jobs MUST be returned.

Time-dependent effects of this method, such as jobs no longer matching to filter criteria on evaluation time, are implementation-specific. The purpose of the filter parameter is to keep scalability with a large number of jobs per session. Applications therefore must consider the possibly changed state of jobs during their evaluation of the method result.

[32] Comparison to DRMAA 1.0: The original separation between synchronize() and wait() was replaced by a complete new synchronization semantic in the API. DRMAA2 has now two methods, waitStarted() and waitTerminated(). The first waits for any state that expresses that the job was started, the second for any terminal status. Both methods are available on session level (wait for any of the given jobs to start / end) or on single job level (solves issue #5880 and #2838). The function returns always a Job object, in order to allow chaining, e.g. job.wait(JobStatus.RUNNING).hold(). The session-level functions implement the old DRMAA wait(SESSION_ANY). The old synchronize() semantics are no longer directly supported - instead, the DRMAA application should use a looped `Job.wait... / JobSession.waitAny...` call. The result is a more condensed and responsive API, were the application can decide to keep the user informed during synchronization on a set of jobs. DRMAA library implementations should also become easier to design, since the danger of multithreading side effects inside the DRMAA API is reduced by this change. As a side effect, JOB_IDS_SESSION_ANY and JOB_IDS_SESSION_ALL are no longer needed. The special consideration of a partial failures during SESSION_ALL wait activities is also no longer necessary (F2F meeting July 2009). The JobSession now allows to fetch also information about jobs that were not submitted through DRMAA (conf. call June 23th 2010).

8.2.5 getJobArray

This method returns the `JobArray` instance with the given ID. If the session does not / no longer contain the according job array, `InvalidArgumentException` SHALL be thrown.

(See footnote)[33]

8.2.6 runJob

The `runJob` method submits a job with the attributes defined in the job template parameter. It returns a `Job` object that represents the job in the underlying DRM system. Depending on the job template settings, submission attempts may be rejected with an `InvalidArgumentException`. The error details SHOULD provide further information about the attribute(s) responsible for the rejection.

When this method returns a valid `Job` instance, the following conditions SHOULD be fulfilled:

- The job is part of the persistent state of the job session.
- All non-DRMAA and DRMAA interfaces to the DRM system report the job as being submitted to the DRM system.
- The job has one of the DRMAA job states.

8.2.7 runBulkJobs

The `runBulkJobs` method creates a set of parametric jobs, each with attributes defined in the given job template. Each job in the set is identical, except for the job template attributes that include the `JobTemplatePlaceholder::PARAMETRIC_INDEX` macro (see Section 5.7).

If any of the resulting parametric job templates is not accepted by the DRM system, the method call MUST raise an `InvalidArgumentException`. No job from the set SHOULD be submitted in this case.

The first job in the set has an index equal to the `beginIndex` parameter of the method call. The smallest valid value for `beginIndex` is 1. The next job has an index equal to `beginIndex + step`, and so on. The last job has an index equal to `beginIndex + n * step`, where n is equal to`(endIndex - beginIndex) / step`. The index of the last job may not be equal to `endIndex` if the difference between `beginIndex` and `endIndex` is not evenly divisible by step. The `beginIndex` value must be less than or equal to the `endIndex` value, and only positive index numbers are allowed, otherwise the method SHOULD raise an `InvalidArgumentException`.

Jobs can determine the index number at run time with the mechanism described in Section 8.6.

The `maxParallel` parameter allows to specify how many of the bulk job's instances are allowed to run in parallel on the utilized resources. Implementations MAY consider this value if the DRM system supports such functionality, otherwise the parameter MUST be silently ignored. If the parameter value is `UNSET`, no limit SHOULD be applied on the bulk job. If given, the support MUST be expressed by the `DrmaaCapability::BULK_JOBS_MAXPARALLEL` capability flag (see Section 4.5).

The `runBulkJobs` method returns a `JobArray` (see Section 8.5) instance that represents the set of `Job` objects created by the method call under a common array identifier. For each of the jobs in the array, the same conditions as for the result of `runJob` SHOULD apply.

[33] June 2011 conf. call decided to not support JobArray filtering in the session at this point. The face-to-face meeting in June 2011 identified that DRM systems typically do not support the identification of bulk jobs in the system, so it would be hard to implement the according reporting function.

The largest (syntactically) allowed value for endIndex MUST be defined by the language binding.

Further restrictions on the maximum endIndex MAY be implied by the implementation.

(See footnote)[34]

8.2.8 waitAnyStarted / waitAnyTerminated

The waitAnyStarted method blocks until any of the jobs referenced in the jobs parameter entered one of the "Started" states. The waitAnyTerminated method blocks until any of the jobs referenced in the jobs parameter entered one of the "Terminated" states (see Section 8.1). If the input list contains jobs that are not part of the session, waitAnyStarted SHALL fail with an InvalidArgumentException.

