jOCCI – General-Purpose OCCI Client Library in Java

Michal Kimle, Boris Parák, Zdeněk Šustr[*]
CESNET z. s. p. o.
Zikova 4, 160 00, Praha 6, Czech Republic
E-mail: kimle.michal@gmail.com, boris.parak@cesnet.cz, zdenek.sustr@cesnet.cz

The Open Cloud Computing Interface (OCCI) standard by OGF has become widely adopted in various cloud environments, such as the EGI Federated Cloud. It is currently supported by mainstream open source cloud management frameworks, e.g., OpenStack (through OCCI-OS) or OpenNebula (rOCCI) as well as others, less wide-spread ones. It is likewise supported by many workflow and submission tools used by user communities – VMDIRAC, JSAGA or SixSq. SlipStream to name but a few. OCCI is, however, found somewhat lacking in the availability of general-purpose clients supporting the standard. Only recently, its use was enabled only in the Ruby programming language through the rOCCI Framework, and command-line/scripting use was facilitated by the rOCCI-cli client. Naturally, there has been long-standing demand for OCCI support in other programming languages, primarily in Java. It has now been answered by the introduction of jOCCI – a native Java library implementing the OCCI class structure, rendering and transport specifications, currently in accordance with the OCCI v. 1.1 specification. Provided by the same product team already producing the rOCCI framework, it is more than "just another feature in the cloudscape." Rather than a simple translation of the client part of rOCCI, it is a brand new product – a choice that has been made not only to make it a truly native Java library, but also to introduce additional, independent, client to validate the generic functionality of existing OCCI server applications. This work describes the new library and compares it to rOCCI in terms of design and interoperability when used against other server-side OCCI implementations. It also discusses the relative merits of implementing the client library as a fresh product, relatively separate from rOCCI, rather than just providing Java bindings for the client side of the rOCCI Framework. Finally, the future of rOCCI and jOCCI is briefly discussed in view of the emerging OCCI v. 1.2 specification.

International Symposium on Grids and Clouds (ISGC) 2015
15 – 20 March, 2015
Taipei, Taiwan

[*]Speaker.

1. Introduction

Open standards are essential for building interoperable infrastructures. In the world of infrastructure clouds standards are now abundant, and providers adopt them at various speeds in their IaaS frameworks. While server-side adoption is gradually picking up, general client availability may be lacking. This article discusses a recent addition among the choice of client-side options.

To begin with, Section 2 introduces current open standards for infrastructure clouds, focusing primarily on the Open Cloud Computing Interface (OCCI). Next, Section 3 introduces the newly developed jOCCI library – an OCCI client library implemented in Java. Finally, Section 4 shows how this library fits into the existing cloudscape, especially from the point of view of the EGI Federated Cloud, and what was the motivation to implement it alongside the existing Ruby library.

2. Cloud Standards

Open standards for the IaaS (Infrastructure as a Service) model of cloud service delivery were – as is often the case – late in coming to the scene, especially compared to proprietary protocols, now known as widely accepted de-facto standards. Among open standards, at least the following are worth mentioning:

CDMI – ***Cloud Data Management Interface*** [1], introduced in 2010 by SNIA (Storage Networking Industry Association), focusing solely on object storage. As such, it complements rather than competes with later standards for virtual machine management.

OCCI – ***Open Cloud Computing Interface*** [2], introduced in 2011 by the OGF (Open Grid Forum [3]), is a text-based protocol intended primarily for describing and managing IaaS resources. Its highly generic design, however, makes it easily extensible to other areas of application, such as PaaS. OCCI is discussed in more detail in Subsection 2.1.

CIMI – ***Cloud Infrastructure Management Interface*** [4], introduced in 2012 by DMTF (Distributed Management Task Force), is another protocol for managing IaaS resources. Compared to OCCI it relies less on extensibility, defining numerous attributes to describe the properties of said resources directly in the body of the standard.

Compare the release dates of those standards with Amazon EC2 [5], which has been first introduced in 2006. As already stated above, open standards in this case trail proprietary industrial solutions by several years.

