Distributed Resource Management Application API (DRMAA) Working Group

Peter Tröger, Humboldt University (editor)
Daniel Templeton, Sun Microsystems
Hrabri Rajic, Intel Americas Inc.
Roger Brobst, Cadence Design Systems

Distributed Resource Management Application API 2.0

Peter Tröger 7.7.09 00:10
Kommentar: TODO: According to survey, DRMAA2 should be aligned to OGSA-BES, SAGA, and Windows HPC. The cross-comparison is still pending.

Peter Tröger 3.9.08 13:59
Kommentar: TODO: #6275 – Define all default values.

Abstract

This document describes the common base for the Distributed Resource Management Application API v2.0 (DRMAA) bindings for procedural and object-oriented languages.

Table of Contents

… (LEFT OUT FOR EASIER CHANGE TRACKING) …

1 Introduction

Peter Tröger 3.9.08 14:04
Kommentar: TODO: Describe relation to GFD.130 / 133

This document gives an IDL description for the DRMAA interface. The specification provided by this document is completely language-independent, even though some of the examples are given in Java. Adopters of this specification are expected to derive a language-binding specification (as described in Section 2.2), which can then be centrally published by the DRMAA working group. This ensures portability for DRMAA applications in one programming language, and ensures consistent API semantics over all possible DRMAA language bindings.

1.1 Notational Conventions

In this document, the following conventions are used:

- `IDL language elements and definitions` are represented in a fixed-width font.
- *References to IDL language elements and definitions* are represented in italics.

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 [RFC 2119].

The document describes the DRMAA interface semantics with the help of OMG IDL [OMG IDL]. It includes a set of overall rules for the creation of specific language bindings for the given specification. Specific examples are given for the Java language. These examples are not normative.

1.2 Related Work

There are other relevant OGF standards in the area of job submission and monitoring. An in-depth comparison and positioning of DRMAA v1.0 is provided by a conference publication [IJGUC08].

2 General Concepts

2.1 Design Decisions

An effort has been made to choose design patterns that are not unique to a specific language. However, in some cases, various languages disagree over some points. In those cases, the most meritous approach was taken, irrespective of language.
The following text bases on the terminology of OMG IDL. For this reason, all operational semantics are described in terms of interfaces and not of classes. This concept ensures the possibility to map the described operational semantics to a variety of object-oriented, and even procedural, languages. The usage of a class concept depends on the specific language-mapping rules.
The DRMAA specification assumes that destination languages for a binding typically support the concepts of *exceptions*. If a destination language does not support the notion of exceptions (like ANSI C), the language binding SHOULD map error conditions to an appropriate consistent concept. A language binding MAY chose to model exceptions as numeric error code return values, and return values as additional output parameters of the operation.

2.2 IDL language mapping

Language binding documents based on this specification MUST define a mapping between the IDL constructs used in this specification and their specific language constructs. A language binding SHOULD NOT rely itself completely on the OMG language mapping documents available for many programming languages. It must be considered that the OMG mappings bring a huge overhead of irrelevant CORBA-related mapping rules into the specification. Therefore it must be carefully decided whether a binding decision reflects a natural and simple mapping of the intended purpose for the DRMAA interfaces. In most situations it SHOULD be enough to reuse value type mappings only and to define custom mappings for the reference types.

The language binding MUST use the described concept mapping in a consistent manner for the overall specification.

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

Access to scalar attributes (`string, Boolean, long`) MUST operate in a pass-by-value mode. An according 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.

Languages without an explicit notion of enumerations MAY map the IDL enumeration values to constant class members, enabled by the distinct naming of enumeration values.

Peter Tröger 25.6.09 19:49
Kommentar: TODO: #6277 – Relax this formulation to ease up the Python binding.

Some attributes and operation parameters are scoped ("DRMAA::"), in order to avoid naming clashes in case-insensitive programming languages. Language bindings for case-sensitive languages SHOULD omit this explicit scoping.

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

Java binding example:

IDL	*Java language*
`module` definition	`package` keyword
`interface` definition	`public abstract interface` definition
`enum` definition with enumeration members	Enumeration members become Java `int` constants in the surrounding interface definition
`string` type	`java.lang.String`
`long` type	`int`
`long long` type	`long`
`const` type	`public static final`
`boolean` type	`boolean`
`[readonly] attribute` type	Getter [and setter] methods in JavaBeans™ style, boolean readonly attribute names are prefixed with "get".
`exception` type	Class definition, derived from `java.lang.Exception`
`raises` clause	`throws` clause
`valuetype` definition	`public class` definition, may additionally implement the *Cloneable*, *Serializable*, and *Compareable* interfaces

The DRMAA specification defines specialized custom types as new value types, in order to express their intended semantics: *OrderedStringList*, *StringList*, *Dictionary* and *TimeAmount*. The language-binding author SHOULD replace these type

definitions directly with semantically equal references or value types from the according language. This MAY include the creation of new complex language types for one or more of the above concepts, depending on the context.

Java binding example:

IDL	*Java*
`StringList`	`java.util.Set`
`OrderedStringList`	`java.util.List`
`TimeAmount`	`long`
`Dictionary`	`java.util.Map`

3 Changes in comparison to GFD.130 (DRMAA 1.0)

- The module name was change to DRMAA2, in order to intentionally break backward compatibility of the interface.
- 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 concept of a factory in GFD.130 was removed (solves issue #6276).
- 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).
- The *PartialTimestamp* functionality was completely removed. Absolute date and time values are now expressed as RFC822 conformant string (conf. call Mar 31st 2009).
- 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 *jobCategory* attribute in the job template was renamed to *configurationName*, in order express the administrator-side static configuration aspect behind this attribute. The description text was clarified accordingly. The DRMAA home page is planned to contain a list of recommend *configurationName* strings and their meaning. The adoption in the field might then provide a better understanding of which configuration names can become part of the standard. (F2F meeting July 2009, solves issue #5853)

- The *nativeSpecification* attribute in the job template was renamed to *nativeOptions* for better understanding. The description text is now also more specific. (F2F meeting July 2009)
- 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 was removed, since it was redundant to the *drmsInfo* attribute. This one is now available in the new *SessionManager* interface. (F2F meeting July 2009)
- 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.
  The former *HoldInconsistentStateException, ReleaseInconsistentStateException*, *ResumeInconsistentStateException*, and *SuspendInconsistentStateException* from DRMAA v1.0 are now expressed as single *InconsistentStateException* with different meaning per dedicated function. String list for job identifiers are replaced by Job object lists (F2F meeting July 2009)
- The binding of job template attribute names and exception names to strings was removed from the main specification. Language bindings such as for the C programming languages have to define their own mapping. It is recommended to keep string identifiers from DRMAA 1.0 as far as possible.
- The original separation between *synchronize()* and *wait()* was replaced by a complete new synchronization semantic in the API. DRMAA2 has now only two *wait()* methods - one on *JobSession* level for all jobs of a session, and one on *Job* level. The job-level function allows waiting for one of a given set of job states to occur (solves issue #5880). One example is to wait for the starting of a job (solves issue #2838). Waiting for any kind of state change is expressed by an according constant. The function returns its own job object again, in order to allow chaining, e.g. *job.wait(JobStatus.RUNNING).hold()*.
  The session-level *waitAny()* implements 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 with the according state set. 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 minimized 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)
- Issue #5877 (support for direct job signaling) was rejected.