The timeout argument specifies the desired behavior when a result is not immediately available. The constant value INFINITE_TIME may be specified to wait indefinitely for a result. The constant value ZERO_TIME may be specified to return immediately. Alternatively, a number of seconds may be specified to indicate how long to wait for a result to become available. If the invocation exits on timeout, an TimeoutException SHALL be raised.

An application waiting for some condition to happen in *all* jobs of a set is expected to perform looped calls of these waiting functions.

(See footnote)[35]

8.3 DrmaaCallback Interface

The DrmaaCallback interface allows the DRMAA library or the DRM system to inform the application about relevant events from the DRM system in an asynchronous fashion. One expected use case is continuous monitoring of job state transitions. However, the implementation MAY decide to not deliver all events occurring in the DRM system. The support for such callback functionality is optional, indicated by DrmaaCallback::CALLBACK, but all implementations MUST define the DrmaaCallback interface type as given in the language binding.

```
interface DrmaaCallback {
  void notify(in DrmaaNotification notification);
};

struct DrmaaNotification {
  DrmaaEvent event;
  Job job;
  JobState jobState;
```

[34] There was a discussion (mailing list Jan 2011) about having specialized job templates for bulk submission, with support for the start / end index and a slots limit. We rejected that, since job templates are intended for re-usage.

The May 4th 2011 conf call identified Grid Engine, Torque and LSF as the only systems having support for maxParallel. The feature was determined as critical enough for still adding it, therefore the ignorance rule and the MAY semantics are applied.

[35] People typically ask for the waitAll..() counterparts of these functions. Since they are so easy to implement in the application itself, we could not see any benefit in adding them. Due to their intended long-blocking operation, the DRM system would no be able to offer any better (meaning much faster) implementation to be wrapped by DRMAA.

A section on synchronization of multi-threaded parallel wait calls was removed. This would complicate DRMAA implementations, since synchronization does not map to the obvious state polling approach. An optimization like this would be classically a task of application-oriented APIs - so, Andre has to solve it.

```
  };

  enum DrmaaEvent {
    NEW_STATE, MIGRATED, ATTRIBUTE_CHANGE
  };
```

The application callback interface is registered through the `SessionManager::registerEventNotification` method (see Section 7.1). The `DrmaaNotification` structure represents the notification information from the DRM system. Implementations MAY extend this structure for further information (see Section 5). All given information SHOULD be valid at least at the time of notification generation. The `DrmaaNotification::jobState` attribute expresses the state of the job at the time of notification generation, while the `DrmaaNotification::job` attribute allows to retrieve latest job information.

The `DrmaaEvent` enumeration defines standard event types for notification:

NEW_STATE The job entered a new state, which is described in the `jobState` attribute of the notification structure.

MIGRATED The job was migrated to another execution host, and is now in the given state.

ATTRIBUTE_CHANGE A monitoring attribute of the job, such as the memory consumption, changed to a new value. The `jobState` attribute MAY have the value UNSET on this event.

DRMAA implementations SHOULD protect themselves from unexpected behavior of the called application. This includes indefinite delays or unexpected exceptions from the callee. The implementation SHOULD prevent a nested callback at the time of occurrence, and MAY decide to deliver the according events at a later point in time.

Scalability issues of the notification facility are out of scope for this specification. Implementations MAY decide to support non-standardized throttling configuration options.

(See footnote)[36]

8.4 Job Interface

Every job in the `JobSession` is expressed by an own instance of the `Job` interface. It allows one to instruct the DRM system for a job status change, and to query the status attributes of the job in the DRM system. Implementations MAY return `Job` objects for jobs created outside of a DRMAA session.

```
  interface Job {
    readonly attribute string jobId;
    readonly attribute JobSession session;
    readonly attribute JobTemplate jobTemplate;
    void suspend();
    void resume();
    void hold();
    void release();
    void terminate();
    JobState getState(out any jobSubState);
```

[36] We intentionally did not add `subState` to the notification information, since this would make callback interface implementations specific for the DRM system, without any chance for creating a portable DRMAA application.

```
    JobInfo getInfo();
    Job waitStarted(in TimeAmount timeout);
    Job waitTerminated(in TimeAmount timeout);
  };
```

(See footnote)[37]

8.4.1 jobId

This attribute provides the string job identifier assigned to the job by the DRM system. It is intended as performant alternative for fetching a complete `JobInfo` instance for this information.

8.4.2 session

This attribute offers a reference to the `JobSession` instance that represents the session used for the job submission creating this `Job` instance.

8.4.3 jobTemplate

This attribute provides a reference to a `JobTemplate` instance that has equal values to the one that was used for the job submission creating this `Job` instance.

For jobs created outside of a DRMAA session, implementations MUST also return a `JobTemplate` instance, which MAY be empty or only partially filled.

8.4.4 suspend / resume / hold / release / terminate

The job control functions allow modifying the status of the single job in the DRM system, according to the state model presented in Section 8.1.

