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

Towards Standardized Job Submission and Control in Infrastructure 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 further present OCCI-DRMAA as one approach to 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.

Peter Tröger
Hasso Plattner Institute, University of Potsdam
E-mail: peter.troeger@hpi.uni-potsdam.de

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

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 in a distributed environment. A cluster infrastructure maps incoming computational jobs to the available set of distributed resources. Typical examples for such systems are GridEngine, the Portable Batch System (PBS), Torque, LoadLeveler, or HTCondor.

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. Meanwhile, the federated usage of geographically distributed resources through a single interface became another important part of Distributed Resource Management System (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 is not only an HPC concept – IBM mainframe systems, for example, support a wide range of transaction-oriented applications with their Job Control Language (JCL) description syntax and scheduler implementations. Other examples are modern data-parallel execution frameworks for the map-reduce programming model, which automatically organize job distribution and management for 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 owners and users are more decoupled than for most other modes of computing.

One design perspective is the end user perspective, where 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 a *job description* is handed over to the DRM system for interpretation, scheduling and finally execution.

When 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 federations and grid computing, it became increasingly important to provide inform 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 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 respective standards being in place, a new utility computing paradigm arose around 2000 - *cloud computing*. We refer to [18] for a discussion about the commonalities between grid and cloud computing, 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, 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 Marketplace.

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. Several data centers, originally designed with a *closed world assumption* in mind, already opened their resources to grid computing initiatives for external usage [6]. 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, where organizations operate their own cloud infrastructure for the purpose of easier management and internal accounting [11]. Due to both trends, the relevance of classical job submission is increasing for cloud computing infrastructures.

With this article, we want to open a discussion about the application of standardized programming interfaces and job description formats from the cluster and grid systems world to batch-processing-based IaaS offerings. This would allow job management applications to interact with more than one cloud provider, support cost optimization and provide failure management. It also increases the chance to attract new customers which demand standardized resource access.

Due to the different nature of DRM implementations, we first define a role model for the context of this article.

A *Distributed Resource Management System (DRMS)* is any system that implements the execution of computational tasks on distributed resources. 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.

An *implementation* is a software artifact that realizes some standardized job description format or a standardized programming interface for a particular DRM system.

An *application* is a software artifact that utilized an *implementation* for gaining access to one or multiple DRM systems in a standardized way.

The *submission host* is a resource in the DRM system that runs the *application*. In traditional local cluster environments, a *submission host* may also be the *execution host*.

The *execution Host* is a resource in the DRM system that can run a submitted *job*.

A *job* is a computational activity submitted from an *application* to an *implementation*. The *job* is expected to run as one or more operating system processes on one or more *execution hosts*.

The following Section 3 discusses the nature of standardization attempts to batch processing DRM systems. In Section 4 and 5, we then describe some of the existing API standards and job descriptions formats for batch processing systems and IaaS systems. Section 6 discusses one possible approach for bringing standardized job submission to cloud environments.

3 The Feature / Interoperability Dilemma

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 display a tradeoff between the level of application portability and/or system interoperability, and the level of feature completeness."

Whether the particular standardization relates to portability and interoperability depends on how the DRM system interfaces are standardized [19]. *Portability* implies a vertical system layering, where applications can use an abstraction layer (typically a library) in their local execution to access the DRM system. An application is portable if it can be moved to a different DRM system environment without significant code changes.

Interoperability, on the other hand, describes the possibility for horizontal interaction of software entities in a distributed system. Exposing standardized interfaces and using standardized job description formats supports the interoperation of DRM systems – they are thus interoperable. In a cloud computing environment, system interoperability relates to the kind of service interfaces offered by the cloud provider.

For the remaining text, we focus on the portability aspects, as those are more prevalently handled in the DRMS systems discussed here.

The implemented degree 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 many implications on the standardization of job description formats and programming interfaces details. 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 specific design decisions. Given this hypothesis, any standardization of job submission and control in IaaS cloud environments must make a specific choice in respect to this dilemma.

