Journal of Grid Computing manuscript No.
(will be inserted by the editor)

Towards Standardized Job Submission and Control in IaaS Clouds

Peter Tröger · Andre Merzky

Received: date / Accepted: date

Abstract The submission and management of computational jobs in a distributed batch processing system is a traditional part of utility computing environments. End users and developers of domain-specific software abstractions often have to deal with the heterogeneity of such systems. This lead to a number of application programming interface and job description standards in the past, which are implemented and established for cluster and Grid systems.

With the recent rise of Cloud Computing as new utility computing paradigm, the standardized access to batch processing facilities operated on cloud resources becomes an important issue. Furthermore, the design of such a standard has to consider a tradeoff between feature completeness and the achievable level of interoperability. This article discusses this general challenge, and presents some existing standards with traditional cluster and Grid computing background that may be applicable to Cloud environments. We furthermore present the OCCI-DRMAA approach as one way of defining a standardized access to batch processing facilities hosted in a cloud.

1 Introduction

Batch processing is a traditional approach in IT systems. It originated in the batched execution of punch cards on early computer systems, and helped to improve the throughput of computational job execution on very expensive IT resources. Many modern operating system concepts, such as supervisor mode versus user mode, have their foundation in these batch systems.

With the advent of cheap distributed hardware in the 80's, batch processing moved to distributed installations of collaborating computer systems, called *clusters*. This lead to the development of cluster management systems for job scheduling, execution host management and monitoring. Typical examples for such systems are GridEngine, the Portable Batch System (PBS), Torque, LoadLeveler, or HTCondor. A cluster infrastructure maps incoming computational jobs to the available set of distributed resources. The largest user base for massively scaled cluster environments exists in the High Performance Computing (HPC) community, where the scalable execution of massively parallel jobs is the primary goal. For several decades, the HPC community had a primary focus on locally distributed cluster systems, where parallel application instances are managed by a local Distributed Resource Management System (DRMS) scheduler, for a set of nodes. Meanwhile, the federated usage of geographically distributed resources through a single interface became another important part of DRMS operation – this is the main usage mode for *grid computing* environments, which is largely motivated by applications with large scientific computational and storage demands, such as the *Large Hadron Collider* [**?**] project.

Batch processing in itself is not only an HPC concept – IBM mainframe systems, for example, support a wide area of transaction-oriented applications with their Job Control Language (JCL) description syntax and scheduler implementations. Another example are modern data-parallel execution frameworks for the map-

Peter Tröger
Hasso Plattner Institute
University of Potsdam
Tel.: +49 - 331 - 5509 234
E-mail: peter.troeger@hpi.uni-potsdam.de

Andre Merzky
CCT, Louisiana State University
E-mail: andre@merzky.net

reduce programming model, which implicitly organize job distribution and management for executed applications.

For each of these usage scenarios for a batch processing system, a common question is the design of interfaces to the Distributed Resource Management (DRM) system functionality: how can they be developed so as to use as broad a range of infrastructure as possible, without vendor or middleware lock-in, yet with the flexibility and performance that is required. This question assumes greater significance in the distributed computing context, such as grids and clouds, where resource ownership, software providers, run-time environment and users are more decoupled than for most other modes of computing.

One design perspective is the end user perspective, were humans (or their shell scripts) interact with the DRM system. Typical solutions motivated by that perspective are command-line tools and graphical user interfaces provided by the particular DRM framework. An alternative design perspective focuses more on programmatic access to the DRM system's functionality – whenever the batch job submission is used by higher level software (e.g. meta-scheduling systems or domain-specific user interface implementations), that software controls the DRM system through some Application Programming Interface (API).

Whether the DRM system is controlled through user tools or through a product API, it must be possible to describe job requirements in some format. This relates to the fact that batch processing is inherently tied to non-interactive jobs, where the request for execution is formulated as a set of job parameters. This includes properties such as the executable name and location, and hardware and operating system requirements. Such *job description* is handed over to the DRM system for interpretation, scheduling and finally execution.

As long as cluster systems were mainly used as organization-specific local resource, it was acceptable to realize job requirement descriptions and programmatic product access through vendor-specific solutions. With the increasing relevance of Grid Computing and it's federation aspect, it became more and more important to have more unified interfaces to resource management systems, as users were exposed to a multitude of system flavors within a single Grid infrastructure. This lead to a number of standardization efforts, mainly in the Open Grid Forum (OGF) standardization body. Different job description formats and interface approaches gained wide-spread acceptance, especially in the academic HPC community. It is now widely and independently acknowledged that a uniform, simple and stable access layer is necessary (but not sufficient) to improve end user experience on distributed computing infrastructures.

With the established Grid Computing paradigm and the according standards being in place, a new utility computing paradigm arose around 2000 - *cloud computing*. We refer to related articles for a discussion about the commonalities between Grid and Cloud Computing [1], but want to emphazise the fact that Cloud Computing is an industry-driven business model, where a provider offers one or more of the following service classes:

Infrastructure as a Service (IaaS) - The service provider offers virtualized compute or storage resources as remotely accessible asset. Well-known examples for providers are Amazon EC2, Microsoft Azure, Rackspace Cloud, or Google Compute Engine. The offers often include auxilliary services such as virtual network switches or software bundles

Platform as a Service (PaaS) - The service provider offers an execution platform for software written by the customer. The platform provides automated application scalability and availability, as long as the customer software follows a particular programming model. Well-known examples for such offerings are the Google AppEngine or Heroku.