- Issue #2782 (change attributes of submitted, but pending jobs) was rejected.
- DRMAA2 supports the monitoring of execution resources through the *MonitoringSession* interface. The *JobInfo* interface was heavily extended for providing more information (solves issue #2827).
- The DRMAA *JobSession* interface can additionally support a callback facility, where the DRMAA library informs the application about state change events in the DRM system.

4 The DRMAA2 API

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

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.

The following text shows the complete IDL description of the DRMAA2 interface. Later chapters in this document explain the different parts.

```
module DRMAA2{
  // unbounded native ordered string list
  valuetype OrderedStringList sequence<string>;
  // unbounded native string list
  valuetype StringList sequence<string>;
  // dictionary type, for unbounded key-value pair storage
  valuetype Dictionary sequence< sequence<string,2> >;
  // amount of time, at least with a resolution to seconds
  valuetype TimeAmount long long;

  // Data Types (Section 5)

  enum JobState {
    UNDETERMINED, QUEUED_ACTIVE, HOLD, RUNNING,
    SYSTEM_SUSPENDED, USER_SUSPENDED,USER_SYSTEM_SUSPENDED,
    DONE, FAILED
  };

const sequence<JobState> ANY_STATE=[];
  const long long TIMEOUT_WAIT_FOREVER = -1;
  const long long TIMEOUT_NO_WAIT = 0;

  enum JobSubmissionState {
    HOLD_STATE, ACTIVE_STATE
  };

  enum DrmaaEvent {
    NEW_STATE_UNDETERMINED, NEW_STATE_QUEUED_ACTIVE,
    NEW_STATE_HOLD, NEW_STATE_RUNNING,
    NEW_STATE_SYSTEM_SUSPENDED, NEW_STATE_USER_SUSPENDED,
    NEW_STATE_USER_SYSTEM_SUSPENDED, NEW_STATE_DONE,
    NEW_STATE_FAILED};

  valuetype DrmaaNotification {
    readonly attribute DrmaaEvent event;
    readonly attribute Job job;
  };
```

```
valuetype FileTransferMode {
  attribute boolean transferInputStream;
  attribute boolean transferOutputStream;
  attribute boolean transferErrorStream;
};

valuetype Version {
  readonly attribute long major;
  readonly attribute long minor;
};

// Exceptions (Section 6)

exception AuthorizationException {string message;};
exception ConflictingAttributeValuesException
          {string message;};
exception DefaultContactStringException {string message;};
exception DeniedByDrmException {string message;};
exception DrmCommunicationException {string message;};
exception DrmsExitException {string message;};
exception DrmsInitException {string message;};
exception ExitTimeoutException {string message;};
exception InconsistentStateException {string message;};
exception IllegalStateException {string message;};
exception InternalException {string message;};
exception InvalidArgumentException {string message;};
exception InvalidAttributeFormatException {string message;};
exception InvalidAttributeValueException {string message;};
exception InvalidContactStringException {string message;};
exception InvalidJobException {string message;};
exception InvalidJobTemplateException {string message;};
exception NoActiveSessionException {string message;};
exception NoDefaultContactStringSelectedException
          {string message;};
exception OutOfMemoryException {string message;};
exception TryLaterException {string message;};
exception UnsupportedAttributeException {string message;};

// Session Manager (Section 7)

interface SessionManager{
  readonly attribute string drmsInfo;
  readonly attribute Version version;

  JobSession createJobSession(  in string sessionName,
                                in string contactString)
    raises (???);
  void closeJobSession(in JobSession s)
    raises (???);
  void destroyJobSession(in string sessionName)
```

```
    raises (???);
  string[] getJobSessions()
    raises (???);
  MonitoringSession createMonitoringSession (in string
contactString)
    raises (???);
  void closeMonitoringSession(in MonitoringSession s)
    raises (???);
};

// Job Sessions (Section 8)

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

interface JobSession{
  readonly attribute string contact;

  JobTemplate createJobTemplate()
    raises (DrmCommunicationException,
            NoActiveSessionException,
            OutOfMemoryException,
            AuthorizationException,
            InternalException);

  void deleteJobTemplate(in DRMAA::JobTemplate jobTemplate)
    raises (DrmCommunicationException,
            NoActiveSessionException,
            OutOfMemoryException,
            AuthorizationException,
            InvalidArgumentException,
            InvalidJobTemplateException,
            InternalException);

  Job runJob(in DRMAA::JobTemplate jobTemplate)
    raises (TryLaterException,
            DeniedByDrmException,
            DrmCommunicationException,
            AuthorizationException,
            InvalidJobTemplateException,
            NoActiveSessionException,
            OutOfMemoryException,
            InvalidArgumentException,
            InternalException);

  sequence<Job> runBulkJobs(
            in DRMAA::JobTemplate jobTemplate,
            in long beginIndex,
            in long endIndex,
            in long step)
```

```
    raises (TryLaterException,
            DeniedByDrmException,
            DrmCommunicationException,
            AuthorizationException,
            InvalidJobTemplateException,
            NoActiveSessionException,
            OutOfMemoryException,
            InvalidArgumentException,
            InternalException);

  sequence<Job> waitAnyStarted(in sequence<Job> jobs,
                               in long long timeout)
    raises (DrmCommunicationException,
            AuthorizationException,
            ExitTimeoutException,
            InvalidJobException,
            NoActiveSessionException,
            OutOfMemoryException,
            InvalidArgumentException,
            InternalException);


  sequence<Job> waitAnyTerminated(in sequence<Job> jobs,
                                  in long long timeout)
    raises (DrmCommunicationException,
            AuthorizationException,
            ExitTimeoutException,
            InvalidJobException,
            NoActiveSessionException,
            OutOfMemoryException,
            InvalidArgumentException,
            InternalException);


  void registerEventNotification(in DrmaaCallback callback)
    raises (UnsupportedFeatureExeption, ....);


// Job template (Section 9)

interface JobTemplate{
  const string HOME_DIRECTORY = "$drmaa_hd_ph$";
  const string WORKING_DIRECTORY = "$drmaa_wd_ph$";
  const string PARAMETRIC_INDEX = "$drmaa_incr_ph$";
  attribute string remoteCommand;
  attribute OrderedStringList args;
  attribute DRMAA::JobSubmissionState jobSubmissionState;
  attribute Dictionary jobEnvironment;
  attribute string workingDirectory;
  attribute string configurationName;
```

```
    attribute string nativeOptions;
    attribute StringList email;
    attribute boolean blockEmail;
    attribute string startTime;
    attribute string jobName;
    attribute string inputPath;
    attribute string outputPath;
    attribute string errorPath;
    attribute boolean joinFiles;
    attribute FileTransferMode transferFiles;
    attribute string deadlineTime;
    attribute TimeAmount hardWallclockTimeLimit;
    attribute TimeAmount softWallClockTimeLimit;
    attribute TimeAmount hardRunDurationLimit;
    attribute TimeAmount softRunDurationLimit;
    readonly attribute StringList attributeNames;
      …
    [language-specific operations for implementation-specific attributes]
      …

  // Job (Section 10)

interface Job{
    void suspend()  // suspend a running job
      raises (DrmCommunicationException,
              AuthorizationException,
              InconsistentStateException,
              InvalidJobException,
              NoActiveSessionException,
              OutOfMemoryException,
              InvalidArgumentException,
              InternalException);


    void resume()  // resume a suspended job
      raises (DrmCommunicationException,
              AuthorizationException,
              InconsistentStateException,
              InvalidJobException,
              NoActiveSessionException,
              OutOfMemoryException,
              InvalidArgumentException,
            InternalException);


    void hold()   // put a queued job on hold
      raises (DrmCommunicationException,
              AuthorizationException,
              InconsistentStateException,
              InvalidJobException,
              NoActiveSessionException,
              OutOfMemoryException,
              InvalidArgumentException,
```

```
            InternalException);

  void release() // release a job on hold
    raises (DrmCommunicationException,
            AuthorizationException,
            InconsistentStateException,
            InvalidJobException,
            NoActiveSessionException,
            OutOfMemoryException,
            InvalidArgumentException,
            InternalException);


  void terminate()  // terminate a running job
    raises (DrmCommunicationException,
            AuthorizationException,
            InconsistentStateException,
            InvalidJobException,
            NoActiveSessionException,
            OutOfMemoryException,
            InvalidArgumentException,
            InternalException);

  JobState getState(out native subState)
    raises (DrmCommunicationException,
            AuthorizationException,
            InvalidJobException,
            NoActiveSessionException,
            OutOfMemoryException,
            InternalException);

  JobInfo getInfo()
    raises (DrmCommunicationException,
            AuthorizationException,
            InvalidJobException,
            NoActiveSessionException,
            OutOfMemoryException,
            InternalException);

  void waitStarted(in long long timeout)
  void waitTerminated(in long long timeout)

};

// Extended information about a job (Section 11)

interface JobInfo {
  // unique job identifier
  readonly attribute string jobId;
  // ???
  readonly attribute Dictionary resourceUsage;
  // ???
```

```
    readonly attribute boolean hasExited;
    // ???
    readonly attribute long exitStatus;
    // ???
    readonly attribute boolean hasSignaled;
    // ???
    readonly attribute string terminatingSignal;
    // ???
    readonly attribute boolean hasCoreDump;
    // ???
    readonly attribute boolean wasAborted;
    // state of the job according to DRMAA model
    readonly attribute JobState jobState;
    // DRM-dependent concretization of jobState
    readonly attribute string jobSubState;
    // host name of the execution host
    readonly attribute string masterMachine;
    // participating hosts in a parallel job
    // beside the masterMachine
    readonly attribute string[] slaveMachines;
    // host name of the submission host
    readonly attribute string submissionMachine;
    // username of the job owner
    readonly attribute string jobOwner;
    // amount of time since job was started
    readonly attribute long wallclockTime;
    // remaining time until job termination
    readonly attribute long wallclockLimit;
    // amount of CPU seconds consumed
    readonly attribute long cpuTime;
    // date and time when the job was submitted
    readonly attribute long submissionTime;
    // date and time when the job was dispatched (???)
    readonly attribute long submissionTime;
    // date and time when the job started first execution
    readonly attribute long startTime;
    // date and time when the job finished execution
    readonly attribute long finishTime;


  // Support for machine monitoring (Section 12)

  interface MonitoringSessions{

    //// attributes on DRM system level ////
    readonly attribute string[] drmVersionString;

    //// attributes on session level ////
    readonly attribute string[] drmMachineNames;

    //// attributes on machine level ////
```