2.1 OCCI Description

The Open Cloud Computing Interface is a standard of choice for the EGI Federated Cloud platform [6], which currently federates sites running different cloud management frameworks, namely OpenNebula, Open Stack and Synnefo. Technologies used by EGI (namely the `rOCCI-server` component) also allow for the inclusion of resources managed by Amazon Web Services into the federation, and there is development towards supporting MS Azure in the same manner. EGI Federated Cloud platform also supports CDMI for managing object storage at its sites.

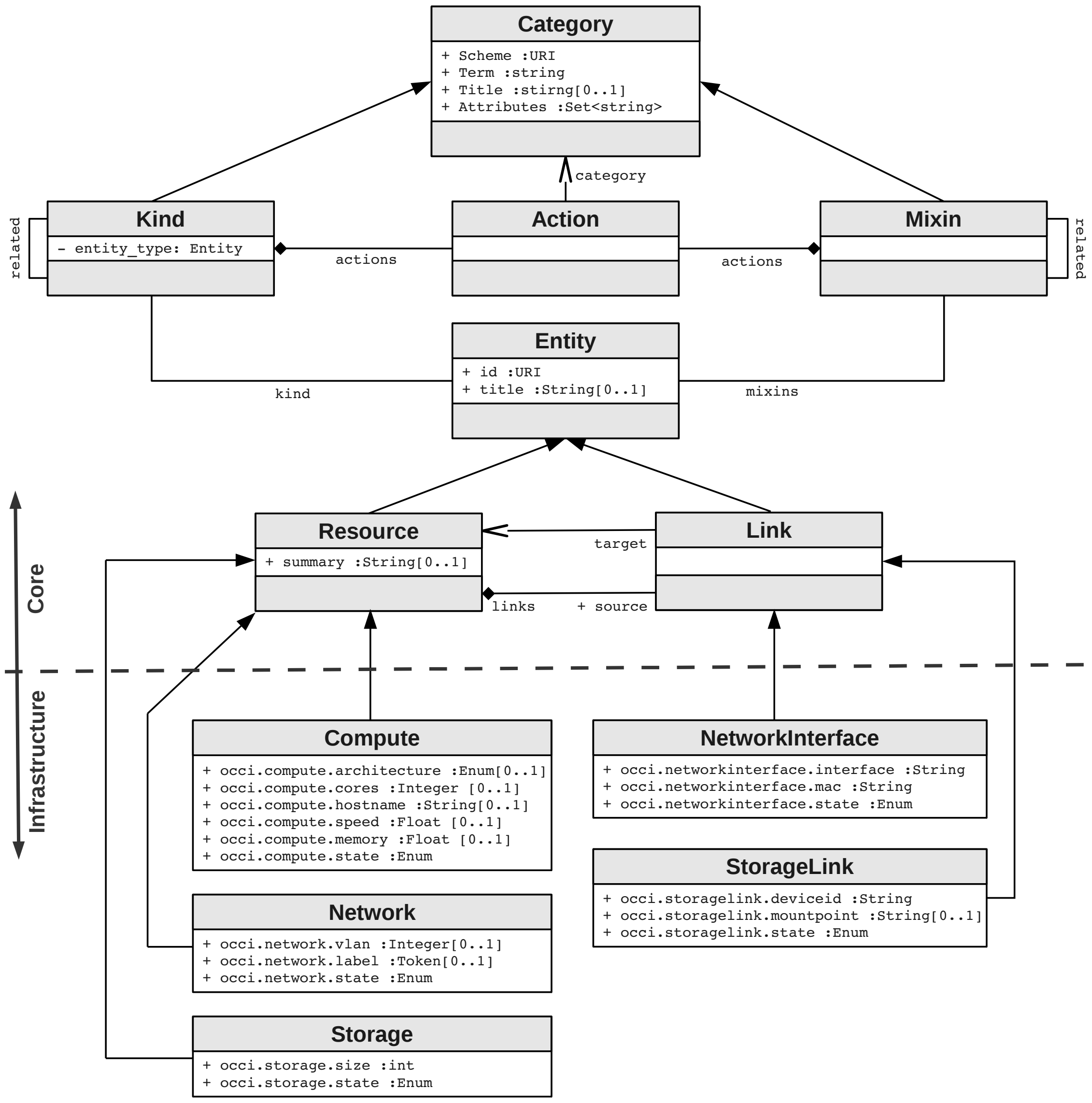
Category
+ Scheme :URI
+ Term :string
+ Title :stirng[0..1]
+ Attributes :Set<string>
category
Kind
- entity_type: Entity
related
actions
Action
actions
Mixin
related
Entity
+ id :URI
+ title :String[0..1]
kind
mixins
Core
Infrastructure
Resource
+ summary :String[0..1]
target
links
+ source
Link
Compute
+ occi.compute.architecture :Enum[0..1]
+ occi.compute.cores :Integer [0..1]
+ occi.compute.hostname :String[0..1]
+ occi.compute.speed :Float [0..1]
+ occi.compute.memory :Float [0..1]
+ occi.compute.state :Enum
NetworkInterface
+ occi.networkinterface.interface :String
+ occi.networkinterface.mac :String
+ occi.networkinterface.state :Enum
StorageLink
+ occi.storagelink.deviceid :String
+ occi.storagelink.mountpoint :String[0..1]
+ occi.storagelink.state :Enum
Network
+ occi.network.vlan :Integer[0..1]
+ occi.network.label :Token[0..1]
+ occi.network.state :Enum
Storage
+ occi.storage.size :int
+ occi.storage.state :Enum