Software as a Service (SaaS) - The service provider operates remotely usable software that has tenant capabilities. Well-known examples are Microsoft Office 365, Salesforce or the Amazon Web Services.

Even though the majority of sources declares these service classes as layered concepts, providers can operated and offer them independently: one example is the Google AppEngine offering, which does not fully rely on the IaaS offering by the same company.

2 Motivation

With the recent acceptance of the Cloud Computing paradigm as billing and management approach for federated resource usage, it became also interesting for the HPC community. Different data centers, originally designed with a *closed word assumption* in mind, already opened their resources for external usage through Grid Computing initiatives. These stakeholders are now about to introduce Cloud offerings for their resources, in order to maximize utilization and gain some revenue from the provisioning. A second relevant trend is the rise of *Private Cloud* solutions, were organizations operate their own cloud infrastructure for the purpose of easier management and internal accounting. Both trends lead to the fact that classical job submission is

also about to become a relevant topic in cloud computing infrastructures. This mainly relates to the class of IaaS offerings, were users book virtual execution resources to submit jobs to them. On the other hand, batch processng can also be realized as SaaS offering, were the cloud provider operates the cluster system as remotely usable service. Even in this case, it is not clear how the standardized access could be realized.

Within this article, we want to open the disucssion of how standardized programming interfaces and job description formats can be moved from the world of cluster and Grid systems to IaaS-based batch processing offerings. This would allow job management applications to interact with more than one provider, which gives the chance for cost optimization or improved provider failure management. It also gives providers a better chance to attract new customers that demand the support for standardized access to the resources.

2.1 Role model

Due to the different nature of DRM implementations, we first need define a stakeholder model to be applied in this article.

Distributed Resource Management System (DRMS) : Any system that supports the concept of distributing computational tasks on execution resources by 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, grid systems, or cloud offerings with a *job* concept.

Implementation : Realizes some standardized job description format or a standardized programming interface for a particular DRM system.

Application : A software artifact that utilized an *implementation* for gaining access to one or multiple DRM systems in a standardized way.

Submission Host : A resource in the DRM system that runs the *application*. In traditional local cluster environments, a *submission host* may also be able to act as *execution host*.

Execution Host : A resource in the DRM system that can run a submitted *job*.

Job : Computational activity submitted by the *application* with the help of an *implementation*. A *job* is expected to run as one or many operating system processes on one or many *execution hosts*.

In the following Section 3, we first discuss our hypothesis about the nature of such standardization attempt in batch processing DRM systems. In Section 4, we then explain some of the existing standards and job descriptions formats available for batch processing systems. In the next step, we discuss in Section **??** one possible approach for bringing standardized job submission to Cloud environments.

3 The Feature / Interoperability Dillemma

As part of the discussion of standards development, we introduce a hypothesis regarding any kind of standardization in this application field:

> "The standardization of description formats and interfaces for distributed resource management systems is the permanent mandatory choice between the level of application portability / system interoperability and the level of feature coverage on all detail levels."

The choice between portability and interoperability as correct description term relates to the way how the DRM system interfaces are standardized [2].

Portability implies a vertical system layering, where applications can use an abstraction layer (typically a library) in their location execution to access the DRM system. When such an application can be moved to a different DRM system environment without significant code changes, it becomes portable. Application portability is the design goal for standards targeting local cluster or private cloud environments.

Interoperability, on the other hand, describes the possibility for horizontal interaction of software entities in a distributed system. If the standardized interface or job description format implementation of the DRM system is exposed to remote clients, such as with web service technology, it is described as *interoperable* interface. In a cloud computing environment, system interoperability relates to the kind of service interfaces offered by the cloud provider.

For the remaining text, we stick with the term portability due to the focus on given cluster and Grid solutions.

The implemented level of portability provided by a standard can be further sub-divided into approaches that utilize *introspection* for portability, or the ones that do not rely on such a mechanism. Introspection is a well-known concept in programming language theory and describes the ability of a running program to examine types and values of object properties at run-time. In the context of standardized DRM system interfaces, we refer to the following definition:

> "Introspection is the ability to determine job submission interface properties or job description document properties at runtime."

This allows a portable application to programmatically detect the support for some system-specific features or properties and use them. Applications relying on such a standardized interface or document format become aware of the differing nature of target systems, which lowers their level of portability to some extend. We therefore treat standards with introspection as slightly "less portable" than approaches that have the exactly same feature and property set on all platforms.

While the given hypothesis sounds straight-forward in general terms, it has heavy implications on the standardization of job description formats and programming interfaces in detail. Our experience with more than 8 years of standardization work shows that even though most people agree to this hypothesis in general, they tend to forget about it when it comes to detailed design decisions. Given this hypothesis, any attempt for standardization of job submission and control in IaaS cloud environments must particulary also make a choice with respect to this dillemma.

