Chapter 2
Open Cloud Computing Interface in Data Management-Related Setups

Andrew Edmonds, Thijs Metsch, and Alexander Papaspyrou

Abstract The Cloud community is a vivid group of people who drive the ideas of Cloud computing into different fields of Information Technology. This demands for standards to ensure interoperability and avoid vendor lock-in. Since such standards need to satisfy many requirements, use cases, and applications, they need to be extremely flexible and adaptive. The Open Cloud Computing Interface (OCCI) family of specifications aims to achieve this goal: originally developed for the deployment of infrastructure Clouds, it can also be used in different service and deployment models. This article will outline the OCCI specifications and demonstrate how they can be used in data management-related setups. Not only can OCCI be easily integrated but it can also be used to deploy data-centric applications (which are secured by SLAs), support data-awareness in scheduling, as well as directly interface with data management tools in a PaaS-based manner. To demonstrate this, three use cases are discussed in this article.

2.1 Introduction

Next to traditional HPC and Grid computing, Cloud computing has become a new driver for the global IT market. The overall idea is to deliver a service to the customer. Instead of traditionally boxing and shipping of software products,

A. Edmonds (✉)
Intel Ireland Branch, Collinstown Industrial Park, Leixlip, County Kildare, Ireland
e-mail: andrewx.edmonds@intel.com

T. Metsch
Platform Computing GmbH, Europaring 60, 40878 Ratingen, Germany
e-mail: tmetsch@platform.com

A. Papaspyrou
Technische Universität Dortmund, Institut für Roboterforschung, 44221 Dortmund, Germany
e-mail: alexander.papaspyrou@tu-dortmund.de

S. Fiore and G. Aloisio (eds.), *Grid and Cloud Database Management*,
DOI 10.1007/978-3-642-20045-8_2, © Springer-Verlag Berlin Heidelberg 2011

software is now delivered as a service to the customer directly. This change in use of computing services changes the IT landscape drastically – not only will data centers most probably transform into service providers but also the way service providers and customers interact will change.

One example is billing in all businesses where a Pay-per-Use model can be easily established. The next major change in this area will be the management of data: starting with the idea of moving compute resources to the data (data-aware scheduling) as an obvious step also the way how data is treated in the Cloud (manipulation of data – NoSQL vs. Relational Databases vs. Virtual Disc Images) will evolve. Countless other opportunities such as signing, tracing changes and movement of data are still ahead of us.

Since many customers move into the cloud the deployment of their data and the applications becomes very important to them. Still, most Cloud computing providers currently focus on providing *Infrastructure-as-a-Service* (IaaS)[1] but this might change as the industry moves its focus into the idea of providing *Platform-as-a-Service* (PaaS) where services are constructed on a higher (non-OS, but rich API) level to provide services surrounding the data.

Still, the underlying technology is evolving: standards are being developed and technologies emerge (like virtualisation). As such, there is a demand for ensuring clean interfaces and protocols which are easy to use and can be used for multiple kinds of service offerings to prevent a vendor lock-in.

In the context of these developments, the Open Cloud Computing Interface (OCCI) working group works towards forming such a standard. The OCCI family of specifications can be used for IaaS and PaaS offerings. In this paper, it is demonstrated how OCCI can be used in data-centric setups for IaaS and PaaS offerings. To this end, a setup is described in which Virtual Machines (containing Databases etc.) can be deployed in a Cloud environment while ensuring certain Service Level Agreements (SLAs). Another use case demoes the ability of OCCI for moving compute resource towards large datasets. The last scenario works (in contrast to the former two) towards a PaaS scenario: it shows a Key-Value store implementation over OCCI.

The purpose of these use cases is to show the need for an interoperable Cloud interface/protocol which can be used in all layers of the Cloud stack. Furthermore, it demonstrates that OCCI provides flexible usage models for a very heterogeneous field of scenarios in the broader field of data management in the Cloud.

The rest of the paper is organised as follows: in Sect. 2.2, the OCCI family of specification is introduced. Next, three use cases for the application of OCCI are exemplified in Sects. 2.3–2.5. Finally, the paper concludes with a summary of achievements and shows future work.

[1] Usually in the form of virtual machines.

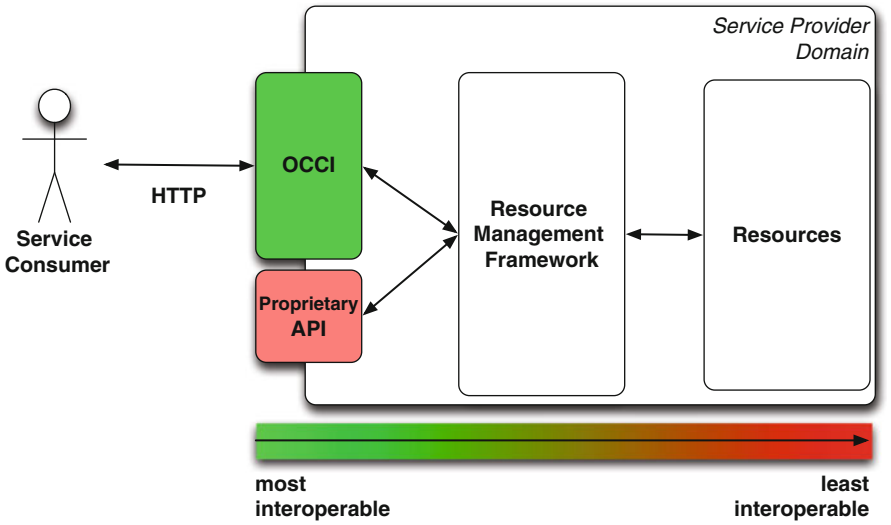

Fig. 2.1 OCCI and its position in the service provider context

2.2 Open Cloud Computing Interface

OCCI is an effort driven by a working group in the standards track of the Open Grid Forum.[2] It strives to create an open, interoperable protocol and API for the Cloud.

The group started with a clear focus on provisioning IaaS but later extended the focus to include other layers in the Cloud stack as well. The following diagram (Fig. 2.1) shows where OCCI fits in the service provider context.

The OCCI protocol can be used for integration, ensuring interoperability and portability between service providers. Proprietary APIs can be used alongside OCCI in the case that other features than those of OCCI are maintained.

The specification strives to be very easy, flexible and extensible. Therefore, it is broken into different modules. It starts with a module describing the core models. Another module describes how this model can be mapped and rendered using a HTTP/REST approach. The third module describes the infrastructure entities and how they related to the core model.

2.2.1 Motivation for Standards

Main driver for standards in the past has been interoperability. This is still a fundamental part of what standards want to achieve. Still there are nuances in the term interoperability which are important and need to be looked upon separately:

[2] http://www.ogf.org/.

Interoperability. Describes how two services can inter-operate on the fly. This demands a standardised API and protocol (e.g. live migrating a virtual machine from one host to another, which are in different management domains).
Integration. Describes how a service provider can bring together different technologies and interconnect them within his domain (e.g. integrate a virtual machine management tool with an identity management system).
Portability. This is mostly about the porting between service providers. In comparison with interoperability, there is no direct connection between the service provider. This demands that there are standardised data formats which providers can understand (e.g. porting a virtual machine from one hypervisor to another).
Innovation. Standards have always been started when a field in the IT community gains popularity, is widely adopted and begins on a path of commoditisation. Next to interoperability, standards can be a driver for innovation as well as widely adopted innovations can demand standards.
Reusability. This can be seen on two levels. First the reuse of (legacy) codes through basic standardised APIs and the reuse of the standard itself in different fields.

2.2.2 The Core Model

The core meta-model [10] for OCCI imposes a general means of handling general resources, providing semantics for defining the type of a given entity, describing interdependencies in between different entities, and defining operating characteristics on them. Although the meta-model aims to ease the implementation burden by setting a common ground for other OCCI-related specifications, it can be used as a standalone component in other contexts (e.g. Resource Oriented Architectures (ROAs)) as well.