Peter Tröger 28.9.09 21:30

Kommentar: Consider other standards here – CIM, RUS-WG, JSDL …

```
    // number of processor sockets in the machine
    int machineSockets(in string machineName);
    // number of CPU cores per socket, ASMP ?
    int machineCoresPerSocket(in string machineName);
    // Normalized (?) load on the core, -1 for accumulated
    int machineLoad(in string machineName, in long coreNumber);
    // Physical memory installed in the machine
    int machinePhysMemory(in string machineName);
    // virtual memory available in the machine
    int machineVirtMemory(in string machineName);
    // Operating system on the machine, based on
    // on some standardized enumeration from somewhere else
    string machineOperatingSystem(in string machineName)
    // Processor architecture the machine, based
    // on some standardized enumeration from somewhere else
    string machineArchitecture(in string machineName)

  };
}
```

5 Data Types

5.1 JobState enumeration

Peter Tröger 29.9.09 16:40
Kommentar: The introduction of a "re-scheduled" state was proposed in one of the telcos. Needs cross-check with DRM systems.

The *JobState* enumeration is used as return value type for fetching the general job status in the DRMS. The elements have the following meaning:

- `JobState:UNDETERMINED`: The job status cannot be determined. This is a permanent issue, not being solvable by querying again for the job state.
- `JobState:QUEUED_ACTIVE`: The job is queued for being scheduled and executed.
- `JobState:HOLD`: The job has been placed on hold by the system, the administrator, or the user.
- `JobState:RUNNING`: The job is running in the DRM system.
- `JobState:SYSTEM_SUSPENDED`: The job has been suspended by the system or the administrator.
- `JobState:USER_SUSPENDED`: The job has been suspended by a user.
- `JobState:USER_SYSTEM_SUSPENDED`: The job has been suspended by both the system or administrator and a user.
- `JobState:DONE`: The job finished without an error.
- `JobState:FAILED`: The job exited abnormally before finishing.

A DRMAA language binding implementation is not required to be able to return all of the job state values in the *JobState* enumeration. If a given job state has no representation in the underlying DRMS, the DRMAA implementation MAY ignore that job state value. All DRMAA implementations MUST, however, define the *JobState* enumeration, and the definition MUST include **all** job state values, including those for unused job states. 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:

Valuator -> Queued_Active
Valuator -> Hold
Valuator -> Rejected
Queued_Active <-> Hold
Queued_Active -> Running
Running <-> System_Suspended
Running <-> User_Suspended
Running <-> User-System_Suspended
Running -> Failed
System_Suspended -> Failed
User_Suspended -> Failed
User-System_Suspended -> Failed
System_Suspended -> Queued_Active
User_Suspended -> Queued_Active
User-System_Suspended -> Queued_Active
Running -> Done

Figure 1: DRMAA Job State Transition Diagram

Peter Tröger 7.7.09 00:14
Kommentar: Redraw state transition diagram

5.2 JobSubmissionState enumeration

The *JobSubmissionState* enumeration is used as the type of the *JobTemplate::jobSubmissionState* interface attribute. In the context of the job template, the enumeration values have the following meaning:

- *HOLD_STATE:* The job may be queued, but it is not eligible to run.
- *ACTIVE_STATE:* The job is eligible to run.

5.3 FileTransferMode value type

The *FileTransferMode* value-type is used by a *JobTemplate* instance to indicate the value for the *transferFiles* attribute. The type contains three attributes, which determine the streams that will be staged in or out.

transferInputStream

This attribute defines whether to transfer an input stream file. If this attribute contains true, the transfer*inputStream* attribute of the corresponding job template SHALL be treated as the source from which the input file should be copied.

transferOutputStream

This attribute defines whether to transfer an output stream file. If this attribute contains true, the *transferOutputStream* attribute of the corresponding job template SHALL be treated as the destination to which the output file should be copied.

transferErrorStream

This attribute defines whether to transfer an error stream file. If this attribute contains true, the *transferErrorStream* attribute of the corresponding job template SHALL be treated as the destination to which the error file should be copied.