The `suspend` method triggers a transition from `RUNNING` to `SUSPENDED` state. The `resume` method triggers a transition from `SUSPENDED` to `RUNNING` state. The `hold` method triggers a transition from `QUEUED` to `QUEUED_HELD`, or from `REQUEUED` to `REQUEUED_HELD` state. The `release` method triggers a transition from `QUEUED_HELD` to `QUEUED`, or from `REQUEUED_HELD` to `REQUEUED` state. The `terminate` method triggers a transition from any of the "Started" states to one of the "Terminated" states. If the job is in an inappropriate state for the particular method, the method MUST raise an `InvalidStateException`.

The methods SHOULD return after the action has been acknowledged by the DRM system, but MAY return before the action has been completed. Some DRMAA implementations MAY allow this method to be used to control jobs submitted externally to the DRMAA session, such as jobs submitted by other DRMAA sessions in other DRMAA implementations or jobs submitted via native utilities. This behavior is implementation-specific.

[37] In comparison to DRMAA v1.0, DRMAA2 replaces the identification of jobs by strings with Job objects. This enables a tighter integration of job meta-data and identity, for the price of reduced performance in (so far not existing) DRMAA RPC scenarios. The former DRMAA control() with the JobControlAction structure is now split up into dedicated functions (such as hold() and release()) on the Job object.

Even though the DRMAAv2 surveys showed interest in interactive job support, this feature was intentionally left out. Reasons are the missing support in some major DRM systems, and the lack of a relevant DRMAA-related use case (conf. call Jan 7th 2010)

Issue #5877 (support for direct job signaling) was rejected, even though there was an according request from the SAGA WG. Issue #2782 (change attributes of submitted, but pending jobs) was rejected based on group decision.

8.4.5 getState

This method allows one to gather the current status of the job according to the DRMAA state model, together with an implementation specific sub state (see Section 8.1). It is intended as performant alternative for fetching a complete `JobInfo` instance for state checks. The timing conditions are described in Section 5.5.

(See footnote)[38]

8.4.6 getInfo

This method returns a `JobInfo` instance for the particular job under the conditions described in Section 5.5.

8.4.7 waitStarted / waitTerminated

The `waitStarted` method blocks until the job entered one of the “Started” states. The `waitTerminated` method blocks until the job entered one of the “Terminated” states (see Section 8.1). The `timeout` argument specifies the desired behavior when a result is not immediately available. The constant value `INFINITE_TIME` may be specified to wait indefinitely for a result. The constant value `ZERO_TIME` may be specified to return immediately. Alternatively, a number of seconds may be specified to indicate how long to wait for a result to become available. If the invocation exits on timeout, an `TimeoutException` SHALL be raised. If the job is in an inappropriate state for the particular method, the method MUST raise an `InvalidStateException`.

8.5 JobArray Interface

The following section explains the methods and attributes defined in the `JobArray` interface. An instance of this interface represent a *job array*, a common concept in many DRM systems for a set of jobs created by one operation. In DRMAA, `JobArray` instances are only created by the `runBulkJobs` operation (see Section 8.2). `JobArray` instances differ from the `JobList` data structure due to their potential for representing a DRM system concept, while `JobList` is a DRMAA-only concept realized by language binding support. Implementations SHOULD realize the `JobArray` functionality as wrapper for DRM system job arrays, if possible. If the DRM system has only single job support or incomplete job array support with respect to the DRMAA-provided functionality, implementations MUST realize the `JobArray` functionality on their own, for example based on looped operations with a list of jobs.

```
interface JobArray {
  readonly attribute string jobArrayId;
  readonly attribute JobList jobs;
  readonly attribute JobSession session;
  readonly attribute JobTemplate jobTemplate;
  void suspend();
  void resume();
  void hold();
  void release();
  void terminate();
};
```

[38] The getState() function now also returns job subState information. This is intended as additional information for the given DRMAA job state, and can be used for expressing the hold state differentiation from DRMAA 1.0 (conf. call Mar 31st 2009).

(See footnote)[39]

8.5.1 jobArrayId

This attribute provides the string job identifier assigned to the job array by the DRM system. If the DRM system has no job array support, the implementation MUST generate a system-wide unique identifier for the result of the successful `runBulkJobs` operation.

8.5.2 jobs

This attribute provides the static list of jobs that are part of the job array.

(See footnote)[40]

8.5.3 session

This attribute offers a reference to a `JobSession` instance that represents the session which was used for the job submission creating this `JobArray` instance.

Mariusz: what about job objects returned in the monitoring session? Which session should be referred then?

8.5.4 jobTemplate

This attribute provides a reference to a `JobTemplate` instance that has equal values to the one that was used for the job submission creating this `JobArray` instance.

(See footnote)[41]

8.5.5 suspend / resume / hold / release / terminate

The job control functions allow modifying the status of the job array in the DRM system, with the same semantic as with the counterparts in the `Job` interface (see Section 8.4). If one of the jobs in the array is in an inappropriate state for the particular method, the method MUST raise an `InvalidStateException`.