4 Existing Standards for Cluster and Grid Environments

The following section presents a chosen set of standards in the field of job control and monitoring in DRM systems, which can serve as starting point for a cloud batch processing standardization attempt. We focus solely on the OGF standardization body in this summary, since all relevant activities in the area of cluster and grid systems happen in this organization. Figure 1 shows how the different activities map into the design space given be the hypothesis discussed in Section 3. We put special focus on the Distributed Resource Management Application API (DRMAA) specification, in order to give the reader all relevant information for the technical discussion in Section **??**.

4.1 GLUE

Beside the role model for this article defined in Section 2.1, there are other attempts for such a generalization of the stake holders in a DRM systems. One prominent example is the GLUE specification, which defines a conceptual model for

> grid entities as UML class diagrams. GLUE covers not only the stakeholders defined for this article, but also compute and storage facilities on a fine-grained detail level. A DRM system is denoted as *Manager* entity in the GLUE context. The execution host concept can be mapped to the idea of an *ExecutionEnvironment* and the *ComputingManager* in GLUE. A job is described as *ComputingActivity*.
> GLUE in itself is not intended a foundation for technical implementations. It provides a consistent terminology model for varying kinds of batch processing environment, which as the advantage of using precise terminology in the mapping of standards to other environments.

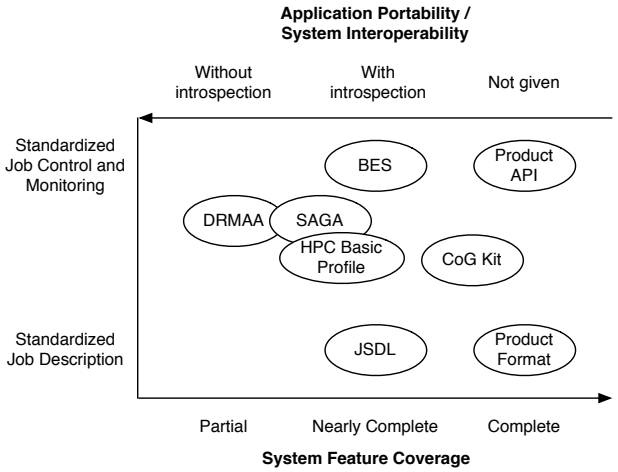

Fig. 1 Classification of Job Submission and Control Standards for Cluster and Grid Computing

4.2 JSDL

The Job Submission Definition Language (JSDL) specification from OGF defines a data scheme for job attributes, their inter-relationship and their value range. Job descriptions based on this schema can be re-used in different DRM systems that support the specification. JSDL therefore provides portability for the job description itself, but does explicitely not consider the interfaces for job management and monitoring. Since re-use of job descriptions is one of the primary design goals in JSDL, several description properties are limited. One example is the explicitely not supported notion of maximum job run-time, since this is an environment-specific property that cannot be reused.
JSDL is a popular and widely accepted format in large-scale DRM systems. It also helps to implement meta scheduling facilities in Grid environments, where job descriptions are forwarded by central management entitity to connected cluster resources. JSDL forms one of the cornerstones for HTTP-based job submission and con-

trol protocols, such as Web Service Resource Framework (WSRF) or Basic Execution Service (BES)
The specification clarifies how an *extension schema* formulated in XML Schema can be used to create JSDL XML documents that relate to product-specific features and properties. This fulfils the goal of best-possible feature coverage in a standardized document format, while still maintaining some level of job description portability. The implementation of introspection capabilities comes from the usage of XML as description format, which has schema extensibility has inherit concept [3].

4.3 BES

The BES specification [4] standardized the remote job submission and control through HTTP-based interaction with the DRM system. Jobs are described through JSDL documents. The standard is defining only a minimal set of mandatory operations and job states to be supported by the interface. Beside that, a well-defined extensibility concept allows to add product-specific states and job properties to the interface functionality. The BES specification distinguishes between different Web Service port types (see also [5]) for factory operations, activities and management operations. The standard provides a basic state model for activities, which explicitely is intended for being extended by implementations. The idea here is to support "composable specializations" based on a common meta model that can support a variety of activities. Examples for from the BES specification are data staging and job suspend operations.
JSDL and BES were combined into a single set of DRM system abstractions by the definition of *profiles*, such as the *High Performance Computing (HPC) Basic Profile* [**?**]. The profile definition tries to increase the level of interoperability by explicitely restriction some freedom for the implementors, f.e. with respect to the job state model. However, the specification still allows the addition of arbitrary extensions to the JSDL job descriptions, while giving the implementation the choice to treat this as an error. Additional efforts in interoperability workshops and tests made BES meanwhile a solid foundation for federated Grid environments, as for example with the Unicore middleware [6].