5.4 Version value type

The *Version* value type is a holding structure for the major and minor version numbers of the DRMAA language binding implementation as contained in the *version* attribute of the *SessionManager* interface. The string of a *Version* instance MUST be of the form “<major>.<minor>”.

major

This attribute SHALL contain the major version number.

minor

This attribute SHALL contain the minor version number.

6 Exceptions

All exceptions in specific bindings MUST contain a possibility to store and read a textual description of the exception cause for the exception instance.
Language bindings MAY decide to derive all exceptions from given environmental exception base class(es). Language bindings SHOULD replace exceptions with a semantically equivalent native runtime environment exception whenever this is appropriate.

Language bindings MAY decide to introduce a hierarchical ordering of the DRMAA exceptions through class derivation. In this case it MAY also happen that new exceptions are introduced for behavior aggregation. In this case, those exceptions SHALL be marked as abstract, to prevent them from being thrown.

If the language supports the distinction between static ('checked') and runtime ('unchecked') exceptions (like Java), all but the following exceptions must be represented as checked exception:

- `InternalException`
- `OutOfMemoryException`
- `InvalidArgumentException`

If a destination language does not support the notion of exceptions (like ANSI C), the language binding SHOULD map error conditions to an appropriate consistent concept. A language binding MAY chose to model exceptions as numeric error code return values, and return values as additional output parameter of the operation. Such a language binding SHOULD specify numeric values for all DRMAA error constants.

6.1 AuthorizationException

The user is not authorized to perform the given operation.

6.2 ConflictingAttributeValuesException

The value of this attribute conflicts with one or more previously set properties.

6.3 DefaultContactStringException

The DRMAA implementation could not use the default contact string to connect to DRM system.

6.4 DeniedByDrmException

The DRM system rejected the job. The job will never be accepted due to DRM configuration or job template settings.

6.5 DrmCommunicationException

Could not contact DRM system.

6.6 DrmsExitException

A problem was encountered while trying to exit the session.

6.7 DrmsInitException

A problem was encountered while trying to initialize the session.

6.8 ExitTimeoutException

The *wait()* or *synchronize()* method call on the *Session* interface returned before all selected jobs entered the *DONE or FAILED* state.

6.9 InternalException

An unexpected or internal DRMAA error occurred, for example a system call failure.

6.10 InvalidArgumentException

A parameter value is fundamentally invalid, such as being of the wrong type or being `null`.

6.11 InvalidAttributeFormatException

The value for the job template property is improperly formatted, such as a badly formatted time stamp.

6.12 InvalidAttributeValueException

The value for the job template property is invalid.

6.13 InvalidContactStringException

The given contact string is not valid.

6.14 InvalidJobException

The job specified by the given object does not exist.

6.15 InvalidJobTemplateException

The job template is not valid. It was either created incorrectly, or it has already been deleted.

6.16 NoActiveSessionException

The method call failed because there is no active session.

6.17 NoDefaultContactStringSelectedException

No default contact string was provided or selected. DRMAA requires the default contact string to be selected when there is more than one possible contact string due to multiple DRMAA implementations being present and available.

6.18 OutOfMemoryException

This exception can be thrown by any method at any time when the DRMAA implementation has run out of free memory.

6.19 InconsistentStateException

The current job state does not allow the request state change.

6.20 TryLaterException

The DRMS rejected the operation, possibly due to excessive load. A retry attempt may succeed, however.

6.21 UnsupportedAttributeException

The given job template attribute is not supported by the current DRMAA implementation.

6.22 IllegalStateException

The *JobInfo* instance is not in the correct state for this kind of operation.

7 SessionManager interface

Peter Tröger 29.9.09 15:54

Kommentar: TODO: Survey showed many requests for:
- Job workflows (but only as add-on)
  - Monitoring of jobs in the DRM system not submitted by the DRMAA session (has an security aspect)
  - List the jobs in a queue

Peter Tröger 6.7.09 23:11

Kommentar: TODO: Survey showed some interest in being able to submit jobs to specific resources

Peter Tröger 6.7.09 23:11

Kommentar: TODO: #5876 – Extend DRMAA by file transfer capabilities

DRMAA supports the concept of sessions, which act as container for a list of either machines or jobs.

Job sessions can be re-opened after they were closed. The re-opening of a session MUST be possible on the machine where the job was originally created. Implementations MIGHT also offer to re-open sessions on another machine. If jobs terminate after closing a session, their termination information MUST be available when the jobs original session is re-opened. The session name is generated by the DRMAA implementation and can also be queried from the Session interface. Multiple concurrent sessions are allowed.

7.1 drmsInfo

A DRM system identifier denotes a specific type of DRM system, e.g. Sun Grid Engine. It allows the application the rely on implementation-specific attributes.

7.2 createJobSession

The *createJobSession()* creates a new *JobSession* object for the application, which allows to submit, monitor and control a group of jobs. The method MUST do whatever work is required to initialize a DRMAA job session for use, for example by connecting the DRMAA library to a DRMS daemon.

The *sessionName* parameter denotes a specific job session to be re-opened. If a job session with such a name was not created before, the method MUST throw an *InvalidArgumentException*. If the provided name is an empty string or `null` , a new job session MUST be created.

The *contactString* parameter is an implementation-dependent string that may be used to specify which DRM system instance to use. A *contactString* represents a specific installation of a specific DRM system, e.g. a Condor central manager machine at a given IP address or a Sun Grid Engine ‘root’ and ‘cell’. The strings are always implementation dependent and SHOULD NOT be interpreted by the application. Contact strings need to be figured out by the application user manually for every DRMS installation the application is executed upon. If contact is `null` or emtpy, the default DRM system SHOULD be used, provided there is only one DRMS available. If contact is `null` or empty, and more than one DRMAA implementation is available, *createJobSession()* SHALL throw a *NoDefaultContactStringSelectedException* or return a corresponding error code if exceptions aren't supported.

In the case that a DRMAA library implementation needs to perform non-thread-safe operations (like *getHostByName()* C library call), it SHOULD perform them in the implementation of the *createJobSession()* operation, in order to ensure thread-safe operations for all other job-related DRMAA methods.

Parameters

`sessionName` - implementation-dependent string that may be used to specify a job session to be re-opened. If `null` or empty, a new job session is created.

`contactString` - implementation-dependent string that may be used to specify which DRM system to use. If `null` or empty, the DRMAA implementation will select the default DRM system if there is only one DRMS available.

7.3 closeJobSession

The *closeJobSession()* method MUST do whatever work is required to disengage from the DRM system and finally persist the list of jobs in the session to some stable storage. This method SHALL NOT affect any jobs in the session (e.g., queued and running jobs remain queued and running). Any job template instances which have not yet been deleted become invalid after *closeJobSession()* is called, even after a subsequent call to *createJobSession()*. *closeJobSession()* SHOULD be called only once, by only one of the threads. Additional calls to *closeJobSession()* beyond the first SHOULD throw a *NoActiveSessionException* or return a corresponding error code if exceptions aren't supported.

7.4 destroyJobSession

7.5 getJobSessions

7.6 createMonitoringSession

7.7 closeMonitoringSession

8 JobSession interface

The following chapter explains the set of constants, methods and attributes defined in the *JobSession* interface. Every DRMAA *JobSession* object provides a container for the jobs submitted by its *runJob()* resp. *runBulkJobs()* methods.