Figure 1: Abstract OCCI Core classes (top) extended with the OCCI Infrastructure specification (bottom) matching real-world concepts such as "compute" or "storage" resources.

As the CIMI standard had yet to be released at the time of drafting the EGI Federated Cloud design, there was no actual comparison done between OCCI and CIMI. No such comparison is therefore given in this article.

The current OCCI specification is available in version 1.1, consisting of three separate documents. The Core specification [7] introduces an essential (core) class structure, forming a common basis for various assumed extensions of the standard. The Infrastructure specification [8] is one such extension, which addresses the area of virtualised resource management. While the Core specification operates with rather abstract classes such as `Resource` or `Link`, Infrastructure maps much more closely to the real world with definition of classes to describe `Compute`, `Network` and `Storage` resources, and links inbetween, such as network interfaces (the `NetworkInterface` class) and mounted storage (the `StorageLink` class). See Figure 1 for a diagram illustrating the OCCI Core class structure extended with the OCCI Infrastructure specification.

The final part of the current OCCI specification – OCCI HTTP Rendering [9] – lays down the rules for expressing OCCI classes in text, typically included in HTTP headers. A rendered object may typically look as the following example:

```
Category: compute;scheme="http://schemas.ogf.org/occi/infrastructure#";class="kind"
X-OCCI-Attribute: occi.core.id="ee13808d-7708-4341-a4ba-0e42e4818218"
X-OCCI-Attribute: occi.core.title="exampleVM"
X-OCCI-Attribute: occi.compute.cores=1
X-OCCI-Attribute: occi.compute.memory=1.7
X-OCCI-Attribute: occi.compute.architecture="x86"
X-OCCI-Attribute: occi.compute.speed=1
X-OCCI-Attribute: occi.compute.state="active"
```

The above excerpt is a rendered representation of a `compute` resource, i.e., a virtual machine. Various attributes specify its size and other properties. This is what a description of a virtual resource returned by a cloud management framework would typically include.

The HTTP rendering has several obvious shortcomings, chief among them the fact that it is rather difficult to parse, limited in size by the fact that it is transported inside HTTP headers, and also unsuitable for describing multiple resources at once. Therefore it cannot be used to describe a collection of resources (such as a VM cluster) in a single message. These problems will be overcome in a new release – OCCI 1.2 – which will introduce and prefer JSON rendering of OCCI objects.

OCCI 1.2 is currently available for public comment and is expected to be released later in 2015. It is discussed in greater detail in Section 5.

2.2 Implementing OCCI

To successfully implement OCCI, both the client side and the server side must support the typical OCCI workflow, which is shown and described in Table 1 in a somewhat simplified form.