4.4 SAGA

The Simple API for Grid Applications (SAGA) specification [7] started as continuation of the Grid Application Toolkit (GAT) library, a project-specific programming interface for the GridLab middleware [**?**]. SAGA is an acronym for "Simple API for Grid Applications", where "simple" is to be understood in the measing of ease of use. Users are intended to be decoupled from the complexities arising in the use of a large-scale distributed system.
The API is structured into various packages for different purposes, such as logical and physical file management, system management, job management job monitoring, distributed communication and advanced reservation. Those packages have limited dependencies amongst each other - not all SAGA implementations implement all packages. All API packages share certain properties: how are synchronous methods expressed, how are notifications realized, how are security tokens expressed, what types of exceptions are defined, etc. Those properties are specified in the SAGA-Core, the API's look and feel.
That design of the SAGA API allows to specify additional API packages adhering to the same look-and-feel. In fact, several such API packages have already been defined (e.g., Service Discovery, Messaging etc.), and are standardized as well, or are in the process of being standardized. The interface relies on a consistently asynchronous call model and is described with an IDL-alike syntax.
The SAGA standardization effort is closely synchronized with other specification and community efforts, within and outside of OGF. In particular, OGF groups ensure that SAGA semantics map well to lover level specifications, such as DRMAA, JSDL, and BES.
The language independent SAGA API specification has been mapped to multiple programming languages, in particular to C++, Java and Python. Multiple implementations exists, the most notable ones are SAGA-C++, JSAGA and JavaSAGA. SAGA-C++ is, as the name suggests, a C++ implementation of the SAGA API, maintained by LSU and Rutgers University, and a growing international community. The SAGA-C++ development was in close lockstep with the API specification efforts, and is considered to be complete at this point.

SAGA-Python is a pure Python implementation from Rutgers University, which tries to address several shortcomings of the SAGA-C++ implementation – it focuses in particular on ease of deployment, small footprint, and code maintenance, and attempts to gather a larger developer community.
JSAGA (from IN2P3 in France) and JavaSAGA (from the Vrije Universiteit, Amsterdam) are API compatible Java implementations (they use the same set of abstract class definitions). JSAGA caters to an active, but small user community in France, and supports a relatively large set of adaptors for the job and file API packages. JavaSAGA is mostly a academic research vehicle, which bases its middleware bindings mostly on JavaGAT (its predecessor), and sees some uptake in the Netherlands and the German D-Grid project.
SAGA-C++, JSAGA and JavaSAGA all provide python bindings - the Java implementations realize those via Jython, the C++-implementation via Boost-Python. The Python bindings are thus implemented as a wrappers around the C++ and Java implementations, and are thus able to utilize their complete set of middleware adaptors. The three Python bindings (including the python-native SAGA-Python) are at the moment being unified, and have already been shown to be interoperable.
Interestingly, all SAGA implementations discussed above are adaptor based: a relatively small library provides the SAGA API, and a set of adaptors translate the SAGA API calls into the respective middleware operations. It is those adaptors which encapsulate most of the complexity which was formerly present in the applications layer. While SAGA adaptors are relatively easy to implement (at least as a prototype), they require significant maintenance effort to keep up with the middleware intricacies and evolution.
While SAGA is foremost an API, the SAGA distributions support end users in a variety of ways. In particular, the SAGA distributions also include command line tools implemented via the SAGA API, and higher level libraries for common distributed programming patterns, also basing on the SAGA API. Further, the SAGA distributions provides comprehensive support to compile, link and run SAGA applications (configure scripts, make support, runtime wrappers, developer tutorials , etc).
Command line tools are, in our experience, amongst the first components of any distributed middleware to be exploited by end users. SAGA-C++ provides a set of command line tools which basically cover the complete semantic set of SAGA API calls, such as job submission and management, file management, or replica management.
Several SAGA based projects are actively developing and using higher level programming abstractions, such as pilotjobs, bigjobs, mapreduce, or workflows. Such components are routinely installed and used by a number of user communities, and represent significant added value, although they are not part of the SAGA core code base.

4.5 DRMAA

The DRMAA specification was designed since the very beginning of Grid Computing to define a fundamental set of operations for programmatic access to common capabilities of DRM systems. DRMAA is focused on the maximum possible level of portability, without disqualify the majority of DRM systems or the majority of applications. This comparatively harsh restrictions leads to several features intentionally left out, since their semantic either differs between different DRM systems or because they are not supported by some of the systems. It also marks the primary distinguishing factor between DRMAA und SAGA. While SAGA aims at a high-level, feature complete abstraction of DRM system functionality, DRMAA aims at the maximum level of portability for applications relying on it. This property makes DRMAA a natural candidate for the implementation of SAGAs job-related functionalities, as shown in Figure 2.
Our practical experience in the DRMAA standardization activity shows that the explicit rejection of API functionality or job information properties is mainly a problem for the implementor side, while the majority of end users does not demand a large set of functionality anyway. Since standardization is also driven by business demands of the participating organizations, it is understandable that vendors want to push their favourite features into the specification.
DRMAA is intended to be adoptable to multiple programming languages. Similar to other specifications, such as the W3C Document Object Model (DOM) specification or SAGA, it relies on the description with a language-agnostic inter-