An application can open more than one DRMAA session at a time, but every job can only belong to one session. Sessions (in terms of their job list) should be persisted after closing the session through *SessionManager.closeJobSession()*, until they are explicitly reaped through *SessionManager.destroyJobSession()*.

The *JobSession* interface has explicit methods for creating and destroying job template objects. Even though some object oriented programming languages might prefer implicit object destruction mechanism instead of explicit cleanup calls, this interface design reflects the close coupling of DRMAA to the underlying DRM system. It also supports the implementation of object oriented DRMAA libraries based on a DRMAA C library.

The interface provides one read-only attribute *contact*. It contains the contact string used on creation of this instance through *SessionManager*, or the default contact string if none was chosen,

8.1 createJobTemplate

The *createJobTemplate()* method SHALL return a new *JobTemplate* instance. The job template is used to set the defining characteristics for jobs to be submitted. Once the job template has been created, it should also be deleted (via *deleteJobTemplate()*) when no longer needed. Failure to do so may result in a memory leak.

Returns

The *createJobTemplate()* method SHALL return a blank *JobTemplate* instance.

8.2 deleteJobTemplate

The *deleteJobTemplate()* method is used to deallocate a job template, and SHALL perform all necessary steps required to free all memory associated with the given *JobTemplate* instance.
In languages where memory is not freed explicitly, e.g. languages that use garbage collectors, this method SHALL perform all necessary steps required to prepare this job template to be freed. In languages where finalizers are supported, the implementation of this method MAY be empty.
This method SHALL have no effect on running jobs. This method MUST only work on *JobTemplate* instances that were created with the *createJobTemplate()* method and have not previously been deleted with the *deleteJobTemplate()* method and MUST otherwise throw an *InvalidJobTemplateException.*

Parameters

`jobTemplate` - the *JobTemplate* instance to delete.

8.3 runJob

Peter Tröger 6.7.09 23:32
Kommentar: TODO: #5884 – apply solution from GFD.133 here

The *runJob()* method SHALL submit a job with attributes defined in the job template given as a parameter. This method MUST only work on *JobTemplate* instances that were created with the *createJobTemplate()* method and have not previously been deleted with the *deleteJobTemplate()* method and MUST otherwise throw an *InvalidJobTemplateException.*

Parameters

`jobTemplate` - the job template to be used to create the job.

Returns

The *runJob()* method SHOULD return a Job object that represents the job in the underlying DRM system.

8.4 runBulkJobs

Peter Tröger 6.7.09 23:32
Kommentar: TODO: #5884 – apply solution from GFD.133 here

The *runBulkJobs()* method SHALL submit a set of parametric jobs, dependent on the implied loop index, each with attributes defined in the given job template. Each job in the set is identical except for its index. The first parametric job has an index equal to *beginIndex*. 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*. Note that the value of the last job's index may not be equal to *endIndex* if the difference between *beginIndex* and *endIndex* is not evenly divisible by *step*. The smallest valid value for *beginIndex* is 1. The largest valid value for *endIndex* is language dependent. The *beginIndex* value must be less than or equal to the *endIndex* value, and only positive index numbers are allowed. The index number can be determined by the job in an implementation-specific fashion.

The *JobTemplate* interface defines a `PARAMETRIC_INDEX` placeholder for use in specifying paths. This placeholder is used to represent the individual identifiers of the tasks submitted through this method.

This method MUST only work on *JobTemplate* instances that were created by the *createJobTemplate()* method and have not previously been deleted by the *deleteJobTemplate()* or *exit()* method and MUST otherwise throw an *InvalidJobTemplateException.*

Parameters

`jobTemplate` - the job template to be used to create the job.
`beginIndex` - the starting value for the loop index.
`endIndex` - the terminating value for the loop index.
`step` - the value by which to increment the loop index each iteration.

Returns

The *runBulkJobs()* method SHOULD return a list of *Job* objects, were each of them represents a job in the underlying DRM system.

8.5 waitAny

Peter Tröger 29.9.09 16:50

Kommentar: Fix description according to new waitAnySTarted() / waitAnyTerminated() model. Check mailing list, Sept. 2nd 2009 for agreed semantics.

This method SHALL wait for any of the session jobs to enter one of the given states. If no jobs are active in the session, the call to *waitAny()* SHALL fail, throwing an *InvalidJobException*. The *timeout* value SHALL be used to specify the desired behavior when a result is not immediately available. The constant value `TIMEOUT_WAIT_FOREVER` may be specified to wait indefinitely for a result. The constant value `TIMEOUT_NO_WAIT` 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 *ExitTimeoutException* SHALL be thrown or a corresponding error code returned if exceptions aren't supported. The caller should check system time before and after this call in order to be sure of how much time has passed.

To avoid thread race conditions in multi-threaded applications, the DRMAA implementation user should explicitly synchronize this call with any other job submission or control calls that may change the number of remote jobs.

In a multi-threaded environment with a *waitAny()*, only one of the active thread gets the status change notification for a particular job, while the other threads continue waiting. If there are no more queryable jobs left in the session, all remaining waiting threads SHOULD fail with an *InvalidJobException*.

Peter Tröger 6.7.09 23:32

Kommentar: TODO: #5879 – Solution applied to GFD.133 needs to be reflected also here.

If thread A is waiting for a specific job with *Job.wait()*, and another thread, thread B, waiting for that same job or with *JobSession.waitAny(),* receives notification that the job has finished, thread A SHOULD fail with an *InvalidJobException*. At any time during a call to *waitAny()*, if no jobs are in the session, the call to *waitAny()* SHALL fail, throwing an *InvalidJobException*.

Parameters

`jobStates` - the job states to be waited for.
`timeout` - the maximum number of seconds to wait.

Returns

This method SHALL return the Job object for the job that reached one of the given states.

9 JobTemplate interface

Peter Tröger 31.1.09 00:53

Kommentar: TODO: #5881 – more optional JT attributes to support resource requirement formulation. Mostly solved by JSDL. Important question according to survey. Dan proposed a "resourceRequest" attribute of type "Dictionary", treated as mandatory, or maybe a "hardResourceRequest" vs. "softResourceRequest"

In order to define the attributes associated with a job, a DRMAA application uses the *JobTemplate* interface. Instances of such templates are created via the active *JobSession* implementation. A DRMAA application gets a *JobTemplate* from the active *JobSession* instance, specifies in the template any required job parameters, and then passes the template back to the DRMAA *JobSession* instance when requesting that a job be executed. When finished, the DRMAA application SHOULD call the *JobSession::deleteJobTemplate()* method to allow the underlying implementation to free any resources bound to the *JobTemplate* instance.

9.1 Interface overview

A language binding specification MUST model the *JobTemplate* interface in the following way:

In languages that do not support the notion of interfaces or objects, the job template attributes SHOULD be modeled as constant parameters to generic getter and setter routines. These routines SHOULD treat all attribute names and values as strings. In the case of such a language, the *attributeNames* attribute SHOULD be modeled as a *getAttributeNames()* routine that returns the names of the available attributes as a list of strings which can be used with the generic getter and setter routines. See section 0 below.

The *JobTemplate* implementation MUST support the following exceptions for the setter operations in case there is a concept of exceptions in the programming language:

- *InvalidAttributeValueException* – The value is invalid for the job template property, e.g. a *startTime* that is in the past.
- *ConflictingAttributeValuesException* – the attribute value conflicts with a previously set attribute value.