There are currently several server-side OCCI implementations in existence. Among those, there are a few aiming at full standard coverage (i.e., a general-purpose full-fledged OCCI interface), although they may not be fully implemented at the moment. Among those, at least the following should be mentioned:

occi-os [10] – an OCCI interface in Open Stack

rOCCI-sever [11] – a standalone “translator” service implementing OCCI interfaces for a number of cloud management frameworks, namely OpenNebula, Amazon Web Services and (currently in development) MS Azure

SNF-OCCI [12] – an OCCI interface in Synnefo

Contrary to the relative multitude of server-side implementations, there has so far been only a single general-purpose OCCI client – the `rOCCI-cli` [13], part of the rOCCI Framework. That does not mean that there are no other OCCI clients in existence, but they are single-purpose implementations, typically providing only a subset of actions required by the workflow in their given user group.

Client must be able to . . .	**Server must be able to . . .**
Receive, parse and understand model	**Render model**
The model is an overview of the server's capabilities. It lists available OS templates (images) and Resource templates (pre-defined virtual machine sizes), actions supported by the underlying cloud management framework (for instance only some frameworks support the `snapshot` action), and generally describes what the server side can do.	
Render request, validate and send	**Parse request and act on it**
The request typically contains an Action and a Category. The Action is for instance `List` or `Create` (full list incidentally given in Subsection 3.1.2), and the Category is an instance of an OCCI `Category` class, or a descendant thereof, on which the Action should be performed. Before the request is even sent to the server, it is validated locally against the model to check whether it is in the server's capabilities to perform the action required. This minimizes communication and load on the server in general as it filters out requests that would inevitably fail with a "not found" or "not supported" anyway.	
Receive response	**Return response**
In the response, the three-digit HTTP return code indicates success or failure (or even that the request has been queued and will be processed asynchronously), and a rendered object gives details of the result, for instance a rendered `Compute` resource instance as shown in Subsection 2.1.	

Table 1: Simplified schema of a single OCCI request workflow

The `rOCCI-cli` package is implemented on top of a pair of libraries (`rOCCI-core` and `rOCCI-api`), allowing developers to make their own clients using the Ruby programming language. Traditionally, JRuby bindings were also being produced for `rOCCI-core` and `rOCCI-api`, making the libraries theoretically available to Java developers. However, maintenance of that dual set of libraries was proving too limiting and too costly, until it was finally decided to implement a completely independent set of native Java libraries – the jOCCI. jOCCI is the main object of this article and is described in the next section.

3. The jOCCI Library

The jOCCI project represents a set of Java libraries implementing the aforementioned OCCI standard. jOCCI currently consists of two client-side components: `jOCCI-core` and `jOCCI-api`. `jOCCI-core` covers basic OCCI class hierarchy from both OCCI Core [7] and OCCI Infrastructure [8] and relations between them. `jOCCI-api` uses `jOCCI-core` and together they create a communication layer for OCCI clients and servers. Both components are in their early versions and the whole project is still under lively development.

3.1 Components

The two components perform distinguished functions.

3.1.1 jOCCI-core

As mentioned before, `jOCCI-core` represents OCCI model classes and their relations. Furthermore, `jOCCI-core` provides methods for parsing and rendering plain-text representations of OCCI classes. This functionality is crucial for transporting data between the client and the remote server via HTTP messages. In addition to this, `jOCCI-core` also validates any OCCI requests with respect to known OCCI model. This helps to avoid a creation of requests that would be rejected by server, even before they are sent.

`jOCCI-core`'s main part consists of representation of OCCI class hierarchy. OCCI core classes are situated in package `cz.cesnet.cloud.occi.core`. The package contains basic classes: `Category`, `Kind`, `Mixin`, `Action`, `Entity`, `Resource`, `Link` and two additional classes: `ActionInstance` (representing an instance of an OCCI *action* class) and `Attribute` (representing and attribute of OCCI entities and its properties).

OCCI infrastructure classes `Compute`, `IPNetwork`, `IPNetworkInterface`, `Network`, `NetworkInterface`, `Storage` and `StorageLink` are available – all together – from package `cz.cesnet.cloud.occi.infrastructure`. All the classes extend either the OCCI *Resource* or *Link* class and come with methods for accessing and modifying their specific attributes. All the attributes' values that can be represented as enumerations are included in the `cz.cesnet.cloud.occi.infrastructure.enumeration` package.