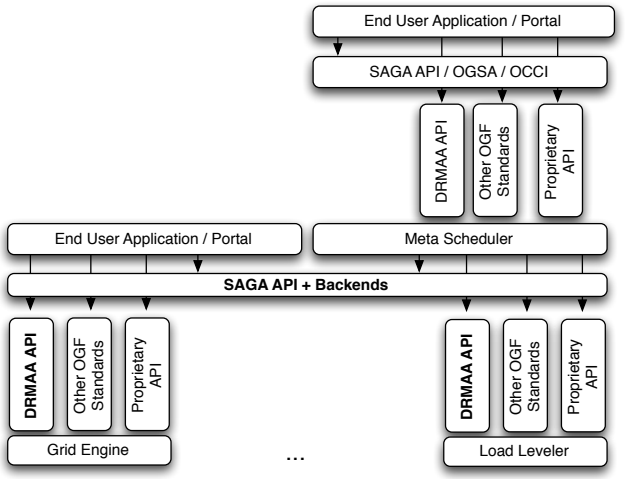

Fig. 2 Relation of DRMAA, SAGA, and other OGF specifications

face description language, in this case CORBA Interface Definition Language (IDL). Based on a root specification in IDL, language bindings can map the syntactical constructs to a particular programming language. This leaves the definition of offered functionality, their grouping and the definition of possible error conditions to a single document.

DRMAA does not consider any security aspect of DRM systems, since this would demand a choice for platform- or middleware-specific security concept (e.g. Unix UID, X.509, Kerberos). Such a choice would be in contrast to the overall goal of platform independence, portability and simplicity. For this reason, DRMAA relies on the security context provided with the user running the application. While this seems to be a strong restriction on first look, it acutally serves as crucial precondition for applying DRMAA in environments where authentication and authorization is anyway handled by lower layers.

The first version of the specification reached the final status of a *grid recommendation* in 2008, with a number of implementations deployed with DRM systems such as GridEngine, HTCondor, and GridWay. Language bindings and their implementation exist for C, Python, Perl, Java, Ruby, TCL and C#.

Due to ongoing development in DRM system technology, a new version of the DRMAA specification was published in 2012 [8]. It considers more functionality now common to different DRM systems and considers also the mapping of DRMAA to a remote usage scenario. The following text refers only to this version of the specification.

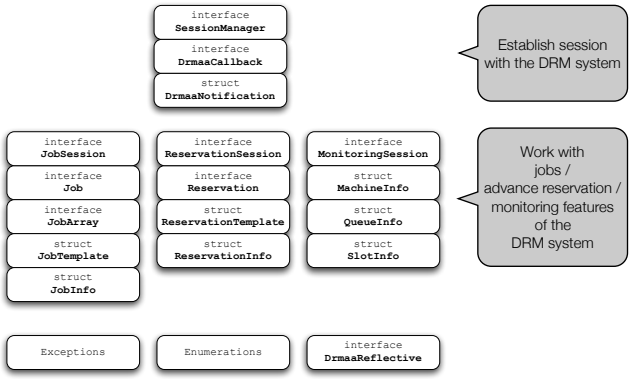

Fig. 3 Functional blocks in the DRMAA specification

DRMAA distinguishes three major classes of functional blocks for job management, advance reservation management and DRM system monitoring (see Figure 3).

The specification relies on a session concept to support the persistency of job and advance reservation information in multiple runs of short-lived applications. Typical examples are a job submission portal or command-line tool. The session concept also allows an implementation to perform attach / detach activities with the DRM system at dedicated points in the application control flow.

The *SessionManager* interface is the main interface of a DRMAA implementation for establishing communication with the 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. The three-fold session concept maps closely to the SAGA API semantics, which ensures that SAGA implementations can easily implement their job-related functionality based on a DRMAA implementation.

The specification mandates the DRM system itself to persist the neccessary state information until the session is explicitely reaped. If state saving is not possible in the particular DRM system, then the state data must be persisted in the implementation of the specification (see also Section 2.1).

A DRMAA job represents a single computational activity that is executed by the DRM system. There are three relevant method sets for working with jobs: The *JobSession* interface represents

all control and monitoring functions available for jobs. The *Job* interface represents the common control functionality for one existing job. Sets of jobs resulting from a bulk submission are controllable as a whole by the *JobArray* interface. A *ReservationSession* instance acts as container for advance reservations in the DRM system, were each of them is represented by a *Reservation* instance. The *MonitoringSession* interface provides a set of stateless methods for fetching information about the DRM system and the DRMAA implementation itself.
In relation to the dillemma discussion in Section 3, DRMAA makes a set of explicit restrictions in the supported functionality. The majority of functionality must be supported by any implementation, in order to ensure the maximum portability of applications relying on these interfaces. Even then, it was neccessary to have a notion of optional functionality in the DRMAA interface, mainly reasoned by cases where only one, but crucial, DRM platform was not supporting a particular feature.
DRMAA solves this by making an explicit distinuishing between *optional* and *implementation-specific* features. Optional features have a clear semantic described by the DRMAA standard. They are part of the interface structures. Any application can programmatically check for the support of a particular optional feature before using it. The concept is similar to the idea of support levels in the POSIX specification series.
The second class of non-mandatory functionality are implementation-specific attributes in data structures. DRMAA only defines how an implementation can add such extensions withouth breaking un-aware applications. This is a similar extensibility approach as with JSDL, BES or any other standard relying on the XML extensibility support.