The methods SHOULD return after the action has been acknowledged by the DRM system for all jobs in the array, but MAY return before the action has been completed. Some DRMAA implementations MAY allow this method to be used to control job arrays created externally to the DRMAA session, such as job arrays submitted by other DRMAA sessions in other DRMAA implementations or job arrays submitted via native utilities. This behavior is implementation-specific.

Mariusz: maybe we should warn here that this operation might not be atomic.

[39] We are aware of the fact that some systems (e.g., LSF at the time of writing) do not support all DRMAA control operations offered for job arrays. Since we intended to avoid optional DRMAA operations wherever we could, the text here mandates the implementation to simulate the array support on its own. For example, looping over all jobs in the array and calling "suspend" for each one is trivial to implement and fulfills the same purpose.

[40] We were asked for offering a filter support similar to JobSession here. This was rejected by discussion on the list (Jan 2011), since the number of jobs returned here is normally comparatively short. In this case, the DRM system cannot provide any benefit over the looped check in the application itself.

[41] The use case from SAGA perspective is that the user can easily resubmit the same job - just changing for example some command line parameter, but leaving the remainder fixed (mail by Andre Merzky, July 29th 2010).

8.6 The DRMAA_INDEX_VAR environment variable

DRMAA implementations SHOULD implicitly set an environment variable with the name `DRMAA_INDEX_VAR` for each submitted job. This environment variable MUST contain the name of the environment variable provided by the DRM system that holds the parametric job index. Examples are `TASK_ID` in GridEngine, `PBS_ARRAYID` in Torque, or `LSB_JOBINDEX` in LSF. By using an indirect fetching of the environment variable value, jobs are enabled to get their own parametric index regardless of the DRM system type. For DRM systems that do not set such an environment variable, `DRMAA_INDEX_VAR` SHOULD not be set.

An expected implementation strategy would be the transparent addition an environment variable specification on job submission. However, this definition SHOULD NOT be visible for the application in the `JobTemplate` instances. If the application defines its own `DRMAA_INDEX_VAR` environment variable, it SHOULD override the implementation-defined value.

9 Working with Advance Reservation

Advance reservation is a DRM system concept that allows the reservation of execution resources for jobs to be submitted. DRMAA encapsulates such functionality of a DRM system with the interfaces and data structures described in this chapter.

DRMAA implementations for DRM systems that do not support advance reservation still MUST implemented the described interfaces, in order to keep source code portability for DRMAA-based applications. Support for advance reservation is expressed by the `DrmaaCapability::ADVANCE_RESERVATION` flag (see Section 4.5). If no support is given by the implementation, all methods related to advance reservation MUST raise an `UnsupportedOperationExeption` if being used.

9.1 ReservationSession Interface

Every `ReservationSession` instance represents a set of advance reservations in the DRM system. Every `Reservation` instance SHALL belong only to one `ReservationSession` instance.

```
interface ReservationSession {
  readonly attribute string contact;
  readonly attribute string sessionName;
  Reservation getReservation(in string reservationId);
  Reservation requestReservation(in ReservationTemplate reservationTemplate);
  ReservationList getReservations();
};
```

9.1.1 contact

This attribute contains the `contact` value that was used in the `createReservationSession` call for this instance (see Section 7.1). If no value was originally provided, the default contact string from the implementation MUST be returned. This attribute is read-only.

9.1.2 sessionName

This attribute contains the name of the session that was used for creating or opening this `Reservation` instance (see Section 7.1). This attribute is read-only.

9.1.3 getReservation

This method returns a Reservation instance that has the given reservationId. Implementations MAY support the access to reservations created outside of a DRMAA session scope, under the same regularities as for the MonitoringSession::getAllReservations method (see Section 10.1.1). If no reservation matches, the method SHALL raise an InvalidArgumentException. Time-dependent effects of this method are implementation-specific.

9.1.4 requestReservation

The requestReservation method SHALL request an advance reservation in the DRM system with attributes defined in the provided ReservationTemplate. On a successful reservation, the method returns a Reservation instance that represents the advance reservation in the underlying DRM system.

If the current user is not authorized to create reservations, DeniedByDrmsException SHALL be raised. If the reservation cannot be performed by the DRM system due to invalid ReservationTemplate attributes, or if the demanded combination of resource demands is not available, InvalidArgumentException SHALL be raised. The exception SHOULD provide further details about the rejection cause in the extended error information (see Section 6).

Some of the requested conditions might be not fulfilled after the reservation was successfully created, for example due to execution host outages. In this case, the reservation itself SHOULD remain valid. A job using such a reservation may spend additional time in one of the non-RUNNING states. In this case, the JobInfo::jobSubState information SHOULD inform about this situation.

(See footnote)[42]

9.1.5 getReservations

This method returns the list of reservations successfully created so far in this session, regardless of their start and end time. The list of Reservation instances is only cleared in conjunction with the destruction of the actual session instance through SessionManager::destroyReservationSession (see also Section 7.1).