OCCI entities can be grouped either in class `Model` or class `Collection`. `Model` represents the OCCI model structure and can contain meta classes such as `Kind`, `Mixin` or `Action`. On the other hand, class `Collection` serves as a container for OCCI instances – `Link`, `Resource` and `ActionInstance`. An instance of the `Model` class can be assigned to a `Collection`, which means that all the entities in the collection have the same model.

All Java counterparts of OCCI classes have methods `toText()` and `toHeaders()`, which allow them to render themselves into their text representations that can be further used in communication with a server.

In the opposite direction, the `Parser` interface and its implementation `TextParser` is used to parse OCCI entities from server responses. `TextParser` currently supports the following content types for OCCI-compliant messages:

- `text/plain` (HTTP body)
- `text/occi` (HTTP headers)
- `text/uri-list` (HTTP body, used for listings)

3.1.2 jOCCI-api

`jOCCI-api` is a Java library implementing transport functions for rendered OCCI queries. It is built on top of `jOCCI-core` and currently provides HTTP transport functionality with a set of authentication methods and basic requesting interface to easily communicate with OCCI servers.

`jOCCI-api`'s design is modular and can be easily extended if needed. The library uses the Apache HttpComponents library [14] internally for implementation of the HTTP communication protocol.

The main component of `jOCCI-api` is its abstract class `Client` and its implementation class `HTTPClient`. Classes provide a communication interface composed of five methods:

- `list` – retrieves locations of entities from remote server
- `describe` – retrieves descriptions of entities from remote server
- `create` – creates a new entity on remote server
- `delete` – deletes entities from remote server
- `trigger` – triggers an action on entities on remote server

Additionally, methods `connect` and `refresh` are provided so that client can establish a connection and update an OCCI model if needed.

`jOCCI-api` also comes with a set of HTTP authentication methods. Abstract class `HTTPAuthentication` provides a foundation for multiple implementations:

- class `NoAuthentication` – a dummy class representing no authentication method
- classes `BasicAuthentication` and `DigestAuthentication` – for BASIC and DIGEST authentication methods
- class `X509Authentication` – for authentication via X509 and VOMS certificates
- class `KeystoneAuthentication` – for authentication against OpenStack's Keystone Identity Service

All authentication methods allow to load custom CAs either from file or directory and use them during connection initialization.

The last component of `jOCCI-api` is class `EntityBuilder`. It serves as a builder for OCCI structures such as `Resource`, `Link`, `Action`, etc. according to retrieved OCCI model. This is helpful while creating instances that have to be part of the server query.

3.2 Usage

For the time being, jOCCI doesn't contain a client application that would communicate with remote server. Instead, a client can be built on top of the `jOCCI-api` library, which provides all the necessary functions a client would need.

Using `jOCCI-api` is quite straightforward. First, a client with a proper authentication method has to be instantiated and a connection established between client and remote server. The client can then communicate with the remote server via its query interface and receive and process the server's responses.

3.2.1 Creating a client

Currently the only client implementation is a class: `HTTPClient`. During its initialization one can specify server URL, authentication method, media type that will be used for HTTP messages, and whether the client should automatically connect to the server or not.

```
Client client = new HTTPClient(URI.create("https://remote.server.net"), new
    BasicAuthentication("username", "password"), MediaType.TEXT_OCCI, true);
```

3.2.2 Using authentication methods

As already mentioned above, there are four different authentication methods already prepared: `BasicAuthentication`, `DigestAuthentication`, `X509Authentication` and finally `KeystoneAuthentication`. Any `HTTPAuthentication` implementation allows loading custom CAs either from file or directory and use them during connection initialization.

```
HTTPAuthentication auth = new X509Authentication("/path/to/certificate.pem", "password");
auth.setCAPath("/etc/grid-security/certificates/"); //path to CA directory
Client client = new HTTPClient(URI.create("https://remote.server.net"), auth);
```

3.2.3 Making requests

Clients can communicate with remote servers through five methods: `list`, `describe`, `create`, `delete` and `trigger`.