The UML class diagram shown (Fig. 2.2) gives an overview of the OCCI core meta-model. At its heart lies the `Resource` type. Any resource exposed through OCCI is a `Resource` or sub-type thereof. A resource can be for example a virtual machine, a job in a job submission system, a user, etc. The `Resource` type contains a number of common attributes that domain-specific `Resource` types inherit. The `Resource` type is complemented by the `Link` type which associates one `Resource` instance with another. The `Link` type also contains a number of common attributes that domain-specific `Link` types inherit.

`Entity` is an abstract type which both `Resource` and `Link` inherit. Each sub-type of `Entity` is identified by a unique `Kind` instance. The `Kind` type comprise the classification system built into the OCCI model. `Kind` is a specialisation of `Category` and introduces additional capabilities in terms of `Action` types.

2.2.2.1 Classification and Identification

The OCCI model provides a built-in classification system allowing for safe extension towards domain-specific usage. This system is like a “type system” but with the possibility of being easily exposed over a text-based protocol.

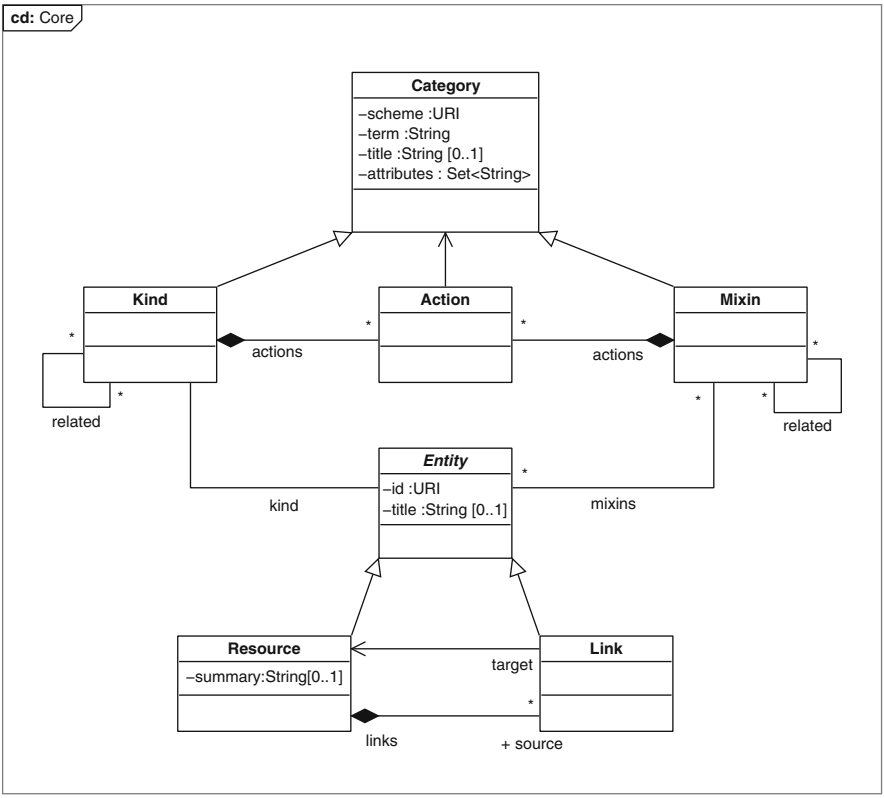

Fig. 2.2 UML class diagram of the OCCI model. The diagram provides an overview of the OCCI model but is not a standalone definition thereof

The classification system can be summarised with the following key features:

- Each OCCI base type and extension thereof is assigned a unique identifier, a structural `Kind`, which allows for dynamic discovery of available types.
- The relationship of structural `Kinds` is part of the system and thus the inheritance model is also discoverable.
- The classification system allows non-structural `Kinds` to be assigned to resource instances adding new capabilities using a mix-in-like model.
- Tagging of resource instances is supported through mix-in of non-structural `Kinds` which have no additional capabilities defined.
- A collection of associated resources is implicitly defined for each structural and non-structural `Kind`. That is all resource instances associated with a particular `Kind` instance form a collection.

2.2.2.2 Categorisation

The `Category` type comprises the basis of the identification mechanism used by the OCCI classification system. Instances of the `Category` type are only used to identify `Action` types. All other uses of `Category` properties are managed through its sub-type `Kind`.

A `Category` is uniquely identified by concatenating the categorisation scheme with the category term, for example *http://example.com/category/scheme#term*. This is done to enable discovery of `Category` definitions in text-based renderings such as HTTP. Sub-types of `Category` such as `Kind` inherit this property.

2.2.2.3 Kind Relationships

The OCCI base types `Resource` and `Link` extend `Entity`. This together with any further sub-typing implies a hierarchy of related structural `Kind` instances. The `Kind` relationships thus mirror the type inheritance structure of the OCCI model and any extension thereof.

In an example where a domain-specific "Custom Compute Resource" is a sub-type, the OCCI infrastructure type Compute, which in turn is a sub-type of the `Resource` type, four related structural `Kind`s would be involved.

One or more `Entity` instances associated with the same `Kind`, automatically form a collection, and each `Kind` identifies a collection consisting of all `Entity` instances of it. For example, an instance of the `Resource` type will always be associated with the structural `Kind` (*http://scheme.ogf.org/occi/core#resource*) and thus part of the collection implied by the `Kind`.

Collections are, by definition of the core model, navigable and support the following operations:

- Retrieve the whole collection.
- Retrieve a specific item in a collection.
- Retrieve a subset of a collection.

2.2.2.4 Discovery

In addition to that, `Kind`s and `Category` instances a particular service provider support can be discovered. By examining these instances a client is enabled to deduce the following information:

- The `Entity` sub-types available from a service provider, including domain-specific extensions.
- The attributes associated with each `Entity` sub-type.
- The invocable operations, that is `Action`s, defined for each `Entity` sub-type.
- Additional mix-ins or tags, that is non-structural `Kind`s, applicable to `Entity` sub-type instances.

Overall, the OCCI core meta-model provides a solid foundation for the remote management of resources offered in an *as-a-Service* manner, allowing for the development of interoperable tools for common tasks including deployment, automatic scaling and monitoring. The explicit split-out of it allows the leverage of the developed models, protocols, and APIs in manners not anticipated and to foster modularity and extensibility for future usage paradigms.

2.2.3 RESTful HTTP Rendering of the OCCI Model

The OCCI Core model which is described in the previous Sect. 2.2.2 is free of any rendering and forms the base of OCCI. Based upon this model, OCCI describes a serialisation rendering. This rendering – or serialisation format – is passed on the wire between client and service, see [11].

OCCI has a default rendering which is text based and uses the HTTP protocol and implements a ROA, see [14]. In this architecture, a system is modelled as a set of related resources. ROA's use Representation State Transfer (REST), see [6], to cater for client and service interactions. In these interactions, clients request to perform operations on the state of an individual or set of resources managed by the service.

HTTP is commonly used in most ROA systems. It provides means to uniquely identify resources through URIs as well as operating upon them with a set of general-purpose operations called verbs. These HTTP verbs map loosely to the resource-related operations of *create* (`POST`), *retrieve* (`GET`), *update* (`POST`, `PUT`) and *delete* (`DELETE`).

2.2.3.1 Rendering of Resources

Each Resource in the OCCI core model will be rendered as a unique URI (for example *http://example.com/foo*). Each resource can be identified uniquely by an URI and has at least one `Category` assigned, which defines the type and the operations that can be performed. This means that from this standpoint a resource can be almost anything like a Database entry, a Virtual Machine, an Image, etc.