9.2 Reservation Interface

The Reservation interface represents attributes and methods available for an advance reservation successfully created in the DRM system. Applications MAY be able to access Reservation instances for advance reservations performed outside of a DRMAA session.

```
interface Reservation {
  readonly attribute string reservationId;
  readonly attribute ReservationSession session;
  readonly attribute ReservationTemplate reservationTemplate;
  ReservationInfo getInfo();
  void terminate();
};
```

[42] In DRMAA 2.0 we do not have an explicit state model for advance reservations, as the reservation state can be easily deducted by comparing current time with reservation start and end time. For this reason, we use the subState approach for informing the user about the described situation.

9.2.1 reservationId

The reservationId is an opaque string identifier for the advance reservation. If the DRM system has identifiers for advance reservations, this attribute SHOULD provide the according string value. If not, the DRMAA implementation MUST generate value this is unique in time and extend of the DRM system.

9.2.2 session

This attribute references the ReservationSession which was used to create the advance reservation instance.

9.2.3 reservationTemplate

This attribute provides a reference to a ReservationTemplate instance that has equal values to the one that was used for the advance reservation creating this Reservation instance. For reservations created outside of a DRMAA session, implementations MUST also return a ReservationTemplate instance, which MAY be empty or only partially filled.

9.2.4 getInfo

This method returns a ReservationInfo instance for the particular job under the conditions described in Section 5.6. This method SHOULD throw InvalidArgumentException if the reservation is already expired (i.e., its end time passed) or was terminated before.

9.2.5 terminate

This method terminates the advance reservation represented by this Reservation instance. All jobs submitted with a reference to this reservation SHOULD be terminated by the DRM system or the implementation, regardless of their current state.

10 Monitoring the DRM System

The monitoring support in DRMAA focusses on the investigation of resources and cross-session data maintained by the DRM system. In contrast, session-related information is available from the JobSession and ReservationSession instances, respectively.

10.1 MonitoringSession Interface

The MonitoringSession interface represents a set of stateless methods for fetching information about the DRM system and the DRMAA implementation itself. It MAY be used to implement DRM system monitoring tools such as qstat.

```
interface MonitoringSession {
  ReservationList getAllReservations();
  JobList getAllJobs(in JobInfo filter);
  QueueInfoList getAllQueues(in StringList names);
  MachineInfoList getAllMachines(in StringList names);
};
```

All returned data SHOULD be related to the current user running the DRMAA-based application. For example, the `getAllQueues` function MAY be reduced to only denote queues that are usable or generally accessible for the DRMAA application and user performing the query.

Because no guarantee can be made as to future accessibility, and because of cases where list reduction may demand excessive overhead in the DRMAA implementation, an unreduced or partially reduced result MAY be returned on all methods returning lists. The behavior of the DRMAA implementation in this regard should be clearly documented. In all cases, the list items MUST all be valid input for job submission or advance reservation through the DRMAA API.

10.1.1 getAllReservations

This method returns the list of all DRMS advance reservations visible for the user running the DRMAA-based application. In contrast to a `ReservationSession::getReservations` call, this method SHOULD also return reservations that were created outside of DRMAA (e.g., through command-line tools) by this user.

The DRM system or the DRMAA implementation is at liberty to restrict the set of returned reservations based on site or system policies, such as security settings or scheduler load restrictions. The returned list MAY contain reservations that were created by other users. It MAY also contain reservations that are not usable for the user.

This method SHALL raise an `UnsupportedOperationException` if advance reservation is not supported by the implementation.

10.1.2 getAllJobs

This method returns the list of all DRMS jobs visible to the user running the DRMAA-based application. In contrast to a `JobSession::getJobs` call, this method SHOULD also return jobs that were submitted outside of DRMAA (e.g., through command-line tools) by this user. The returned list MAY also contain jobs that were submitted by other users if the security policies of the DRM system allow such global visibility. The DRM system or the DRMAA implementation is at liberty, however, to restrict the set of returned jobs based on site or system policies, such as security settings or scheduler load restrictions.

Querying the DRM system for all jobs might result in returning an excessive number of `Job` objects. Implications to the library implementation are out of scope for this specification.

The method supports a `filter` argument for fetching only a subset of the job information available. Both the return value semantics and the filter semantics SHOULD be similar to the ones described for the `JobSession::getJobs` method (see Section 8.2).

> Language bindings SHOULD NOT try to solve the scalability issues by replacing the sequence type of the return value with some iterator-like solution. This approach would break the basic snapshot semantic intended for this method.

(See footnote)[43]

[43] The non-argumentation about the scalability problem was the final result of a clarification attempt. We hand this one over to the implementors. (conf call Jul 14th 2010)

10.1.3 getAllQueues

This method returns a list of queues available for job submission in the DRM system. The names from all QueueInfo instances in this list SHOULD be a valid input for the JobTemplate::queueName attribute (see Section 5.7). The result can be an empty list or might be incomplete, based on queue, host, or system policies. It might also contain queues that are not accessible for the user (because of queue configuration limits) at job submission time.

The names parameter supports restricting the result to QueueInfo instances that have one of the names given in the argument. If the names parameter value is UNSET, all QueueInfo instances should be returned.