For both getter and setter operations, the following exceptions MUST be supported in case exceptions are part of the programming language:

- *NoActiveSessionException*
- *DrmCommunicationException*
- *AuthorizationException*
- *OutOfMemoryException*
- *InternalException*

In most cases, a DRMAA implementation will require that job templates be created through the *Session::createJobTemplate()* method. In those cases, passing a template created other than via this method to the *Session::deleteJobTemplate()*, *Session::runJob()*, or *Session::runBulkJobs()* methods MUST result in an *InvalidJobTemplateException* being thrown or a corresponding error code being returned if exceptions are not supported.

A *JobTemplate* instance SHOULD be convertible to a string for printing. This SHOULD be accomplished through whatever mechanism is most natural for the implementation language. The resulting string MUST contain the values of all set properties.

In the DRMAA job template concept, there is a distinction between mandatory, optional and implementation-specific attributes. A language binding implementation MUST include all DRMAA attributes described here, both required and optional. The setter and getter implementations for optional attributes MUST in case throw *UnsupportedAttributeException*. The service provider implementation SHOULD then override the setters and getters for supported optional attributes with methods that operate normally. In the case of a destination language that does not support the notion of interfaces or objects, the generic getter and setter routines should throw *UnsupportedAttributeException* when called with the name of an unknown or unsupported attribute.

Generic getter / setter routines

In the case of a destination language that does not support the notion of interfaces or objects, the *JobTemplate* interface SHOULD be modeled by a set of generic setter and getter routines. These generic routines are as follows:

```
string getAttribute(string name)
  raises (  DrmCommunicationException,
    AuthorizationException,
    NoActiveSessionException,
    OutOfMemoryException,
    InternalException,
    UnsupportedAttributeException);
};
```

This method SHALL return the string value of the specified attribute. The language binding specification SHOULD consistently specify the string representation for non-string data types. Valid input values are the strings returned by the *getAttributeNames()* operation. An invalid attribute name leads to an *UnsupportedAttributeException*.

```
stringlist getVectorAttribute(string name)
  raises (  DrmCommunicationException,
    AuthorizationException,
    NoActiveSessionException,
    OutOfMemoryException,
    InternalException,
    UnsupportedAttributeException);
};
```

This method SHALL return the list of string values of the specified vector attribute. A vector attribute is one which is prefixed with “v_” in the table in section **Fehler! Verweisquelle konnte nicht gefunden werden.**. The language binding specification SHOULD consistently specify the string representation for non-string vector elements. Valid input values are the strings returned by the

getAttributeNames() operation. An invalid attribute name leads to an *UnsupportedAttributeException*.

```
void setAttribute(string name, string value)
  raises (  DrmCommunicationException,
    UnsupportedAttributeException,
    InvalidAttributeValueException,
    AuthorizationException,
    NoActiveSessionException,
    OutOfMemoryException,
    InternalException);
};
```

This method SHALL change the value of the specified attribute to the given value. Valid input values for the *name* parameter are the strings returned by the *getAttributeNames()* operation. An invalid attribute name leads to an *UnsupportedAttributeException*. An invalid value for a particular attribute leads to an *InvalidAttributeValueException*. The language binding specification SHOULD consistently specify the string representation for non-string data types.

```
void setVectorAttribute(string name, StringList value)
  raises (  DrmCommunicationException,
    UnsupportedAttributeException,
    InvalidAttributeValueException,
    AuthorizationException,
    NoActiveSessionException,
    OutOfMemoryException,
    InternalException);
};
```

This method SHALL replace the list of values of the specified vector attribute to the given list of values. A vector attribute is one which is prefixed with “v_” in the table in section **Fehler! Verweisquelle konnte nicht gefunden werden.**. Valid input values for the *name* parameter are the strings returned by the *getAttributeNames()* operation. An invalid attribute name leads to an *UnsupportedAttributeException*. An invalid value for a particular attribute leads to an *InvalidAttributeValueException*. The language binding specification SHOULD consistently specify the string representation for non-string vector elements.

If a language binding uses this generic getter / setter approach, then it MUST enforce the usage of the attribute names specification from section **Fehler! Verweisquelle konnte nicht gefunden werden.** for all implementations, and all attributes listed in section **Fehler! Verweisquelle konnte nicht gefunden werden.** MUST be implemented.

9.2 Accessing implementation-specific attributes

A language binding MUST provide a means for accessing implementation-specific attributes, as the getters and setters for such attributes are not defined by the *JobTemplate* interface. This access method MUST be consistent for all attributes

and SHOULD be clearly described in the language binding specification. Some destination languages MAY enable more than one access mechanism.

Some common approaches are:

Introspection approach

In order to access the getters and setters for implementation-specific attributes, the developer must use the destination language's introspection mechanisms to locate and then call the attributes' getters and setters at run time. In such a case, the list of attribute names given by the *attributeNames* attribute MUST be names that are meaningful to the destination language's introspection mechanism.

This approach makes it possible to write applications which are completely portable across binding implementations, including previously unknown binding implementations assuming that the naming of implementation-specific attributes is consistent and/or predictable. A significant disadvantage to this approach is the complexity of writing fully dynamic, introspection-based application logic.

Dynamic Loader Approach

In languages that support dynamic class loading, access to implementation-specific attributes can be encapsulated in classes dedicated to accessing the job template attributes of a specific binding implementation. After determining the binding implementation in use, an application in such a language could dynamically load a class that is capable of setting the implementation-specific attributes of the job template.

An advantage of this approach is that within the scope of the dynamically loaded class, the job template may be safely cast to the implementation type without creating a run-time dependency on the implementation class. Within the class access to the job template attributes is done directly using the job template implementation's declared getters and setters. A disadvantage is that such a class is needed for each binding implementation to be supported, and each such class is limited to operating only on that specific binding implementation. Another disadvantage is that it creates a compile-time dependency on all supported binding implementations, i.e. **all** supported binding implementations must be available at the time the application is compiled.

Discouraged approaches

The direct casting of a job template to the job template implementation class without the use of dynamic class loading SHOULD NOT be used. Such casting, while enabling direct access to all job template attribute getters and setters, creates a compile-time and run-time dependency on all supported binding implementations, i.e. such an application must be bundled with **all** binding implementations, even if it will only be run on one of them.

The combination of job template attribute getters and setters with generic getters and setters, where either set of accessors provides access to only a subset of the job template implementations attributes, SHOULD NOT be used. A DRMAA binding MUST provide consistent attribute access, with support for all attribute types

(required, optional and implementation-specific) in only one language-specific method.

9.3 Constants

Peter Tröger 3.9.08 10:40

Kommentar: TODO: #2837 – more placeholders. Was favored by most survey participants. Needs research about common placeholders in today's DRM systems.

Peter Tröger 3.9.08 14:09

Kommentar: TODO: #5873 – support for the placeholders in more of the JT attributes. Needs research about DRM support. Some parts might be implementable in the DRMAA library only.

The *JobTemplate* interface defines a set of constants that are used in the context of some of the attributes:

The *HOME_DIRECTORY* constant is a placeholder used to represent the user's home directory when building paths for the *workingDirectory*, *inputPath*, *outputPath*, and *errorPath* attributes.