Resources can be linked and actions can be performed upon them. Resource of the same type (as in have the same `Category` assigned) can be found under a certain path relative to the root of the service provider (e.g. all storage devices will appear under the path */storage* – still the path name "storage" is freely defined by the Service Provider and can do discovered through the Query interface).

Since `Categories` cannot only be used to define the type of the resource, but also to tag or group resources, resource can show up under multiple paths. The following URL hierarchy demonstrates this feature:

```
/compute/123
/storage/discABC
/database/tableXYZ
/nosql/entry_1
/my-linux-vms/123     // links to /compute/123
/my-datasets/discABC  // links to /storage/discABC
/my-datasets/tableXYZ // links to /database/tableXYZ
/my-datasets/entry_1  // links to /nosql/entry_1
```

This very flexible system allows that the OCCI model can be used for several use cases including for Data Management operations.

2.2.3.2 Discovery of Capabilities Through a Query Interface

One of the main features of OCCI is that clients can discover the capabilities of the service provider through a standardised query interface. This is important since OCCI is designed for extensibility. To query the capabilities of a Service provider implementing OCCI, the Client needs to do a HTTP `GET` on the URI */-/*.

This `Category` management URL allows the client to get a listing of all categories supported by the provider. Should the provider allow and support client-created categories, then this URL endpoint must support the creation of user categories as well.

2.2.3.3 Linking and Performing Actions on Resources

Each of these resources can be linked with other resources. Links again are RESTful resources and have a source and a target attribute. Each link resource is bound to a category identifying it as a link.

Next to linking, some type specific actions can be performed. The set of possible actions is defined by the `Category` of the resource. Actions are triggered by adding a fragment to the URI of the resource indicating which action should be triggered (e.g. *http://example.com/foo;action=shutdown*). Parameters of the action are described in the HTTP message.

2.2.3.4 Use of HTTP Features

The HTTP rendering of OCCI makes use of many HTTP features. This includes for example HTTP headers for Versioning and all Authentication features. OCCI does not explicitly define those but makes use of those features.

Next to these basic features, OCCI also makes use of the *Content-Type* definitions. At a minimum, all information for OCCI resources is transferred in the HTTP

Body. This is defined as the basic *text/plain* content type. Other content types also exist. For example, the information can also be rendered in the HTTP Header or as HTML (e.g. for browsing the OCCI interface using a Web Browser) by supplying the appropriate content-type header as specified in the specification.

Rendering of data is done through simple key value associations. Also, more structural data representations such as JSON of RDFa can easily be added to OCCI.

2.2.4 OCCI for Virtual Machine (Infrastructure) Provisioning

Having described the core model and a way of rendering it on the wire, a concrete compliment to the core model is now explored [12].

The infrastructure specification extends the core model at two key points:

1. To represent various infrastructure-related resources, it extends `Resource` using inheritance.
2. To represent concrete relationships between infrastructural resources, it extends `Link` using inheritance.

To represent the main elemental resources found in infrastructure-type services, OCCI has three specialisations of Resource:

1. `Compute`: Information processing resources.
2. `Network`: Interconnection resources.
3. `Storage`: Information recording resources.

Complimenting these, to allow linkage are:

- `NetworkInterface`: Represents an L2 client device (e.g. network adapter).
- `StorageLink`: Represents a link from a Resource to a target Storage Resource.

The relations of these infrastructure resources are shown in the UML diagram (Fig. 2.3).

When modeling elements, it was found that OCCI needed to support not only generic cases but also specific cases. This issue was exemplified by `Network`. It might be immediately attractive to model all functionalities within this `Resource`, including aspects of IP configuration, however, then the model would force certain technology choices upon implementers. To avoid this, the working group chose to utilise the OCCI mix-in capabilities to avoid such a situation. Where an implementer wishes to offer TCP/IP functionality on top of the `Network` resource, they can do so by implementing the `IPNetworking` mix-in. The `IPNetworking` mix-in allows to supplement the Network Resource with the necessary TCP/IP features. Should an implementer wish to offer another type of L3/L4 technology for example AppleTalk or IPX, then they only need implement a custom network mix-in.

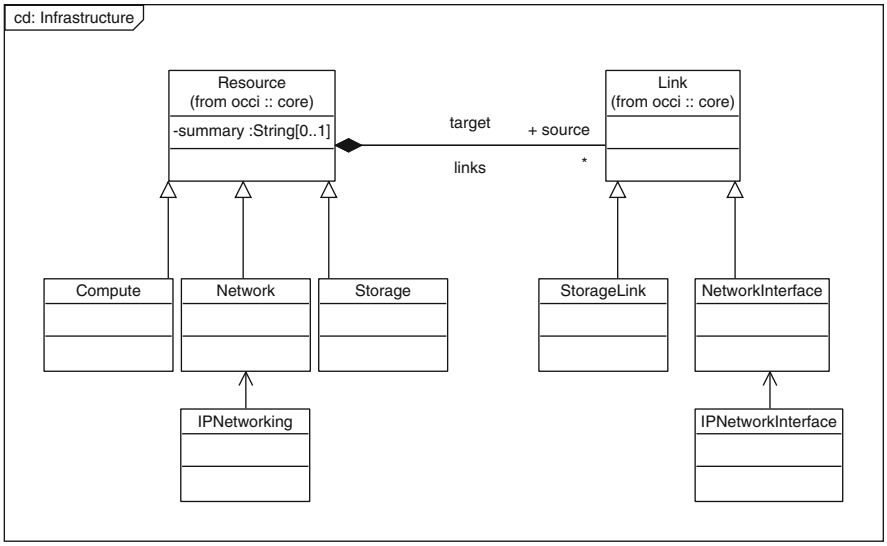

Fig. 2.3 Extended OCCI core model showing infrastructure elements

It was the infrastructure model, along with the OCCI core and HTTP rendering specifications, that aided a successful collaboration that investigated the integration of two large European Union FP7 research projects, SLA@SOI[3] and RESERVOIR.[4] The proposed integration was detailed in a subsequent technical paper [13].

2.2.5 Related Standards and Specifications

A guiding principle in OCCI is to make use of existing standards and specifications where appropriate.

OCCI and the Storage Networking Industry Association's (SNIA)[5] Cloud Data Management Interface (CDMI) working groups have collaborated together so that both specifications are interoperable with each other. It states that

> "The SNIA Cloud Data Management Interface (CDMI) is the functional interface that applications will use to create, retrieve, update and delete data elements from the cloud. As part of this interface the client will be able to discover the capabilities of the cloud storage offering and use this interface to manage containers and the data that is placed in them. In addition, meta-data can be set on containers and their contained data elements through this interface" [16].

[3] http://www.sla-at-soi.eu/.

[4] http://www.reservoir-fp7.eu/.

[5] http://www.snia.org/.

OCCI and the Distributed Management Task Force's (DMTF)[6] Open Virtualization Format (OVF), see [4], can be easily integrated through the use of the resource-type Link. Where a provider wishes to supply an OVF representation of a client's resource instance(s), they can do so by associating the instance(s) with a mirror representation, only the serialisation format is OVF.

Other than this, the OCCI working group is closely working together with other groups inside of the Open Grid Forum. The Distributed Computing Infrastucture Federation (DCI-fed[7]) working group focuses on the creation of models and APIs for setting up distributed federated computing environments. Other than this, the OCCI working group uses Standards like those developed by the Distributed Resource Management Application API (DRMAA[8]) working group for common Job operations on Clusters via the OCCI protocol.