4 Existing Standards for Cluster and Grid Environments

The following section presents a selection of standards related to job control and monitoring in DRM systems,

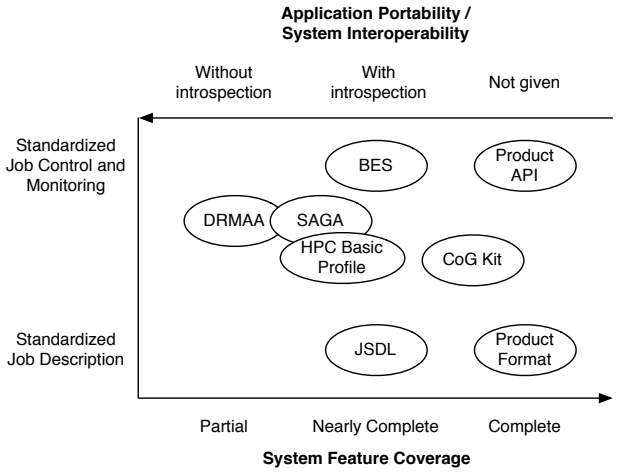

Fig. 1 Classification of Job Submission and Control Standards for Cluster and Grid Computing

which could serve as starting point for a cloud batch processing standardization attempt. We focus solely on the OGF standards at this point, as those cover all relevant activities in the area of cluster and grid systems. Figure 1 shows how the different activities mapped into the design space given be the hypothesis discussed in Section 3.

4.1 GLUE

Beside the role model for this article defined in Section 2, 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.

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 control 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 [7].

4.3 BES

The BES specification [8] 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 [15]) 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* [3]. 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 [17].

4.4 SAGA

The Simple API for Grid Applications (SAGA) specification [10] started as continuation of the Grid Application Toolkit (GAT) library, a project-specific programming interface for the GridLab middleware [2]. 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 lower 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++, SAGA-Python, 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. Basically all SAGA imple-

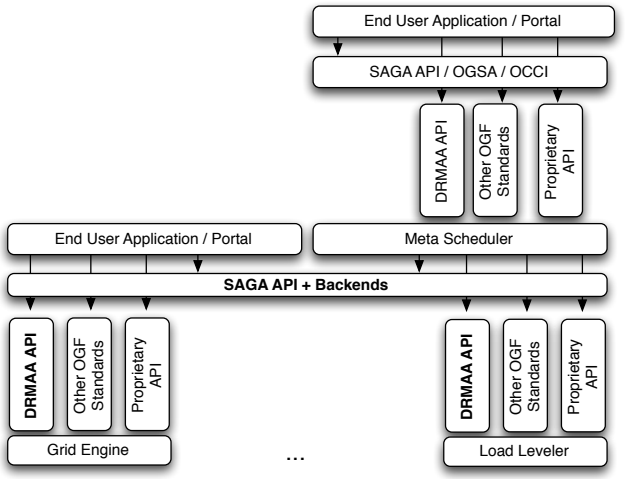

Fig. 2 Relation of DRMAA, SAGA, and other OGF specifications

mentations provide sets of command line tools which cover the important parts of the semantic set of SAGA API calls, such as job submission and management, and file 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 Distributed Resource Management Application API (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 disqualifying the majority of DRM systems or the majority of applications. This comparatively harsh restriction 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 and SAGA. While SAGA aims at a high-level, application oriented 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 SAGA's 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 do 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 adaptable 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 interface 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 [21]. 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.

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

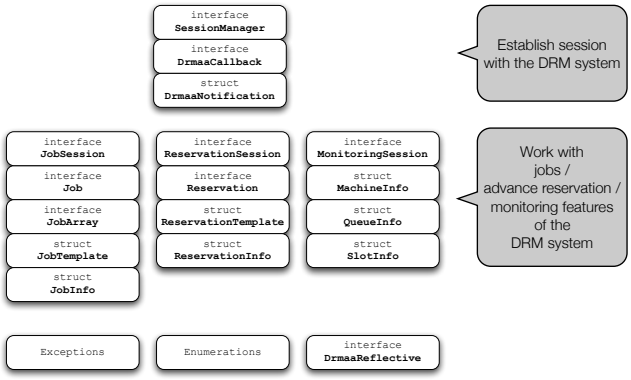

Fig. 3 Functional blocks in the DRMAA specification

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).

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 dilemma 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 without 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

The previous section discussed established standards for job submission and control in cluster and grid systems. This section complements that with the state of standardization in IaaS cloud environments, which is equally important for the scope of this article.

The perceived significance of cloud computing in industry and research projects resulted in numerous standardization activities that are either announced or have started (for an overview, see [1]). One group of cloud standards targets the lifecycle management of service offerings such as virtual machines (IaaS), applications (PaaS) or software blocks (SaaS). Those standards typically cover the deployment and initialization of new instances, instance management operations, backup operations, bootstrapping support and security.

Another group of standards attempts to unify monitoring functionality – those standards focus on surveillance and auditing of service-level agreement properties, such as communication latency, throughput or availability.