The *WORKING_DIRECTORY* constant is a placeholder used to represent the current working directory when building paths for the *inputPath*, *outputPath*, and *errorPath* attributes.

The *PARAMETRIC_INDEX* constant is a placeholder used to represent the id of the current parametric job subtask when building paths for the *workingDirectory*, *inputPath*, *outputPath*, and *errorPath* attributes.

9.4 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 attribute value SHOULD NOT relate to binary file management or file staging activities.

9.5 args

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

9.6 jobSubmissionState

Defines the state of the job at submission time. For more information see section 5.2.

9.7 jobEnvironment

This attribute holds the environment variable values for the execution machine. The values SHOULD override the remote environment values if there is a collision. If this is not possible, the behavior is implementation dependent.

9.8 workingDirectory

This attribute specifies the directory where the job is executed. If the attribute is not set, the behavior is implementation dependent. The attribute value MUST be evaluated relative to the execution host's file system. The attribute value MAY contain the *HOME_DIRECTORY* or *PARAMETRIC_INDEX* constant values as placeholders. A *HOME_DIRECTORY* placeholder at the begin denotes the remaining portion of the attribute value as a relative directory path resolved relative to the job users home directory at the execution host. The *PARAMETRIC_INDEX* placeholder MAY be used at any position within the attribute value in the case of parametric job

templates and SHALL be substituted by the underlying DRM system with the parametric jobs' index.
The *workingDirectory* MUST be specified in a syntax that is common at the host where the job is executed. 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, the job MUST enter the state `JobState.FAILED`.

9.9 configurationName

DRMAA facilitates writing DRM-enabled applications even though the deployment properties, in particular the configuration of the DRMS, cannot be known in advance. Through the *configurationName* string attribute, a DRMAA application can specify additional job needs that are to be mapped by the DRMAA implementation or DRM system to DRMS-specific options. It is intended as non-programmatic extension of DRMAA job submission capabilities. The interpretation of the *configurationName* job template string attribute is implementation-specific, meaning that a DRMAA implementation could even map any or all configuration names to nothing.
The order of precedence for DRMS configuration options produced by the *configurationName* string attribute mapping versus those injected by the native DRMS configuration is unspecified.

9.10 nativeOptions

This string attribute allows the DRMAA library user to pass DRMS-specific native options during job submission. In contrast to the usage of predefined configuration sets with the *configurationName* attribute, this attribute allows to pass direct DRMS-specific options. As far as the DRMAA interface specification is concerned, the *nativeOptions* string attribute is an implementation-defined string and is interpreted by each DRMAA library in its specific way.

One MAY use DRM configuration facilities, the *configurationName* attribute or the *nativeOptions* attribute at the same time. In this case, the *nativeOptions* attribute SHOULD ultimately overrule other, even conflicting, configurations. Care SHOULD be exercised to not change the job submission call semantics, pass options that conflict the already set attributes, or violate the DRMAA API in any way. Implementations are free to reject job templates with invalid native specifications.

9.11 email

This attribute holds a list of email addresses that is used to report the job completion and status.

9.12 blockEmail

This *Boolean* parameter decides whether the sending of email is blocked by default or not, regardless of the DRMS setting. If the parameter is *TRUE*, the sending of email SHALL be blocked regardless of the DRMS setting. If the value is *FALSE*, the sending of email SHALL be determined by the DRMS setting.

9.13 startTime

This attribute specifies the earliest time when the job MAY be eligible to be run. Date and time are expressed as RFC822 conformant string.

9.14 jobName

A job name SHALL be comprised of alphanumeric and '_' characters. The DRMAA implementation MAY truncate any client-provided job name to an implementation-defined length that is at least 31 characters.

9.15 inputPath

Specifies the job's standard input as a path to a file. If this property is not explicitly set in the job template, the job is started with an empty input stream, unless the named configuration, native options, or a DRMS setting causes a source for the input stream to be set. If this attribute is set, it specifies the network path for the job's input stream file in the form:

```
[hostname]:file_path
```

If the *transferFiles* job template attribute is supported and has a value where the *FileTransferMode::inputStream* attribute set to *true*, the input file SHOULD be fetched by the underlying DRM system from the specified host, or from the submit host if no hostname was specified.
If the *transferFiles* job template attribute is not supported or its value's *FileTransferMode::inputStream* is set to *false*, then the input file is always expected at the host where the job is executed, irrespective of whether a hostname was specified.
The *PARAMETRIC_INDEX* placeholder can be used at any position for parametric job templates and SHALL be substituted by the underlying DRM system with the parametric job's index.
A *HOME_DIRECTORY* placeholder at the beginning of the attribute value denotes the remaining portion as a relative file specification resolved relative to the job's user's home directory at the host where the file is located.
A *WORKING_DIRECTORY* placeholder at the beginning of the attribute value denotes the remaining portion as a relative file specification resolved relative to the job's working directory at the host where the file is located.
The *inputPath* MUST be specified in a syntax that is common at the host where the file is located.
If set, and the job were successfully submitted, and the file can't be read, the job enters the state, *JobState.FAILED*.

9.16 outputPath

Specifies how to direct the job's standard output to a file. If this attribute is not explicitly set in the job template, the destination of the job's output stream is not defined, unless the named configuration, native options, or a DRMS setting causes a destination for the output stream to be set. If this attribute is set, it specifies the network path of the job's output stream in the form:

`[hostname]:file_path`

If the *transferFiles* job template attribute is supported and its value's *FileTransferMode::outputStream* attribute is set to *true*, the output file SHALL be transferred by the underlying DRM system to the specified host or to the submit host if no hostname is specified.
If the *transferFiles* job template attribute is not supported or its value's *FileTransferMode::outputStream* attribute is set to *false,* the output file SHALL be kept at the host where the job is executed, irrespective of whether a hostname was specified.
All output sent to the job's standard output stream SHALL be appended to that file. If the file does not exist at the time the job is executed, the file SHALL first be created.
The *PARAMETRIC_INDEX* placeholder can be used at any position with parametric job templates and SHALL be substituted by the underlying DRM system with the parametric job's index.
A *HOME_DIRECTORY* placeholder at the beginning denotes the remaining portion as a relative file specification resolved relative to the job users home directory at the host where the file is located.
A *WORKING_DIRECTORY* placeholder at the beginning denotes the remaining portion as a relative file specification resolved relative to the jobs working directory at the host where the file is located.
The *outputPath* MUST be specified in a syntax that is common at the host where the file is located. If set and the job were successfully submitted and the file can't be written before execution the job MUST enter the state *`JobState.FAILED`*.

9.17 errorPath

Specifies how to direct the jobs' standard error to a file.
If not explicitly set in the job template, the destination of the job's error stream is not defined unless the named configuration, native options, or a DRMS setting causes a destination for the error stream to be set. If this attribute is set, it specifies the network path of the jobs error stream file in the form:

`[hostname]:file_path`

If the *transferFiles* job template attribute is supported and it's value's *FileTransferMode::errorStream* attribute is set to *true*, the error file SHALL be transferred by the underlying DRM system to the specified host or to the submit host if no hostname is specified.
If the *transferFiles* job template attribute is not supported or it's value's *FileTransferMode::errorStream is* set to *false*, the error file is always kept at the host where the job is executed irrespective of whether a hostname was specified.
All output sent to the job's standard error stream SHALL be appended to that file. If the file does not exist at the time the job is executed, the file SHALL first be created.
The *PARAMETRIC_INDEX* placeholder can be used at any position for parametric job templates and SHALL be substituted by the underlying DRM system with the parametric jobs' index.