2.3 SLA Assured Provisioning of Database Services Using OCCI

In today's service marketplaces including cloud-based ones, there exist basic limitations in service offerings. Typically, the customer has little say in what is offered by a service provider and is left with a "take it or leave it" situation. Not only is the customer faced with such a dilemma, with little possibility of negotiation but if they do accept the service offering there is little in the way of service transparency and so detections of service violations are impossible unless that customer implements custom violation detection systems. The SLA@SOI project seeks to address these challenges by providing three major benefits:

Predictability and Dependability: The quality characteristics of service can be predicted and enforced at run-time.
Transparent SLA Management: SLAs defining the exact conditions under which services are provided/consumed can be transparently managed across the whole business and IT stack.
Automation: The whole process of negotiating SLAs and provisioning, delivery and monitoring of services will be automated allowing for highly dynamic and scalable service consumption.

In this section, a use-case that combines the OCCI model and API with an SLA management framework to provide an SLA assured database service is described. In today's service marketplace, there exists a number of service providers who offer database services, for example, the Amazon Relational Database Service,[9]

[6] http://www.dmtf.org.
[7] http://forge.gridforum.org/sf/projects/dcifed-wg.
[8] http://www.drmaa.org.
[9] http://aws.amazon.com/rds.

Microsoft SQL Azure[10] and Longjump Platform as a Service.[11] Other than offering a basic, non-negotiable, non-machine readable SLA, these service providers do not offer certain guarantees that particular consumers will require. A case in point is where a third party service provider wishes to process personal and identifying information. Many law jurisdictions will require that user-supplied data and the processed resultant data remain within the protection of that jurisdiction, which may mean that the physical location of that data must always remain in the country or region where that jurisdiction has powers to protect. If that data at any one time falls outside of those defined physical locations due to actions taken by the service provider that the third party uses, then regardless of knowing or not knowing about such actions, the third party can be liable under the relevant laws set out by the jurisdiction. In the use case presented here, an SLA management framework provides the means to:

1. Customise a service offering.
2. Negotiate on that service offering to the satisfaction of the third party and their legal responsibilities.
3. Be notified when terms of the agreed service offering deviate and have deviations logged as an audit trail.

The use case is realised by the third party provisioning the offered database service using the OCCI API through the facilities of the SLA manager. OCCI provides the standard and interoperable means of provisioning the required database service and the SLA manager provides the means as outlined above. That database service is realised as a pre-built virtual machine with all the requisite database software installed and configured, which once provisioned is accessible by the consumer. The service provider offers means to monitor the agreed terms in the SLA and, in particular for this use case, allows for the physical location of the virtual machine to be monitored. This allows the SLA management framework to monitor constantly the physical location of the virtual machine and in the case that the virtual machine is migrated to an inappropriate physical location the third party will instantly receive notification of that event and logs will provide an audit trail.

2.3.1 SLA@SOI SLA Management Framework

SLA@SOI defines a holistic view for the management of SLAs and implements an SLA management framework that can be easily integrated into a service-oriented infrastructure (SOI), see [17]. The main innovative features of the project are:

- An automated e-contracting framework
- Systematic grounding of SLAs from the business level down to the infrastructure

[10] http://www.microsoft.com/en-us/sqlazure.

[11] http://www.longjump.com.

- Exploitation of virtualisation technologies at infrastructure level for SLA enforcement
- Advanced engineering methodologies for creation of predictable and manageable services

The accompanying diagram (Fig. 2.4) illustrates the anticipated SLA management activities throughout the business/IT stack.

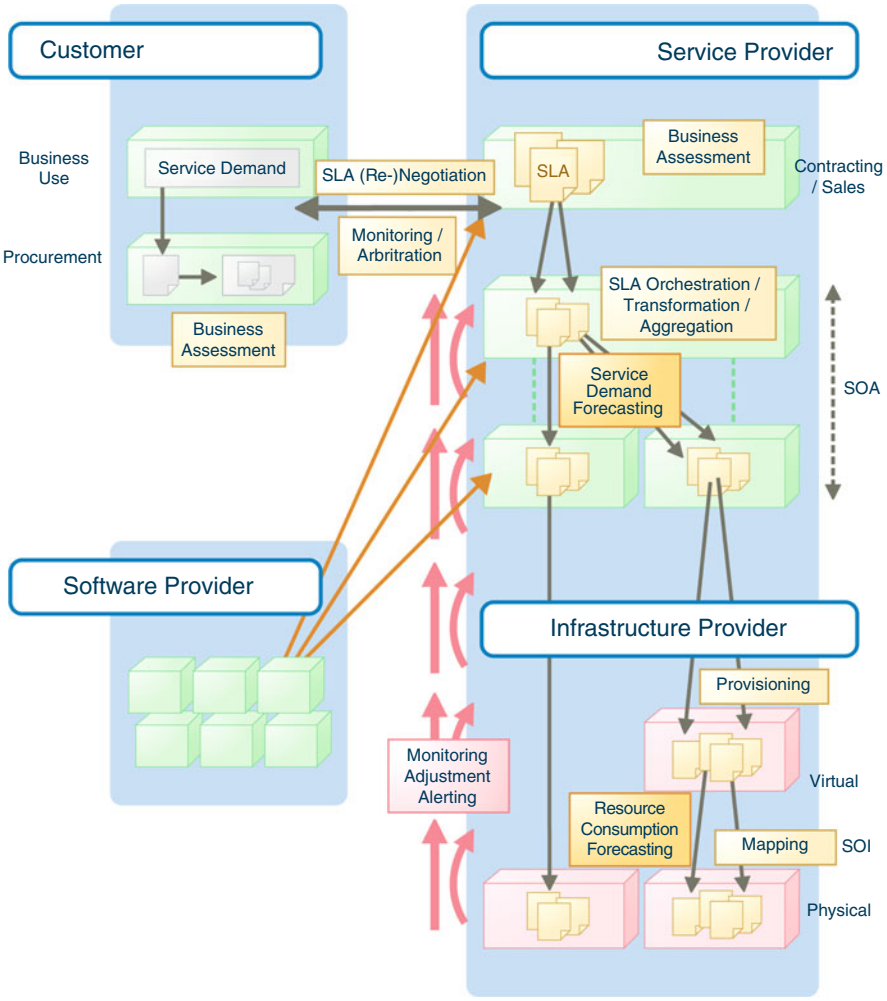

Fig. 2.4 SLA@SOI overview

2.3.2 SLA@SOI and OCCI

In this use case, there are two main components that are required for realisation. The first and most fundamental is a Service Manager that offers a database service.

The Service Manager is the entity that is responsible for providing the client's service. Relevant to this use case is that the Service Manager provides database services as preconfigured virtual machines and that the location of those virtual machines can be monitored. The SLA@SOI framework makes no assumption on this Service Manager only that it has an interface:

1. That can create, retrieve, update and delete its managed services.
2. Through which service instance metrics can be listed and retrieved.

The second entity required is the SLA@SOI framework's SLA Manager. This is a set of both generic and domain-specific components. What is generic relates to the management of SLA templates (what a provider offers) and SLA instances (what a provider runs on their clients behalf and guarantees). The domain-specific components are those that interact with the particular service manager that provides the client services. Further details of the SLA@SOI framework and its architecture can be found, see [18].

The SLA Manager offers to clients one or more SLA Templates, which is expressed using the SLA@SOI SLA model. Through either a UI or API, the client can select, customise, negotiate and provision an SLA-guaranteed service. In the use case scenario, this would entail the third party specifying what physical location (e.g. region, country) is required for their regulatory compliance.