5 Standards for IaaS Cloud Environments

With the given support for standardized job submission and control in cluster and grid systems, it is also relevant to understand the state of standardization in IaaS cloud environments. With the given significance of cloud computing in industry and research projects, a lot of standardization activities are either announced or started.
One group of cloud standards targets the lifecycle management of service offerings such as virtual machines (IaaS), applications (PaaS) or storage blocks (SaaS). Typical functionality being standardized here is the initialization / deployment of new instances, management operations, backup operations, security and bootstrapping support.
Another group of standards tries to unify monitoring functionality. These standards focus on the proper surveillance and auditing of service-level agreement properties, such as data latency, througput or availability.
The third group of specifications deals with data formats, such as for virtual machines, deployed PaaS applications or application data to be used in a SaaS functionality. Standards in this group typically relate to lifecycle management specifications from the first group. This is similar effect as with the JSDL vs. BES / SAGA / DRMAA development in the Grid community - the standardization of description and data formats is separated from the core API standardization.
With the focus on standardized batch processing interfaces in the cloud, the first group of specifications is most interesting here. Prominent standards from this class the area of IaaS are the Open Cloud Computing Interface (OCCI) specification from OGF, the Open Virtualization Format (OVF) by the Distributed Management Task Force (DMTF) [**?**], and the Cloud Data Management Interface (CDMI) specification from the Storage Networking Industry Association (SNIA) standardization body. All three standards have shown to be usable in conjunction to implement a standards-based cloud service offering [**?**]. The OVF specification defines how a deployment package for IaaS environments can be formulated in a portable way, so that virtual machine configurations are usable with different cloud offerings. CDMI offers the means to programmatically manage cloud storage resources. OCCI provides the standardized remote API to manage the IaaS resources instances remotely hosted by the cloud provider. It therefore provides the interface to trigger operations on and within the instantiated cloud resources.

5.1 OCCI

The OCCI specification is defined as a set of complimentary documents with a core specification [9], rendering specifications and extension specifications. The OCCI core specification defines several foundational concepts being used both by OCCI extensions and OCCI renderings:

- A *resource* is an entity that is exposed through an OCCI-compliant implementation. Extension specification can sub-class the resource type to express specific concepts to be managed, such as virtual machines or storage resources.
- A *link* entity associates resource entities with each other. One example is the relation between virtual machines and their storage space,
- Actions and capabilities for resource instances are defined by a *kind* definition. Instances of this type can be linked to resource and link instances, in order to support different activities on the entities.

Extension documents can specify resource types, their actions and attributes for a particular application domain of OCCI. Rendering documents specify a wire protocol that represents the OCCI core and extension concepts, f.e. by HTTP verbs and addressable ReST resources [10].
While the language-agnostic definition of core concepts is similar to the SAGA and DRMAA approach, the extensibility concept is similar to the BES architecture. An OCCI implementation is expected to provide cloud services to a remote client entity as abstraction for the resource management framework on the provider side. Figure **??** shows an example interaction with OCCI based on the HTTP rendering for the OCCI infrastructure extension [11].

6 Standardized Job Submission in the Cloud

From the given set of existing cluster and grid standards, BES has a great potential for being used as interoperable interface to batch processing facilities in the cloud. Recent experiments in the *European Middleware Initiative (EMI)* project show how a BES implementation can be deployed inside a virtual machine instance running at the cloud provider side. The example of BES in the cloud shows an important architectural aspect of batch processing in an IaaS cloud, which we call the *intra-VM* vs. the *inter-VM* approach.
With the *intra-VM* approach, both the DRM system and the standardized interface implementation are deployed as part of virtual machines running on cloud resources. This approach makes use of the inherent nature of IaaS offerings, where virtualized resources are used as simple remotely hosted replacement for local execution resources. In theory, it supports the re-use of existing software stacks and standards by just deploying existing middleware stacks in the cloud. This kind of approach is not new - one example from the Grid computing field is the Condor GlideIn functionality for extending a Condor pool with resources from a remote Globus installation [12]. Practical experience in the Grid community showed that this approach, even though it seems to habe a low entrance barrier, suffers from the dynamic nature of remote resources. Virtual machines hosted by an IaaS provider follow the dynamics defined by the operating provider, while most batch processing systems assume dedicated resource ownership for the system where they are installed. This leads to issue with the bootstrapping of cluster installations on cloud resources, the identification of machines, the handling of data traffic between multiple virtual machines. Furthermore, there is now a competition between the virtual machine resource management by the cloud provider and the resource management implemented by the deployed DRM system.
An alternative approach is the *inter-VM* concept, where the standardized batch processing functionality becomes part of the cloud offering. The job submission and control interface is implemented by the cloud provider, either by the help of the (anyway existing) cloud resource management framework or by the integration of some DRM system in the cloud infrastructure. Such an offering can be interpreted as SaaS offering, since the provider enables the remote utilization of a new kind of software functionality.
With the given distinuishing between intra-VM and inter-VM solutions, we propose the extension of an existing IaaS cloud standard to support the notion of job batch processing as cloud offering. The idea here is to extend a given cloud interface standard, in this case OCCI, with the API concepts known from cluster and grid systems. Our choice here is the DRMAA specification, since it offers the best-possible interoperability in the feature coverage / interoperability tradeoff decision discussed in Section 3. Such an approach allows cloud providers to extend their offerings with a batch processing facility in an *inter-VM* approach.