The third group of specifications focuses on data formats, such as for virtual machines, deployed PaaS applications or application data to be used in a SaaS context. Standards in this group typically relate to lifecycle management specifications from the first group. This relation is comparable to the standardization of description and data formats which is separated from, but related to, the API standardization – similar to JSDL vs. BES / SAGA / DRMAA specifications in the grid context.

In the context of batch processing interfaces for cloud environments, the first and third group of specifications are both relevant, as they apply to infrastructures which are potentially used as DRMS execution resources. Prominent standards from these classes are the Open Cloud Computing Interface (OCCI) specification from the OGF, the Open Virtualization Format (OVF) from the Distributed Management Task Force (DMTF) [4], and the Cloud Data Management Interface (CDMI) specification from the Storage Networking Industry Association (SNIA) standardization body. All three standards have been shown to be combinable to implement a standards-based cloud service offering [5]. 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 hosted by different cloud providers. 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 [16], 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, i.e. by HTTP verbs and addressable ReST resources [14]. An example interaction is shown in Figure 4, where OCCI is used to instantiate a virtual machine on the cloud provider side.

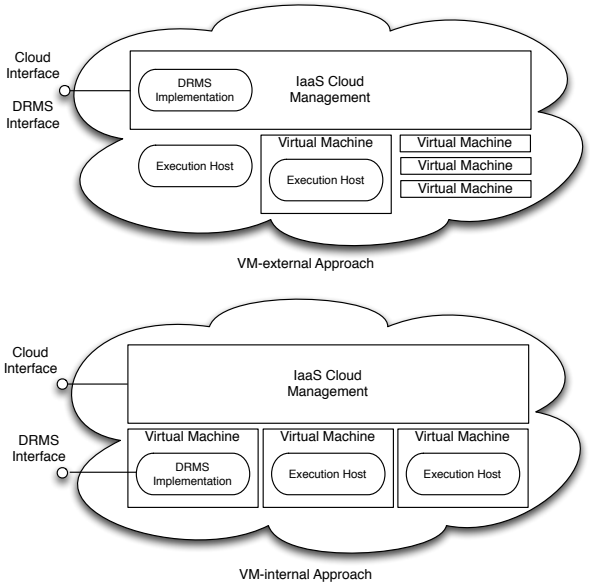

Fig. 5 VM-external vs. VM-internal approach for batch processing in IaaS clouds

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 4 shows an example interaction with OCCI based on the HTTP rendering for the OCCI infrastructure extension [13].

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 [12]. Another initiative is the HPC Ubercloud Experiment [9]. The example of BES in the cloud shows an important architectural aspect of batch processing in an IaaS cloud, which we call the* VM-internal *vs. the* VM-external *approach (see Figure 5).*

With the VM-internal *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*

```
--> POST /compute/ HTTP/1.1
    [...]

    Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure#"; class="kind";
    X-OCCI-Attribute: occi.compute.cores=2
    X-OCCI-Attribute: occi.compute.hostname="foobar"
    [...]
<-- HTTP/1.1 201 OK
    [...]
    Location: http://example.com/vms/foo/vm1
```

Fig. 4 Example: Creating a virtual machine instance

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 [20]. Practical experience in the grid community showed that this approach, even though it seems to have 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 VM-external *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 PaaS offering, since the provider enables the remote utilization of a new kind of software functionality.*

With the given distinuishing between VM-internal and VM-external 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 VM-external *approach.*

6.1 The OCCI-DRMAA approach

The OCCI-DRMAA approach combines the extensibility capability of OCCI with the strict feature set definition given by the the DRMAA specification. It therefore serves both as DRMAA language binding specification and as OCCI extension 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*

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

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

Fig. 6 Example: Retrieving all existing job sessions

mapped to a OCCI attribute multiplicity of 0...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 6 shows an interaction example for the retrieval of all existing job session through OCCI-DRMAA.

Figure 7 shows how to establish a job session and submit a job through a OCCI-DRMAA API. In 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 4.5, 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.

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 8.*

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 occurred 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

```
--> 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. 7 Example: Creating a job session and submitting a job.

```
--> 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. 8 Waiting for job start in OCCI-DRMAA

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

The research field of batch processing systems has a long history that is closely related to the history of distributed systems. Standardization of interfaces and data formats for those environments was always a crucial aspect for ensuring both portability and interoperability for client applications.

We presented a high level overview of different existing standards for batch processing systems in a cluster or grid environment. We introduced the central tradeoff question of standardization in this field, which relates to the choice between maximum feature coverage, and maximum portability and interoperability.