Once the SLA Manager is acting on the client's behalf, it first negotiates with the service manager using the OCCI query interface. The OCCI query interface allows for the various Resource types to be queried for and interrogated and in particular to this use case, the locations that a provider can provision their virtual machines. As an extension to the query interface, SLA@SOI will also allow for per-user quotas to be queried. Using this extension, an SLA Manager can tell whether a client's request will be fully satisfied or not by the current service provider. Having established that the client's quota is sufficient, the next step can either take two paths. The first is that the provisioning of the requested service is done automatically or, second, the provisioning must be explicitly executed by the client. Where a provisioning request is executed in one or the other manner, the next responsibility of the SLA Manager will be to call the provisioning functionality of the Service Manager (relationship and interaction is shown in Fig. 2.5). This again is looked after by the OCCI API and an OCCI request from the SLA manager's domain specific components is sent to the Service Manager. As soon as the provisioning request is successful, the SLA manager then begins to monitor the provisioned service, including the location of the service's virtual machine. It does this by monitoring the various terms of the agreed SLA (e.g. QoS metrics).

For the SLA Manager, the major advantage of choosing to implement OCCI as a means to talk with Service Managers is that in the case where a particular

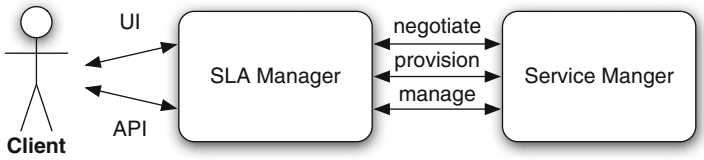

Fig. 2.5 Relationship between SLA manager and service manager

Service Manager cannot satisfy the provisioning of the requested service resources due to insufficient client quotas or unsuitable virtual machine deployment locations, the SLA Manager can, with the necessary logic implemented, look up the next registered service provider and seek to have the remaining service resources provisioned there, without any need for Service Manager protocol or API changes. Such functionality makes an excellent case for SLA-mediated cloud brokerage use cases.

2.4 On-Demand Data-Aware Provisioning of Services

A different application area for OCCI appears in the context of traditional community-based Distributed Computing Infrastructures (DCIs): modern research more and more relies on cross-institutional, cross-project data processing. In many communities, scientists quickly state the requirement to enable exchange of information beyond traditional boundaries such as project collaborations or long-term Virtual Organisations. Rather than that, a more flexible, more agile approach is expected.

This development poses a major challenge not only for the management of data itself (i.e. ensuring authentication and authorisation, planning distribution and replication, and tracking provenance), but also for the management of its computation: workload needs to run close to the data in most cases (since data is usually large), but the compute infrastructure available in the direct proximity of the data may not necessarily provide the correct environment. That is, applications to process the data might be missing, the operating system does not match the application requirements, or – on a higher level – certain services needed for data analysis and manipulation have not been deployed on-site.

Beside, many communities run their own, proprietary workload management software, tailored to the specific needs of their users. As such, it is usually not an option to require a central system, often referred to as a *meta scheduler*[12] for all users of all communities. Rather than that, additional technology needs to be incorporated, which allows dynamic federation of planning domains depending on the current demand.

[12]Mostly found in the context of traditional Grid Computing.

The D-Grid Scheduler Interoperability project (DGSI) in the context of the German D-Grid Initiative[13] aims to provide a solution both issues through the development of a standards-based protocol between Meta schedulers. DGSI approaches interoperability DCIs from two sides, namely *Activity Delegation* (taking care of the handover of workload from one domain to the other), and *Resource Delegation* (taking care of the leasing of resources from one domain to another). Assuming the DGSI protocols and services in place, the notion of delegation can help to avoid traditional data management techniques such as decoupled copying, prefetching, and replication at all.

2.4.1 The Climate Community Use Case

The effects of climate change are one of the major challenges of mankind: stakeholders of many areas strive for strategies to deal with the consequences of pollution and man-made changes to the environment. The basis of all decision making are models of climate processes and the understanding of interplay of the enormous amount of parameters in them. Since the beginning of industrialisation in the nineteenth century, Earth System Science, one of the data sciences, investigates these processes, their chemical formation in the diverse subsystems such as oceans, atmosphere or biosphere, and their long-term influence on climate.

From those investigations, researchers nowadays possess very detailed insight into climate development. This rests on the permanent acquisition, cataloging, and processing of very large (Peta scale) volumes of experimental and model data, as well as the continuous re-evaluation of scientific results using refined models.

Current information technology provides potent means to accelerate these processes of data evaluation and simulation. High Performance Computing (HPC) infrastructure, high speed networks, and modern storage architectures support archival, preprocessing, selection, and transportation of large data amounts as well as the computation of highly demanding simulations (e.g. short term weather forecast or storm track analysis).

For the latter scientific analysis, researchers filter and examine geopotential heights to track and predict the movement of low-pressure areas over time with regard to a given climate model. This is essential as storms and cyclones typically cross such areas [3]. This analysis and simulation is based on long-term acquired global climate data.

Usually, scientists are only interested in a restricted area for a Stormtrack analysis and have to reduce the amount of available base data to the region of interest. Besides a complex combination of several steps, this resorts to either access to a specific amount of climate data (Fig. 2.6a) or execute simple visualisation workflows on selected and preprocessed data (Fig. 2.6b).

[13] http://www.d-grid.de.

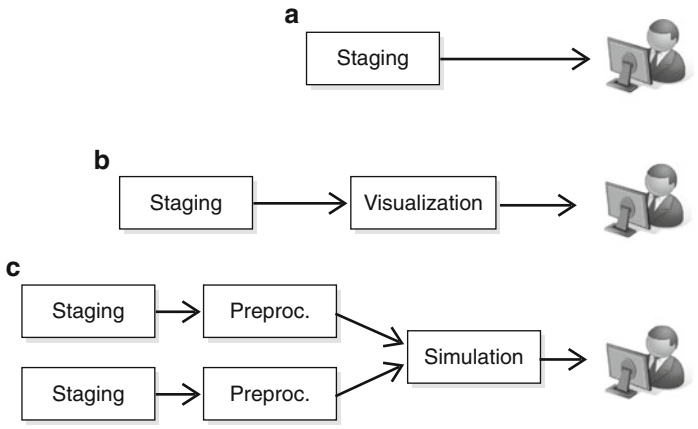

Fig. 2.6 Three simple workflow examples from C3Grid: (**a**) selective data download; (**b**) simple simulation on input data; (**c**) simulation on a set of preprocessed input data

Today, they generally have two possibilities to retrieve the desired data: they either get access to the storage and download the full amount of data (i.e. full replication is performed) or use proprietary programs to reduce the amount of data at the storage site and download the desired data set afterwards. In the first case, the required local storage may simply not be available to the single scientist, or the providing institution may not be willing to provide an external party with access to the full archive due to strategical considerations. In the second case, the user needs to cooperate directly with the data provider, basically via two mechanisms:

1. Having to use tools that are installed, but potentially not known to him,[14] or
2. Having to roll out the software on her own, either doing this as part of the batch processing job or in cooperation with the resource provider.

Obviously, the former is not acceptable from a user's perspective. The latter in turn requires extensive manual intervention and additionally necessitates the acquisition of user rights to retrieve or even locally process the requested data. As most of the climate data is stored in a distributed way, the procedure often has to be repeated for several data sites. Furthermore, it leaves intelligent, automatic load balancing totally to the user, which is generally not desirable. In addition to that, this traditional approach of application deployment massively hinders cross-community collaboration, if they rely on different infrastructure technologies: if the user takes care by himself to deploy the application as part of his computational workload, the number of resources compatible will usually be very restricted.

[14] With the exception of widely accepted and distributed tools such as the Climate Data Operators [15].

2.4.2 An Approach for Dynamic, Cross-Community Resource Allocation

Most DCI environments share the ability to efficiently distribute user workload to the resources available within their community. This issue, usually generalised under the term *Meta Scheduling*, is already very diverse within a community: both submitted jobs and available resources differ considerably, to the extent that coordination has to handle specialised knowledge about usage scenarios and infrastructure. This leads to very different, community-specific approaches for the development of Grid scheduling services [9].