6.1 The OCCI-DRMAA approach

The OCCI-DRMAA approach combines the extensibility capability of OCCI with the strict fea-

ture set definition from the DRMAA specification. It therefore serves both as DRMAA language binding specification and as OCCI extension specification for the domain of DRM systems that are covered by the DRMAA specification. The behavioral semantic of DRMAA-OCCI actions is taken from the DRMAA specification, while all syntactical aspects of the access protocol are defined by a chosen OCCI rendering.
DRMAA interfaces represent activities on instantiatable entities. They are mostly modeled as OCCI resources:

- A *drmaa2* resource represents the container for all OCCI-DRMAA resources and the according functionalities.
- A *jobsession* resource acts as container for *job* resources and *jobarray* resources.
- A *reservationsession* resource acts as container for *reservation* resources.
- A *monitoringsession* resource acts as representation of information about the DRM system on provider side.
- A OCCI-DRMAA *job* resource represents one job in the underlying DRM system on provider side. Similarly, the *jobarray* resource represents a cluster of jobs.
- A *reservation* resource represents a successfully created advance reservation in the DRM system.

DRMAA IDL interface attributes map to OCCI attributes. The *readonly* modifier for DRMAA attributes translates to the immutability property. The concept of optional or possibly *UNSET* attributes in DRMAA is mapped to a OCCI attribute multiplicity of $0 \ldots 1$. Id-based or name-based referencing of instances (e.g. of a DRMAA session) is replaced by URI-based referencing.
Original DRMAA methods that return data structure instances are mapped to OCCI or HTTP verb actions that return the location of a new resource instance. By using an appropriate content type in a GET request for such an instance, the client can be even enabled to retrieve a serialized version of the struct instance. Most of the enumeration members from the DRMAA specification, such as for job states, operating system types or quota types, are mapped directly to JSON strings.
Figure 4 shows an interaction example for the retrieval of all existing job session through OCCI-DRMAA.
Figure 5 shows how to establish a job session and submit a job through a OCCI-DRMAA API. In

```
> GET /drmaa2/jobsession/ HTTP/1.1
> [...]

< HTTP/1.1 200 OK
< Content-type: text/uri-list
< [...]
< http://example.com/drmaa2/jobsession/17
```

Fig. 4 Example: Retrieving all existing job sessions

the first step, a job session is created by a POST request. In the second step, a job template resource is created that contains all the relevant job information. With third step, the job execution is implicitely triggered by creating a new *job* resource that refers to the created session and template. The example shows how the abstract concepts from a specification such as DRMAA can be seamlessly mapped to a distributed cloud environment supporting the OCCI specification.
DRMAA interface methods that trigger state changes in the DRM system map to OCCI actions on OCCI resources. DRMAA functionalities that lead to the creation of instances represented by OCCI resources are available as OCCI resource creation activities. DRMAA interface methods that return named instances are not translated to OCCI actions, since this kind of retrieval is possible by formulating a resource location string explicitely.
DRMAA templates are data structures that express complex information entities as a whole, such as job or advance reservation information. They might be modified by a DRM system after their creation, which makes them additional OCCI resources without actions.
As discussed in Section **??**, the DRMAA session concept models the relationship of *Job* and *JobSession* instances. Similarly, it models the relation between *Reservation* and *ReservationSession* instances. In OCCI-DRMAA, these relationships are represented by OCCI links between the according resource entities - a *joblink* resource represents the connection of a job to it's job session, and a *reservationlink* resource represents the connection of an advance reservation to it's reservation session.
DRMAA also defines a set of exceptions that may be thrown by API activities. In OCCI-DRMAA, the mapping of these exceptions depends on the chosen transport rendering. This demands the specification of exception mappings to a particular rendering method, as for example shown in Table 1.

```
> POST /drmaa2/jobsession/ HTTP/1.1
> [...]
> X-OCCI-Attribute: occi.drmaa2.contact="headnode.testbed.platform.com"
> X-OCCI-Attribute: occi.drmaa2.sessionName="MyTestSession"
> [...]
< HTTP/1.1 201 CREATED
< [...]
< Location: http://example.com/drmaa2/jobsession/session1
< [...]
> POST /drmaa2/jobtemplate/ HTTP/1.1
> [...]
> X-OCCI-Attribute: occi.drmaa2.remoteCommand="/bin/date"
> X-OCCI-Attribute: occi.drmaa2.machineOS="LINUX"
> X-OCCI-Attribute: occi.drmaa2.email=["peter@troeger.eu","tmetsch@platform.com"]
> X-OCCI-Attribute: occi.drmaa2.emailOnTerminated=true
> [...]
< HTTP/1.1 201 CREATED
< [...]
< Location: http://example.com/drmaa2/jobtemplate/template1
< [...]
> POST /drmaa2/job/ HTTP/1.1
> [...]
> X-OCCI-Attribute: occi.drmaa2.session="/drmaa2/jobsession/session1"
> X-OCCI-Attribute: occi.drmaa2.jobTemplate="/drmaa2/jobtemplate/template1"
> [...]
< HTTP/1.1 201 CREATED
< [...]
< Location: http://example.com/drmaa2/job/job43
< [...]
```

Fig. 5 Example: Creating a job session and submitting a job.