10.1.4 getAllMachines

This method returns the list of machines available in the DRM system as execution host. The returned list might be empty or incomplete based on machine or system policies. The returned list might also contain machines that are not accessible by the user, e.g., because of host configuration limits.

The names parameter supports restricting the result to MachineInfo instances that have one of the names given in the argument. If the names parameter value is UNSET, all MachineInfo instances should be returned.

11 Annex A: Complete DRMAA IDL Specification

The following text shows the complete IDL specification for the DRMAAv2 application programming interface. The ordering of IDL constructs here has no normative meaning, but ensures the correct compilation with a standard CORBA IDL compiler for syntactical correctness checks. This demands only some additional forward declarations to resolve circular dependencies.

```
module DRMAA2 {

  enum JobState {
    UNDETERMINED, QUEUED, QUEUED_HELD, RUNNING, SUSPENDED, REQUEUED,
    REQUEUED_HELD, DONE, FAILED};

  enum OperatingSystem {
    AIX, BSD, LINUX, HPUX, IRIX, MACOS, SUNOS, TRUE64, UNIXWARE, WIN,
    WINNT, OTHER_OS};

  enum CpuArchitecture {
    ALPHA, ARM, CELL, PARISC, X86, X64, IA64, MIPS, PPC, PPC64,
    SPARC, SPARC64, OTHER_CPU};

  enum ResourceLimitType {
    CORE_FILE_SIZE, CPU_TIME, DATA_SEG_SIZE, FILE_SIZE, OPEN_FILES,
    STACK_SIZE, VIRTUAL_MEMORY, WALLCLOCK_TIME };

  enum JobTemplatePlaceholder {
    HOME_DIRECTORY,WORKING_DIRECTORY,PARAMETRIC_INDEX };
```

```
enum DrmaaEvent {
  NEW_STATE, MIGRATED, ATTRIBUTE_CHANGE
};

enum DrmaaCapability {
  ADVANCE_RESERVATION, RESERVE_SLOTS, CALLBACK,
  BULK_JOBS_MAXPARALLEL,
   JT_EMAIL, JT_STAGING, JT_DEADLINE, JT_MAXSLOTS,
  JT_ACCOUNTINGID, RT_STARTNOW,
  RT_DURATION, RT_MACHINEOS, RT_MACHINEARCH
};

typedef sequence<string> OrderedStringList;
typedef sequence<string> StringList;
typedef sequence<Job> JobList;
typedef sequence<QueueInfo> QueueInfoList;
typedef sequence<MachineInfo> MachineInfoList;
typedef sequence<SlotInfo> SlotInfoList;
typedef sequence<Reservation> ReservationList;
typedef sequence< sequence<string,2> > Dictionary;
typedef string AbsoluteTime;
typedef long long TimeAmount;
native ZERO_TIME;
native INFINITE_TIME;
native NOW;

struct JobInfo {
  string jobId;
  long exitStatus;
  string terminatingSignal;
  string annotation;
  JobState jobState;
  any jobSubState;
  OrderedStringList allocatedMachines;
  string submissionMachine;
  string jobOwner;
  long slots;
  string queueName;
  TimeAmount wallclockTime;
  long cpuTime;
  AbsoluteTime submissionTime;
  AbsoluteTime dispatchTime;
  AbsoluteTime finishTime;
};

struct ReservationInfo {
  string reservationId;
  string reservationName;
```

```
    AbsoluteTime reservedStartTime;
    AbsoluteTime reservedEndTime;
    StringList usersACL;
    long reservedSlots;
    SlotInfoList reservedMachines;
  };

  struct JobTemplate {
    string remoteCommand;
    OrderedStringList args;
    boolean submitAsHold;
    boolean rerunnable;
    Dictionary jobEnvironment;
    string workingDirectory;
    string jobCategory;
    StringList email;
    boolean emailOnStarted;
    boolean emailOnTerminated;
    string jobName;
    string inputPath;
    string outputPath;
    string errorPath;
    boolean joinFiles;
    string reservationId;
    string queueName;
    long minSlots;
    long maxSlots;
    long priority;
    OrderedStringList candidateMachines;
    long minPhysMemory;
    OperatingSystem machineOS;
    CpuArchitecture machineArch;
    AbsoluteTime startTime;
    AbsoluteTime deadlineTime;
    Dictionary stageInFiles;
    Dictionary stageOutFiles;
    Dictionary resourceLimits;
    string accountingId;
  };

  struct ReservationTemplate {
    string reservationName;
    AbsoluteTime startTime;
    AbsoluteTime endTime;
    TimeAmount duration;
    long minSlots;
    long maxSlots;
    string jobCategory;
    StringList usersACL;
    OrderedStringList candidateMachines;
    long minPhysMemory;
    OperatingSystem machineOS;
    CpuArchitecture machineArch;
  };

  struct DrmaaNotification {
    DrmaaEvent event;
    Job job;
    JobState jobState;
  };

  struct QueueInfo {
    string name;
  };

  struct Version {
    string major;
    string minor;
  };

  struct MachineInfo {
    string name;
    boolean available;
    long sockets;
    long coresPerSocket;
    long threadsPerCore;
    double load;
    long physMemory;
    long virtMemory;
    OperatingSystem machineOS;
    Version machineOSVersion;
    CpuArchitecture machineArch;
  };

  struct SlotInfo {
    string machineName;
    string slots;
  };

  exception DeniedByDrmsException {string message;};
  exception DrmCommunicationException {string message;};