The DGSI provides a standards-based interoperability layer for scheduling and resource management services in DCI ecosystems. By allowing the users of a community to distribute the workload among resources within the management domain of another community while keeping the individual, specialised scheduling solutions being run by the communities, it offers new perspectives for community collaboration, resource federation, and efficient utilisation. The general architecture is depicted in Fig. 2.7.

2.4.2.1 Delegation Models

The DGSI protocols foresee two scenarios to be considered: the delegation of activities and the delegation of resources:

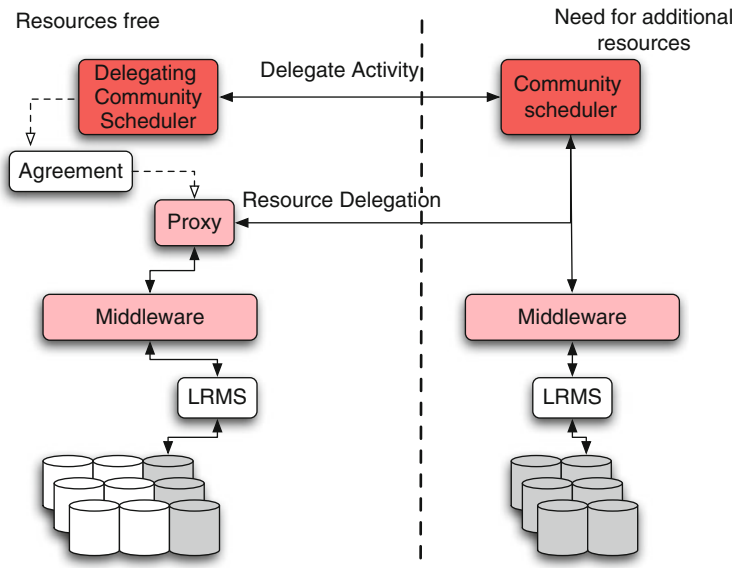

Fig. 2.7 Overall delegation architecture using the DGSI protocols. Meta schedulers from different domains (architectural, organisational, and technological) cooperate using activity and resource delegation

Delegation of activities. By means of DGSI, one meta-scheduler is able to delegate activities, that is single or parallel jobs or workflows, to another meta-scheduler from a different management domain (i.e. another community). By use of WS-Agreement [1], JSDL [2], and OGSA-BES [8], DGSI provides a standardised way of handing over workload to the other domain: a set of jobs that cannot be executed in the local scheduling domain can be channeled to another one (assuming the resource requirements of the jobs match the provided environment) to minimise waiting time induced by a high load on the originating side of the delegation. Via the mechanisms of SLA negotiation and agreement (as provided by the WS-Agreement protocol), it is ensured that both requirements and fulfilment can be negotiated in a reliable manner.

While the initial use case for activity delegation assumes an environment that requires cross-domain load balancing for workload to amplify user experience in a federated DCI environment, it is obvious that, with respect to data management, the very same mechanisms enable Meta Scheduling systems to easily move the workload close to the data: even if the data is assumed in a different community domain, proximity-based approaches for data-aware scheduling systems are easy to implement over the federated nature of the DGSI protocols.

Delegation of resources. To complement the handover of workload between DCIs in a more "as-a-Service"-related manner, the DGSI protocols also support the delegation of resources from one domain to another. This allows one meta-scheduler to effectively "lease" resources from another one over a given period of time and use them in the same way as managed resources within the own domain. Again, by use of WS-Agreement, GLUE, and middleware provisioning, a standardised means for requesting, negotiating, agreeing, monitoring, and provisioning those resources is available: after successfully agreeing on the "lease" contract, the scheduler that requests resources can effectively incorporate them into his planning algorithms for management over the time of lease.

The original use case was tailored to the specific needs of cross-community collaboration: the provisioning mechanisms were merely used to dynamically provide a management endpoint (i.e. a specific Grid middleware) that the requesting scheduler is able to cope with. For example, an environment that is generally managed by UNICORE [5] can provide a resource lease to a scheduler that manages its resources through Globus Toolkit [7] just by provisioning a Globus GRAM endpoint for the leased resources while – at the same time – ensuring the fulfillment of the negotiated SLAs through injected monitoring and enforcement mechanisms. From the perspective of data management, especially in the context of proximity-aware deployment of applications close to their data, much more can be done: by leveraging the provisioning interface to the deployment of the user's application rather than the middleware only, the user is provided with a unified view on the lower-level infrastructure and thus can run his application on a much larger resource space than given in traditional approaches. On the other hand, the meta-scheduler enjoys much more freedom in deploying the application close to the data, without having to give up its planning mandate (as in activity delegation).

2.4.3 The Role of OCCI for a Data-Aware Delegation Scenarios

The OCCI family of specifications, especially the infrastructure rendering, is the key enabler for introducing data-awareness into the different delegation scenarios.

Figure 2.8 depicts the role of OCCI in the overall process.

Enabling activity delegation. While OCCI is not strictly necessary for the activity delegation scenario, it makes the dynamic provisioning of a delegation channel (in case of the initial usage scenario of DGSI a service such as OGSA-BES) much easier. That is, the meta-scheduler that accepts workload delegation can dynamically

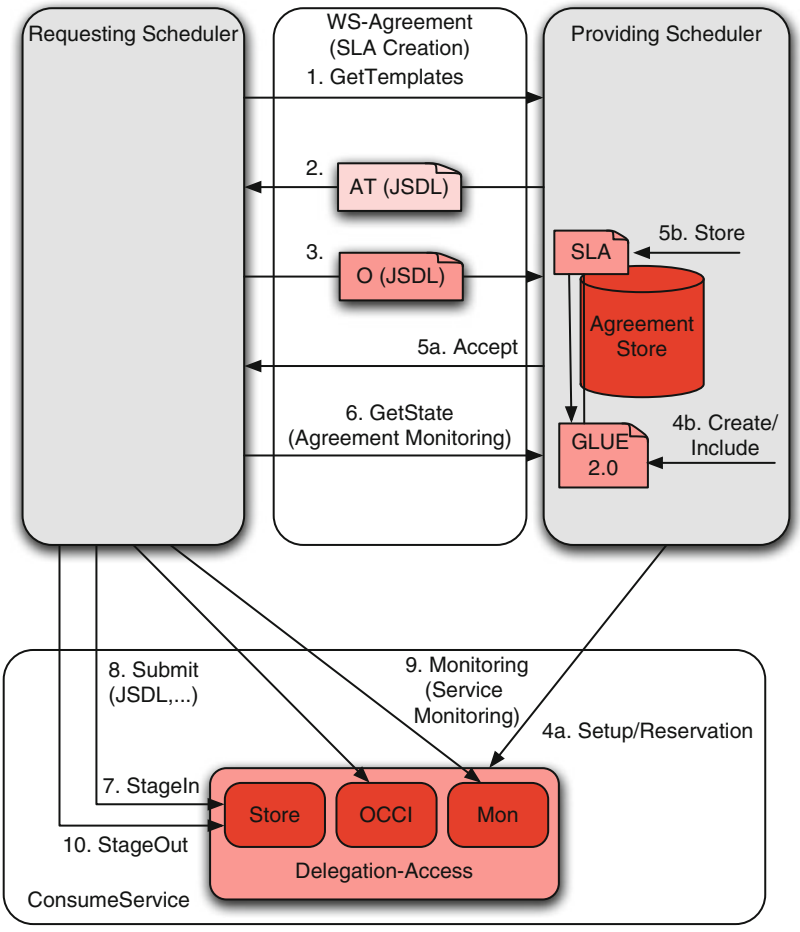

Fig. 2.8 Ten steps for negotiating a delegation within the DGSI protocol stack. While activity delegation only requires six steps (up to the *GetState* call for agreement monitoring), resource delegation runs to completion of the tenth step. Note, however, that – from step six and forth – each step is specific to the concrete usage of the delegated resource (e.g. a single job submission)

create instances of a submission and monitoring interface on its own infrastructure without having to provide own resources for the purpose. OCCI ensures in this context a unified view on underlying resources that can run demand-based, lifecycle-managed middleware services to the DCI ecosystem.
Enabling resource delegation. Here, OCCI is a strong requirement for allowing the user to deploy her own applications in the context of a virtualised environment over a unified interface. With OCCI in place, description, status management, and provisioning of virtual machines can be not only unified within the community itself, thus even there providing strong benefit, but rather beyond the boundaries of domains, allowing easy deployment of applications on the resources of a different community.