DRMAA Exception	HTTP Error Code
DeniedByDrmsException	401 / 403
DrmCommunicationException	500
TryLaterException	503, with retry header
TimeoutException	410
InternalException	500
InvalidArgumentException	400
InvalidSessionException	404
InvalidStateException	409
OutOfResourceException	503, without retry header
UnsupportedAttributeException	400
UnsupportedOperationException	405
ImplementationSpecificException	500

Table 1 Mapping of DRMAA exceptions to HTTP error codes.

As all APIs initially intended for local library implementations, DRMAA supports the notion of blocking status wait calls for both the *Job* and the *JobSession* interface. Our OCCI-DRMAA approach therefore re-models synchronous calls with the concept of a wait handle URI. An example can be seen in Figure 6.

The wait action returns the location of the wait handle, which can be further used for polling GET requests to the server. The server must then return one of these three possible error codes on such request:

- *Still waiting* (HTTP error 404): The blocking wait call is still running, no timeout oc-

```
> GET /drmaa2/job/job43?action=waitstarted HTTP/1.1
> [...]
> X-OCCI-Attribute: occi.drmaa2.timeout="..."
> [...]

< HTTP/1.1 202 ACCEPTED
< [...]
< Location: /drmaa2/job/job43/waithandle1
< [...]

> GET /drmaa2/job/job43/waithandle1 HTTP/1.1
> [...]
< HTTP/1.1 404 NOT FOUND

> GET /drmaa2/job/job43/waithandle1 HTTP/1.1
> [...]
< HTTP/1.1 410 GONE

> GET /drmaa2/job/job43/waithandle1 HTTP/1.1
> [...]
< HTTP/1.1 301 MOVED PERMANENTLY
< [...]
< Location: /drmaa2/job/job43
< [...]
```

Fig. 6 Waiting for job start in OCCI-DRMAA

curred so far. The wait handle location remains valid.
- *Timeout* (HTTP error 410): The blocking call was terminated due to timeout. The wait handle location is now invalid.
- *Success* (HTTP error 301): The blocking call was terminated since the wait condition was fulfilled. The wait handle location is now invalid.

The DRMAA API does not specifically assume the existence of a particular security infrastructure in the DRM system. It is assumed that credentials owned by the application using the API are in effect for the DRMAA implementation too, so that it acts as stakeholder for the application. This relays the responsibility of authentication to the OCCI rendering specification that is used to realize the wire protocol of an implementation.

7 Summary

Acknowledgements

Acronyms

API Application Programming Interface.

BES Basic Execution Service.

CDMI Cloud Data Management Interface.

DMTF Distributed Management Task Force.
DOM Document Object Model.
DRM Distributed Resource Management.
DRMAA Distributed Resource Management Application API.
DRMS Distributed Resource Management System.

GAT Grid Application Toolkit.

HPC High Performance Computing.

IDL Interface Definition Language.

JCL Job Control Language.
JSDL Job Submission Definition Language.

OCCI Open Cloud Computing Interface.
OGF Open Grid Forum.

OVF Open Virtualization Format.

SAGA Simple API for Grid Applications.
SNIA Storage Networking Industry Association.

WSRF Web Service Resource Framework.

References

1. T. Rings, G. Caryer, J. Gallop, J. Grabowski, T. Kovacikova, S. Schulz, I. Stokes-Rees, Journal of Grid Computing: Special Issue on Grid Interoperability **7** (2009). DOI 10.1007/s10723-009-9132-5
2. J. Snell, T. Glover. Portability and interoperability: Similarities and differences explained. http://www.ibm.com/developerworks/webservices/library/ws-port/ (2003)
3. D. Fallside, P. Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation (2004)
4. I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, M. Theimer. OGSA Basic Execution Service v1.0 (GFD-R.108) (2008)
5. J.J. Moreau, A. Ryman, S. Weerawarana, R. Chinnici. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language (2006)
6. M. Riedel, B. Schuller, D. Mallmann, R. Menday, A. Streit, B. Tweddell, M. Memon, A. Memon, B. Demuth, T. Lippert, in *Eleventh International IEEE EDOC Conference Workshop* (IEEE Computer Society, Washington, DC, USA, 2007), pp. 57–60. DOI 10.1109/EDOCW.2007.37
7. T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, C. Smith. A Simple API for Grid Applications (SAGA) Version 1.1 (GFD-R-P.90) (2008)
8. P. Tröger, R. Brobst, D. Gruber, M. Mamonski, D. Templeton. Distributed Resource Management Application API Version 2 (DRMAA). http://www.ogf.org/documents/GFD.194.pdf (2012)
9. R. Nyr en, A. Edmonds, A. Papaspyrou, T. Metsch. Open Cloud Computing Interface - Core. http://www.ogf.org/documents/GFD.183.pdf (2011)
10. T. Metsch, A. Edmonds. Open Cloud Computing Interface - RESTful HTTP Rendering. http://www.ogf.org/documents/GFD.185.pdf (2011)
11. T. Metsch, A. Edmonds. Open Cloud Computing Interface - Infrastructure. http://www.ogf.org/documents/GFD.184.pdf (2011)
12. D. Thain, T. Tannenbaum, M. Livny, *Condor and the Grid* (John Wiley & Sons Inc., 2002), pp. 299–336