With the given set of specifications, we presented a new approach for re-using existing standards for job submission and control in infrastructure clouds. Our OCCI-DRMAA example represents an approach that combines an existing cloud interoperability specification with a specification originally targeting local DRM systems. With a finalized version of such a specification, both cloud providers and cloud customers are enabled to perform a seamless transition from cluster or grid resources to batch processing offers in the cloud.

References

1. http://cloud-standards.org/.
2. Gabriel Allen, Kelly Davis, Konstantinos Dolkas, Nikolaos Doulamis, Tom Goodale, Thilo Kielmann, Andre Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. *Enabling Applications on the Grid: A GridLab Overview.* International Journal of High Performance Computing Applications, *4:449–466, 2003.*
3. Blair Dillaway, Marty Humphrey, Chris Smith, Marvin Theimer, and Glenn Wasson. *HPC Basic Profile, Version 1.0.* http://www.ogf.org/documents/GFD.114.pdf, *August 2007.*
4. *DMTF Virtualization, Partitioning, and Clustering Working Group. Open Virtualization Format Specification, February 2009.*
5. Andy Edmonds, Thijs Metsch, and Eugene Luster. *An Open, Interoperable Cloud.* http://www.infoq.com/articles/open-interoperable-cloud, *July 2011.*
6. Dietmar Erwin and David Snelling. *UNICORE: A Grid Computing Environment.* Lecture Notes in Computer Science, *2150:825, 2001.*
7. David Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition. W3C Recommendation, October 2004.*
8. I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. *OGSA Basic Execution Service v1.0 (GFD-R.108), November 2008.*
9. Wolfgang Gentzsch and Burak Yenier . *HPC Experiment: Round 1 – Final Report. 2012.* http://www.hpcexperiment.com/.
10. Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. *A Simple API for Grid Applications (SAGA) Version 1.1 (GFD-R-P.90), January 2008.*
11. Craig Lee. *A perspective on scientific cloud computing. In* 19th ACM International Symposium on High Performance Distributed Computing, *pages 451–459, New York, NY, USA, 2010. ACM.*
12. Shahbaz Memon, Eric Yen, Morris Riedel, Mischa Salle, and Oscar Koeroo. *EMI Cloud Strategy.* http://indico.egi.eu/indico//contributionDisplay.py?contribId=210&resId=4&confId=452.
13. Thijs Metsch and Andy Edmonds. *Open Cloud Computing Interface - Infrastructure.* http://www.ogf.org/documents/GFD.184.pdf, *April 2011.*
14. Thijs Metsch and Andy Edmonds. *Open Cloud Computing Interface - RESTful HTTP Rendering.* http://www.ogf.org/documents/GFD.185.pdf, *June 2011.*
15. Jean-Jacques Moreau, Arthur Ryman, Sanjiva Weerawarana, and Roberto Chinnici. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, March 2006.*
16. Ralf Nyren, Andy Edmonds, Alexander Papaspyrou, and Thijs Metsch. *Open Cloud Computing Interface - Core.* http://www.ogf.org/documents/GFD.183.pdf, *April 2011.*
17. Morris Riedel, Bernd Schuller, Daniel Mallmann, Roger Menday, Achim Streit, Bastian Tweddell, M. Memon, A. Memon, Bastian Demuth, and Thomas Lippert. *Web Services Interfaces and Open Standards Integration into the European UNICORE 6 Grid Middleware. In* Eleventh International IEEE EDOC Conference Workshop, *pages 57–60, Washington, DC, USA, 2007. IEEE Computer Society.*
18. Thomas Rings, Geoff Caryer, Julian Gallop, Jens Grabowski, Tatiana Kovacikova, Stephan Schulz, and Ian Stokes-Rees. *Grid and Cloud Computing: Opportunities for Integration with the Next Generation Network.* Journal of Grid Computing: Special Issue on Grid Interoperability, *7, August 2009.*
19. James Snell and Tom Glover. *Portability and interoperability: Similarities and differences explained.* http://www.ibm.com/developerworks/webservices/library/ws-port/, *March 2003.*
20. Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid, *pages 299–336. John Wiley & Sons Inc., December 2002.*
21. Peter Tröger, Roger Brobst, Daniel Gruber, Mariusz Mamonski, and Daniel Templeton. *Distributed Resource Management Application API Version 2 (DRMAA).* http://www.ogf.org/documents/GFD.194.pdf, *January 2012.*