As such, OCCI fulfills two major requirements to enable this technology: interoperability and portability of the applications, and dynamic provisioning of infrastructure. The packaging paradigm of Virtual Machines additionally allows easy movement and infrastructure-agnostic capacity planning with data requirements in mind. Speaking of such, OCCI is the enabling technology for making data aware, proximity-based scheduling and resource management happen in federated DCI environments.

2.5 Use of OCCI for a Simple Key-Value Store

The previous Sects. 2.3 and 2.4 showed the usage of OCCI in virtualised environments (but data-centric setups). This last use case shows how the exact same standard can be used to give a database application a RESTful standard OCCI compliant interface.

A very simple use case is taken to demonstrate the abilities of OCCI as a data management front-end interface. Many NoSQL databases such as CouchDB[15] are deployed with a built-in RESTful interface. With the proliferation of NoSQL databases and their various RESTful APIs, there is a perceivable need for a standardised interface through which a client could discover the abilities and functionalities of the service provider (and in this use case the NoSQL Database).

Clients can then decide which service provider to use. This is essentially important since Cloud computing is all about delivering services experience to the customer. The customer should decide which service to use based on the experience, the functionalities and the price the service provider offers.

The discovery interface described in the OCCI section of this paper describes these self-discovery features. Section 2.3 on SLA@SOI describes how the OCCI core model can be extended for provisioning virtual machines.

[15] http://couchdb.apache.org/.

In this use case, the Core model is only extended by the one class which derives from `Resource`. It is called a *Key-Value resource* and has two attributes: `key` and `value`. A simple flip actions is defined. When the client queries the discovery interface, it will see the `Category` definition of this resource type:

```
> GET /-/ HTTP/1.1
> User-Agent: curl/7.21.0 (x86_64-pc-linux-gnu) [...]
> Host: localhost:8888
> Accept: */*
> Cookie: [...]
>
< HTTP/1.1 200 OK
< Content-Length: 517
< Etag: "89c0aeace4f7209b57d38cb0c4877bb9b22ad7a4"
< Content-Type: text/plain
< Server: pyocci OCCI/1.1
<
Category: keyvalue;
          scheme=http://example.com/occi/keyvalue;
          title=A Resource which holds a Key and a Value;
          location=/keyvalues/;
          rel=http://schemas.ogf.org/occi/core#resource;
          attributes=key value;
          actions=flip
Category: flip;
          scheme=http://example.com/occi/keyvalue;
          title=Flips the key and the value;
          attributes=foo bar
Category: keyvaluelink;
          scheme=http://example.com/occi/keyvalue;
          title=A link between two Key Value Resources;
          location=/keyvalues/links/;
          rel=http://schemas.ogf.org/occi/core#link;
          attributes=source target
```

The `GET` on the path */-/* indicates that one wants to discover what the service provider offers. It returns a `Category` definition showing the scheme of the category and which attributes it supports. As there will be no actions, this is all the `Category` features.

Now Key-Value resources can easily be created using this `Category` and retrieved through the OCCI interface:

```
> POST / HTTP/1.1
> User-Agent: curl/7.21.0 (x86_64-pc-linux-gnu) [...]
> Host: localhost:8888
> Cookie: [...]
> Content-Type: text/occi
> Category: keyvalue;
            scheme=http://example.com/occi/keyvalue;
```

```
> X-OCCI-Attribute: key=foo, value=bar
>
< HTTP/1.1 200 OK
< Content-Length: 2
< Content-Type: text/html; charset=UTF-8
< Location: /users/foo/keyvalues/dba17696-[...]
< Server: pyocci OCCI/1.1
<
```

The request indicates the type (via `Category`), a Key-Value resource, that the new resource should be. Also, two attributes are delivered alongside providing values to the key and value attributes of this new resource instance. The service will return a location of the new resource. This location can be used to retrieve the resource instance:

```
> GET /users/foo/keyvalues/dba17696-[...] HTTP/1.1
> User-Agent: curl/7.21.0 (x86_64-pc-linux-gnu) [...]
> Host: localhost:8888
> Accept: */*
> Cookie: [...]
>
< HTTP/1.1 200 OK
< Content-Length: 191
< Etag: "6bad49cb7785101006593a9fe79d5b54a4a19516"
< Content-Type: text/plain
< Server: pyocci OCCI/1.1
<
Category: keyvalue;
          scheme=http://example.com/occi/keyvalue
Link: </users/foo/keyvalues/dba17696-[...]?action=flip>
X-OCCI-Attribute: value=bar
X-OCCI-Attribute: key=foo
```

The response tells what type the REST resource is (via the `Category` header). It also returns us the two attributes which where defined during the creation of the resource.

Updating the attributes can be done using the HTTP `PUT` verb and it provides a new set of attributes. Deletion of the resource can be done through the HTTP `DELETE` verb.

Next to these HTTP basic renderings, the implementation[16] used for this example can also render OCCI using HTML by specifying the *text/html* content-type. This allows the user on client side to use the browser to discover the Query interface and the resources using a web browser (Fig. 2.9).

[16] http://pyssf.sf.net/.

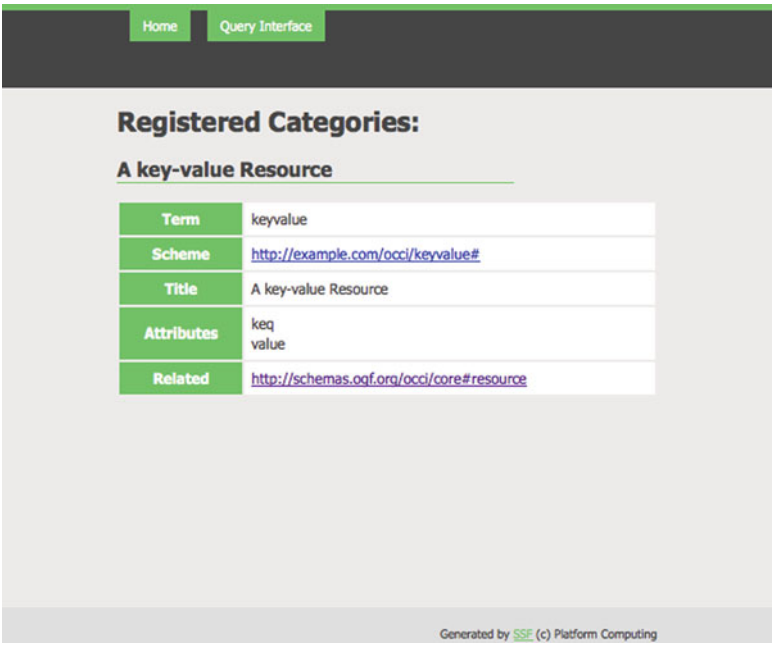

Fig. 2.9 Screen-shot of an HTML rendering of the OCCI query interface

This is a very generic interface and shows that OCCI can be used for provisioning infrastructure as well as PaaS based offerings.

Regardless of the type (OCCI `Kind`), a REST-Resource (represented through an URI) represents the interface will not change. This even means that the GUI (Fig. 2.9) is also generic and it would look and work the same for different types of `Resources`. The implementation of the OCCI interface demoed here can therefore be used for Infrastructure provisioning or other Platform offerings (like Job submission for Clusters).