  exception TryLaterException {string message;};
  exception SessionManagementException {string message;};
  exception TimeoutException {string message;};
  exception InternalException {string message;};
  exception InvalidArgumentException {string message;};
```

```
exception InvalidSessionException {string message;};
exception InvalidStateException {string message;};
exception OutOfResourceException {string message;};
exception UnsupportedAttributeException {string message;};
exception UnsupportedOperationException {string message;};

interface DrmaaReflective {
  readonly attribute StringList jobTemplateImplSpec;
  readonly attribute StringList jobInfoImplSpec;
  readonly attribute StringList reservationTemplateImplSpec;
  readonly attribute StringList reservationInfoImplSpec;
  readonly attribute StringList queueInfoImplSpec;
  readonly attribute StringList machineInfoImplSpec;
  readonly attribute StringList notificationImplSpec;

  string getInstanceValue(in any instance, in string name);
  void setInstanceValue(in any instance, in string name, in string value);
  string describeAttribute(in any instance, in string name);
};

interface DrmaaCallback {
  void notify(in DrmaaNotification notification);
};

interface ReservationSession {
  readonly attribute string contact;
  readonly attribute string sessionName;
  Reservation getReservation(in string reservationId);
  Reservation requestReservation(in ReservationTemplate reservationTemplate);
  ReservationList getReservations();
};

interface Reservation {
  readonly attribute string reservationId;
  readonly attribute ReservationSession session;
  readonly attribute ReservationTemplate reservationTemplate;
  ReservationInfo getInfo();
  void terminate();
};

interface JobArray {
  readonly attribute string jobArrayId;
  readonly attribute JobList jobs;
  readonly attribute JobSession session;
  readonly attribute JobTemplate jobTemplate;
  void suspend();
  void resume();
  void hold();
```

```
    void release();
    void terminate();
  };

  interface JobSession {
    readonly attribute string contact;
    readonly attribute string sessionName;
    readonly attribute StringList jobCategories;
    JobList getJobs(in JobInfo filter);
    JobArray getJobArray(in string jobArrayId);
    Job runJob(in JobTemplate jobTemplate);
    JobArray runBulkJobs(
        in JobTemplate jobTemplate,
        in long beginIndex,
        in long endIndex,
        in long step,
        in long maxParallel);
    Job waitAnyStarted(in JobList jobs, in TimeAmount timeout);
    Job waitAnyTerminated(in JobList jobs, in TimeAmount timeout);
  };

  interface Job {
    readonly attribute string jobId;
    readonly attribute JobSession session;
    readonly attribute JobTemplate jobTemplate;
    void suspend();
    void resume();
    void hold();
    void release();
    void terminate();
    JobState getState(out any jobSubState);
    JobInfo getInfo();
    Job waitStarted(in TimeAmount timeout);
    Job waitTerminated(in TimeAmount timeout);
  };

  interface MonitoringSession {
    ReservationList getAllReservations();
    JobList getAllJobs(in JobInfo filter);
    QueueInfoList getAllQueues(in StringList names);
    MachineInfoList getAllMachines(in StringList names);
  };

  interface SessionManager{
    readonly attribute string drmsName;
    readonly attribute Version drmsVersion;
    readonly attribute string drmaaName;
    readonly attribute Version drmaaVersion;
    boolean supports(in DrmaaCapability capability);
    JobSession createJobSession(in string sessionName,
                                in string contactString);
    ReservationSession createReservationSession(in string sessionName,
                                                in string contactString);
    MonitoringSession createMonitoringSession (in string contactString);
    JobSession openJobSession(in string sessionName);
    ReservationSession openReservationSession(in string sessionName);
    void closeJobSession(in JobSession s);
    void closeReservationSession(in ReservationSession s);
    void closeMonitoringSession(in MonitoringSession s);
    void destroyJobSession(in string sessionName);
    void destroyReservationSession(in string sessionName);
    StringList getJobSessions();
    StringList getReservationSessions();
    void registerEventNotification(in DrmaaCallback callback);
  };

};
```

12 Security Considerations

The DRMAA API does not specifically assume the existence of a particular security infrastructure in the DRM system. The scheduling scenario described herein presumes that security is handled at the point of job authorization/execution on a particular resource. It is assumed that credentials owned by the application using the API are in effect for the DRMAA implementation too.