A *HOME_DIRECTORY* placeholder at the beginning denotes the remaining portion as a relative file specification, resolved relative to the job users home directory at the host where the file is located.
A *WORKING_DIRECTORY* placeholder at the beginning denotes the remaining portion as a relative file specification resolved relative to the jobs working directory at the host where the file is located.
The *errorPath* MUST be specified in a syntax that is common at the host where the file is located.
If set and the job were successfully submitted and the file can't be written before execution, the job enters the state `JobState.FAILED`.

9.18 joinFiles

Specifies whether the error stream should be intermixed with the output stream. If not explicitly set in the job template, this attribute defaults to *false*. If this attribute is set to *true*, the underlying DRM system SHALL ignore the value of the *errorPath* attribute and intermix the standard error stream with the standard output stream as specified by the *outputPath*.

9.19 transferFiles

Specifies how to transfer files between hosts.
If this attribute is not explicitly set in the job template, the effect is the same as setting the property to a *FileTransferMode* instance with all members set to `false`.
This attribute works in conjunction with the *inputPath*, *outputPath* and *errorPath* attributes.

This attribute is optional. An implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

9.20 deadlineTime

Specifies a deadline after which the DRMS will abort the job. Date and time are expressed as RFC822 conformant string.

This attribute is optional. An implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

9.21 hardWallclockTimeLimit

This attribute specifies when the job's wall clock time limit has been exceeded. An implementation SHALL terminate a job that has exceeded its wall clock time limit. Suspended time SHALL also be counted towards this limit.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

9.22 softWallClockTimeLimit

This attribute specifies an estimate as to how much wall clock time the job will need to complete. Note that the suspended time is also counted towards this estimate.

This attribute is intended to assist the scheduler. If the time specified is insufficient, the implementation MAY impose a scheduling penalty.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

9.23 hardRunDurationLimit

This attribute specifies how long the job MAY be in a running state before its limit has been exceeded, and therefore is terminated by the DRMS.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

9.24 softRunDurationLimit

This attribute specifies an estimate as to how long the job will need to remain in a running state to complete. This attribute is intended to assist the scheduler. If the time specified is insufficient, the implementation MAY impose a scheduling penalty.

This attribute is optional. In case an implementation MUST throw an *UnsupportedAttributeException* if this attribute is not supported.

9.25 attributeNames

This read-only attribute specifies the list of supported attribute names. This list includes supported DRMAA reserved attribute names (both required and optional) and implementation-specific attribute names. The listed attribute name MUST be of a format that is meaningful to the destination language for use in introspection, if supported, or with the *getAttribute()* and *setAttribute()* methods if introspection is not supported. See section **Fehler! Verweisquelle konnte nicht gefunden werden.** for a given names of the job template attributes.

10 Job interface

The following chapter explains the set of constants, methods and attributes defined in the *Job* interface. Every job in the session is expressed by an own instance of the *Job* interface. It allows to instruct the DRM system for a job status change, and to query the status attributes of the job in the DRM system. Status values read from the Job object SHOULD reflect the current status of the job in the DRM system at the time of the call.

The job control functions allow modifying the status of the single job in the DRM system. They SHALL return once 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. The behavior is implementation-specific.

To avoid thread races in multi-threaded applications, the DRMAA implementation user should explicitly synchronize this call with any other call to the same object. This MAY be already realized by the DRMAA implementation.

10.1 suspend

10.2 resume

10.3 hold

10.4 release

10.5 terminate

10.6 wait

10.7 getState

Peter Tröger 7.7.09 00:00
Kommentar: TODO: #2824 – Clarify status query on reaped jobs

The DRMAA implementation MUST always get the status of the job from the DRM system unless the status has already been determined to be `FAILED` or `DONE` and the status has been successfully cached. It is up to the implementation to determine whether this method is capable of operating on jobs submitted outside of the current DRMAA session.

Returns

The getState() method SHALL return the job status, together with an implementation specific sub state. This is intended to be a more detailed description of the current

DRMAA job state, for example the specific kind of HOLD state (user-triggered, system-triggered). Applications SHOULD NOT expect this information to be available in all cases. Language bindings MUST allow the application to discard this information (e.g. by passing a NULL value), and SHOULD use a generic reference data type (e.g. *void or Object pointer). Implementations of the DRMAA API SHOULD then define a DRMS-specific data structure for the sub-state information.

10.8 getInfo

TBD

11 JobInfo interface

TBD

Peter Tröger 29.9.09 16:58

Kommentar: There was a user demand for figuring out why a job remains in queued state – is it only a lack of available execution resources, or something else ?

12 MonitoringSession interface

The *MonitoringSession* interface in DRMAA supports the monitoring of execution resources in the DRM system. This is distinct from the monitoring of jobs running in the DRM system, which is covered by the *Job* and the *JobInfo* interface.

The *MonitoringSession* interface supports four basic units of monitoring:

- Properties of the DRM system as a whole (e.g. DRM system version number) that are independent from the particular session resp. contact string
- Properties of the DRM system that depend on the current contact string (e.g. list of machines in the currently accessed Sun Grid Engine cell)
- Properties of single machines available with the current contact string (e.g. amount of physical memory in a chosen machine)

As with the *JobInfo* interface, the access to properties is organized through key-value-pairs.

… TBD

13 Annex

14 DRMAAv2 JSDL Profile

TBD

15 Security Considerations

Security issues are not discussed in this document. The scheduling scenario described here assumes that security is handled at the point of job authorization/execution on a particular resource.

16 References

[OMG IDL] Object Management Group. Common Object Request Broker Architecture: Core Specification, Chapter 3, March 2004

[RFC 2119] S. Bradner. RFC 2119 – Key words for use in RFCs to Indicate Requirement Levels, March 1997

[IJGUC08] Peter Tröger, Hrabri Rajic, Andreas Haas, and Piotr Domagalski. Standardized Job Submission and Control in Cluster and Grid Environments. In International Journal of Grid and Utility Computing (IJGUC). 2008. ISSN 1741-847X

[GFD133] Hrabri Rajic, Roger Brobst, Waiman Chan, Fritz Ferstl, Jeff Gardiner, Andreas Haas, Bill Nitzberg, John Tollefsrud, and Peter Tröger. Distributed Resource Management Application API Specification 1.0 (GFD.133). Grid Recommendation. Open Grid Forum, 2008.

17 Contributors

Peter Tröger
peter@troeger.eu

18 Acknowledgements

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've missed) Guillaume Alleon, Ali Anjomshoaa, Ed Baskerville, Harald Böhme, Matthieu Cargnelli, Karl Czajkowski, Piotr Domagalski, Fritz Ferstl, Paul Foley, Nicholas Geib, Becky Gietzel, Alleon Guillaume, 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, Ruben S. Montero, Greg Newby, Steven Newhouse, Michael Primeaux, Greg Quinn, Martin Sarachu, Jennifer Schopf, Enrico Sirola, Chris Smith, Ancor Gonzalez Sosa, Douglas Thain, John Tollefsrud, Jose R. Valverde, and Peter Zhu.

19 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.

20 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.

21 Full Copyright Notice