2.6 Conclusions

With the last use case, the authors of this paper want to demonstrate the flexibility and extensibility of the OCCI interface. OCCI can be used using different setups especially the discovery functionalities and the extensible Core model support this. This demonstrates that OCCI can be used in IaaS and PaaS setups which relate to Data management.

This paper strives not to give a complete overview of all possible setups regarding OCCI and Data management. Still it demoed how some setups can be

implemented using OCCI. Most often OCCI plays the role of ensuring and safeguarding Interoperability as described in Sect. 2.2.1.

Several implementations of the OCCI specification exist notworthly does in research projects and currently ongoing working in commercial applications. Most notable is the currently in development effort which tries to incorporate the OCCI standard in the OpenStack[17] Cloud framework. Research Projects like the previously noted RESERVOIR and SLA@SOI (see Sect. 2.2.4) have adopted OCCI as well.

Next to this interoperability aspect, it is important to state that OCCI does not try to replace existing proprietary interfaces. It is defined for interoperability means as described before. Service Providers can still use their proprietary API/Interface to deliver higher-level functionalities, which is very specific to their offerings.

This idea of brokerage could either be realised in an automated fashion or with the user's interaction. Still OCCI makes this idea possible. Without an interoperable interface, a Cloud Broker of querying different cloud providers using one client would be impossible. Indeed, there is a current trend to enable interoperability through the use of facade/proxy service intermediaries. This is but a temporary solution as this approach leads to additional overhead in terms of inefficiencies, additional maintenance, configuration and management. This is something that OCCI seeks to remove and solve by doing so.

Next to driving adoption, the OCCI working group will focus on standardized interfaces for advanced reservation, monitoring and billing techniques. Also semantic enabled renderings will be added to the specification. Currently, the group is looking into JSON, XML or RDF/RDFa renderings.

What this paper demonstrated is that OCCI can be used on many layers of the Cloud Stack (IaaS and PaaS) and is possibly one of the small but important contributions to realise Cloud offerings. Even when narrowing the field to Data Management in Cloud and Grids, the OCCI interface can and must play a roll as an enabler.

Acknowledgements The authors acknowledge the contributions of all members of the OCCI working group. This work is partially supported by the German Ministry of Education and Research under project grant #01IG09009, and is partially supported by the European Community Seventh Framework Programme (FP7/2001-2013) under grant agreement no.216556.

References

1. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web Services Agreement Specification (WS-Agreement). In: Standards Track, no. GFD-R.107 in The Open Grid Forum Document Series, Grid Resource Allocation Agreement Protocol (GRAAP) Working Group, Muncie (IN) (2007)
2. Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, S., Pulsipher, D., Savva, A.: Job Submission Description Language (JSDL) Specification, Version 1.0. In: Standards Track, no. GFD-R.56 in The Open Grid Forum Document Series, Job Submission Description Language (JSDL) Working Group, Muncie (IN) (2005)

[17] http://www.openstack.org.

3. Blackmon, M.L.: A climatological spectral study of the 500 mb geopotential height of the northern hemisphere. J. Atmos. Sci. **33**, 1607–1623 (1976)
4. Crosby, S., Doyle, R., Gering, M., Gionfriddo, M., Hand, S., Hapner, M., Hiltgen, D., Johanssen, M., Leung, J., Machida, F., Maier, A., Mellor, E., Parchem, J., Pardikar, S., Schmidt, S.J., Warfield, A., Weitzel, M.D., Wilson, J.: Open virtualization format specification. In: Grarup, S., Lamers, L.J., Schmidt, R.W. (eds.) Standards and Technology, no. DSP0243 in DMTF Specifications, Distributed Management Task Force (2009)
5. Erwin, D.W., Snelling, D.F.: UNICORE: A grid computing environment. In: Sakellariou, R., Gurd, J., Freeman, L., Keane, J. (eds.) Proceedings of the 7th International Euro-Par Conference, Lecture Notes in Computer Science (LNCS), vol. 2150, pp. 825–834. Springer, Heidelberg (2001)
6. Fielding, R.T.: Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine (2000)
7. Foster, I., Kesselman, C.: Globus: A toolkit-based grid architecture. In: The grid: Blueprint for a future computing infrastructure, pp. 259–278, 1st edn. Morgan Kaufman, San Mateo (1998)
8. Foster, I., Grimshaw, A., Lane, P., Lee, W., Morgan, M., Newhouse, S., Pickles, S., Pulsipher, D., Smith, C., Theimer, M.: OGSA Basic Execution Service Version 1.0. In: Standards Track, no. GFD-R.108 in The Open Grid Forum Document Series, Open Grid Services Architecture Basic Execution Services (OGSA-BES) Working Group, Muncie (IN) (2006)
9. Grimme, C., Papaspyrou, A.: Cooperative negotiation and scheduling of scientific workflows in the collaborative climate community data and processing grid. Future Generat. Comput. Syst. **25**, 301–307 (2009)
10. Metsch, T., Edmonds. A., et al.: Open Cloud Computing Interface – Core and Models. In: Standards Track, no. GFD-R in The Open Grid Forum Document Series, Open Cloud Computing Interface (OCCI) Working Group, Muncie (IN) (2010)
11. Metsch, T., Edmonds, A., et al.: Open Cloud Computing Interface – HTTP Rendering. In: Standards Track, no. GFD-R in The Open Grid Forum Document Series, Open Cloud Computing Interface (OCCI) Working Group (2010)
12. Metsch, T., Edmonds, A., et al.: Open Cloud Computing Interface – Infrastructure. In: Standards Track, no. GFD-R in The Open Grid Forum Document Series, Open Cloud Computing Interface (OCCI) Working Group, Muncie (IN) (2010)
13. Metsch, T., Edmonds, A., Bayon, V.: Using cloud standards for interoperability of cloud frameworks. Tech. rep., SLA@SOI project (FP7 ICT-2007.1.2-216556. http://sla-at-soi.eu/wp-content/uploads/2010/04/RESERVOIR-SLA@SOI-interop-techReport.pdf(2010)
14. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media, Sebastopol (CA) (2007)
15. Schulzweida, U., Kornblueh, L.: CDO User's Guide, Climate Data Operators, Version 1.0.6 (2006)
16. Slik, D., Siefer, M., Hibbard, E., Schwarzer, C., Yoder, A., Bairavasundaram, L.N., Baker, S., Carlson, M., Nguyen, H., Ramos, R.: Cloud data management interface. In: SNIA Technical Position Series, 1st edn. Storage Network Industry Association, San Francisco (2010)
17. Theilmann, W., Yahyapour, R., Butler, J.: Multi-level SLA management for service-oriented infrastructures. In: Mähonen, P., Pohl, K., Priol, T. (eds.) Towards a Service-Based Internet, Lecture Notes in Computer Science (LNCS), vol. 5377, pp, 324–335. Springer, Heidelberg (2008)
18. Theilmann, W., Happe, J., Ellahi, T., Torelli, F., Kearney, K., Lambea, J., Fuentes, B., Vuk, M., Guinea, S., Edmonds, A., Nolan, M., Brosch, F., Kotsokalis, K.: Deliverable D.A1a: Framework Architecture (full lifecycle). Tech. rep., SLA@SOI project (FP7 ICT-2007.1.2-216556, http://sla-at-soi.eu/wp-content/uploads/2009/07/D.A1a-M26-FrameworkArchitecture.pdf(2010)

http://www.springer.com/978-3-642-20044-1

Grid and Cloud Database Management
Fiore, S.; Aloisio, G. (Eds.)
2011, X, 353 p., Hardcover
ISBN: 978-3-642-20044-1