It is conceivable an authorized but malicious user could use a DRMAA implementation or a DRMAA enabled application to saturate a DRM system with a flood of requests. Unfortunately for the DRM system this case is not distinguishable from the case of an authorized good-natured user who has many jobs to be processed. For temporary load defense, implementations SHOULD utilize the `TryLaterException`. In case of permanent issues, the implementation SHOULD raise the `DeniedByDrmsException`.

DRMAA implementers should guard against buffer overflows that could be exploited through DRMAA enabled interactive applications or web portals. Implementations of the DRMAA API will most likely require a network to coordinate subordinate DRMS; however the API makes no assumptions about the security posture provided the networking environment. Therefore, application developers should further consider the security implications of "on-the-wire" communications.

For environments that allow remote or protocol based DRMAA clients, the implementation SHOULD offer support for secure transport layers to prevent man in the middle attacks.

13 Contributors

Roger Brobst
Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, CA 95134

United States
Email: rbrobst@cadence.com

Daniel Gruber
Univa GmbH
c/o Rüter und Partner
Prielmayerstr. 3 80335 München
Germany
Email: dgruber@univa.com

Mariusz Mamoński
Poznań Supercomputing and Networking Center
ul. Noskowskiego 10
61-704 Poznań
Poland
Email: mamonski@man.poznan.pl

Daniel Templeton
Cloudera Inc.
210 Portage Avenue
Palo Alto, CA 94306
United States
Email: daniel@cloudera.com

Peter Tröger (Corresponding Author)
Hasso-Plattner-Institute at University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam
Germany
Email: peter@troeger.eu

We are grateful to numerous colleagues for support and discussions on the topics covered in this document, in particular (in alphabetical order, with apologies to anybody we have missed):

Guillaume Alleon, Ali Anjomshoaa, Ed Baskerville, Harald Böhme, Nadav Brandes, Matthieu Cargnelli, Karl Czajkowski, Piotr Domagalski, Fritz Ferstl, Paul Foley, Nicholas Geib, Becky Gietzel, Alleon Guillaume, Daniel S. Katz, Andreas Haas, Tim Harsch, Greg Hewgill, Rayson Ho, Eduardo Huedo, Dieter Kranzmüller, Krzysztof Kurowski, Peter G. Lane, Miron Livny, Ignacio M. Llorente, Martin v. Löwis, Andre Merzky, Thijs Metsch, Ruben S. Montero, Greg Newby, Steven Newhouse, Michael Primeaux, Greg Quinn, Hrabri L. Rajic, Martin Sarachu, Jennifer Schopf, Enrico Sirola, Chris Smith, Ancor Gonzalez Sosa, Douglas Thain, John Tollefsrud, Jose R. Valverde, and Peter Zhu.

Special thanks must go the Andre Merzky, who participated as SAGA working group representative in numerous DRMAA events.

14 Intellectual Property Statement

The OGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat.

The OGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the OGF Executive Director.

15 Disclaimer

This document and the information contained herein is provided on an "as-is" basis and the OGF disclaims all warranties, express or implied, including but not limited to any warranty that the use of the information herein will not infringe any rights or any implied warranties of merchantability or fitness for a particular purpose.

16 Full Copyright Notice

17 References

[1] Scott Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), March 1997. URL `http://tools.ietf.org/html/rfc2119`.

[2] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. OGSA Basic Execution Service v1.0 (GFD-R.108), nov 2008.

[3] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid Applications (SAGA) Version 1.1 (GFD-R-P.90), jan 2008.

[4] Object Management Group. Common Object Request Broker Architecture (CORBA) Specification, Version 3.1. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF, jan 2008.

[5] The IEEE and The Open Group. The Open Group Base Specifications Issue 6 IEEE Std 1003.1. http://www.opengroup.org/onlinepubs/000095399/utilities/ulimit.html.

[6] Distributed Management Task Force (DMTF) Inc. CIM System Model White Paper CIM Version 2.7, jun 2003.

[7] Hrabri Rajic, Roger Brobst, Waiman Chan, Fritz Ferstl, Jeff Gardiner, Andreas Haas, Bill Nitzberg, Daniel Templeton, John Tollefsrud, and Peter Tröger. Distributed Resource Management Application API Specification 1.0 (GFD-R.022), aug 2007.

[8] Hrabri Rajic, Roger Brobst, Waiman Chan, Fritz Ferstl, Jeff Gardiner, Andreas Haas, Bill Nitzberg, Daniel Templeton, John Tollefsrud, and Peter Tröger. Distributed Resource Management Application API Specification 1.0 (GWD-R.133), jun 2008.

[9] Peter Tröger, Daniel Templeton, Roger Brobst, Andreas Haas, and Hrabri Rajic. Distributed Resource Management Application API 1.0 - IDL Specification (GFD-R-P.130), apr 2008.

[10] Peter Tröger, Hrabri Rajic, Andreas Haas, and Piotr Domagalski. Standardised job submission and control in cluster and grid environments. *International Journal of Grid and Utility Computing*, 1: 134–145, dec 2009. doi: {http://dx.doi.org/10.1504/IJGUC.2009.022029}.