Method `list` retrieves a list of locations of specified entity or locations of all entities when called without an argument. Entities can be specified by their term (if specific enough) or by their whole identifier (term + scheme, e.g., *http://schemas.ogf.org/occi/infrastructure#compute*).

```
List<URI> list = client.list();
...
list = client.list("compute");
...
list = client.list(URI.create("http://schemas.ogf.org/occi/infrastructure#network"));
```

Method `describe` retrieves a descriptions of specified entities from the remote server. Entities to describe can be specified in ways explained for method `list` and also by their location on the remote server, which will select one specific entity. The method's return value is a list of fully populated `Entity` instances (from `jOCCI-core`) carrying all the information obtained from the remote server.

```
List<Entity> entityList = client.describe("compute");
...
entityList = client.describe(URI.create("https://remote.server.net/storage/123"));
```

Method `create` creates a new entity on the remote server. The entity passed to the `create` method has to be populated at least to the extent that server will have enough information to create it. The `EntityBuilder` class can be used to simplify the process of entity creation. The return value of the method call is a location of the newly created entity.

```
Model model = client.getModel();
EntityBuilder entityBuilder = new EntityBuilder(model);
Resource resource = entityBuilder.getResource("compute");
resource.addMixin(model.findMixin("debian7", "os_tpl"));
resource.addMixin(model.findMixin("small", "resource_tpl"));
resource.addAttribute(Compute.MEMORY_ATTRIBUTE_NAME, "2048");
URI location = client.create(resource);
```

Method `delete` can be used to delete entity instances from the remote server. Entities can be specified the same way as in the `describe` method and method's return value indicates the success of deletion.

```
boolean wasSuccessful = client.delete("compute");
...
wasSuccessful = client.delete(URI.create("https://remote.server.net/storage/123"));
```

The client's last query method is `trigger`. It triggers an action on entities on the remote server. An `ActionInstance` instance has to be passed to the `trigger` method call, which represents the action that will be triggered on the entities. Again, the `EntityBuilder` class can help to create a correct `ActionInstance`. The method's return value indicates whether the action was successfully executed on selected entities or not.

```
Model model = client.getModel();
EntityBuilder entityBuilder = new EntityBuilder(model);
ActionInstance actionInstance = entityBuilder.getActionInstance("start");
boolean wasSuccessful =
    client.trigger(URI.create("https://remote.server.net/compute/456"), actionInstance);
```

4. jOCCI in the Context of Other Products

By mid 2014, there were already numerous user groups relying on OCCI in their workflows, affiliated with the EGI [15, 16, 17], as well as e-infrastructures external to EGI [18, 19]. Server-side implementations for most popular cloud management frameworks, especially OpenNebula and Open Stack, had already been in production for some time, but the choice of client-side solutions was somewhat limited. The rOCCI framework libraries were available for applications using Ruby, but most user groups chose to wrap around the rOCCI command line interface (`rOCCI-cli`), which is a valid approach, but relatively difficult to maintain and also lacking the benefits of better integrated client applications.

Demand for Java support was apparent, and the jOCCI library (section 3) was finally produced and released in early 2015, allowing Java-oriented developers among various user groups to start developing more cohesive and efficient client applications for OCCI-enabled cloud management frameworks.

Figure 2 shows an EGI-centric view of the OCCI cloudscape as it is today. While OCCI support on the server side is also growing (OpenNebula, Open Stack, Synnefo, Amazon Web Services, and an announced support for MS Azure), this article focuses on clients. There are now at least four ways get an OCCI client suitable for the needs of any given user group:

1. There can be a completely independent OCCI implementation. No full-fledged general-purpose clients are known to the authors, but a simple subset of OCCI actions can be managed even with wrapping around `curl`. That is not a recommended approach (as it skips validation altogether, for one), but it is nevertheless possible.

2. A native Ruby application building on `rOCCI-core` and `rOCCI-api`. That provides for a very efficient use of OCCI, including the ability to use OCCI concepts (class structure ...) natively in the client application.

3. An application or script wrapping around `rOCCI-cli`. Despite being considered the "option of the last resort", this approach is used surprisingly often. The main disadvantages are the fact that since the command-line client is stateless, the model must be requested from the server side on every call. Also, obviously, console output from the CLI must be parsed and that process can be broken by simple changes in the clients' printed messages. Thus, wrappers require constant maintenance. Nevertheless this approach is quite popular.

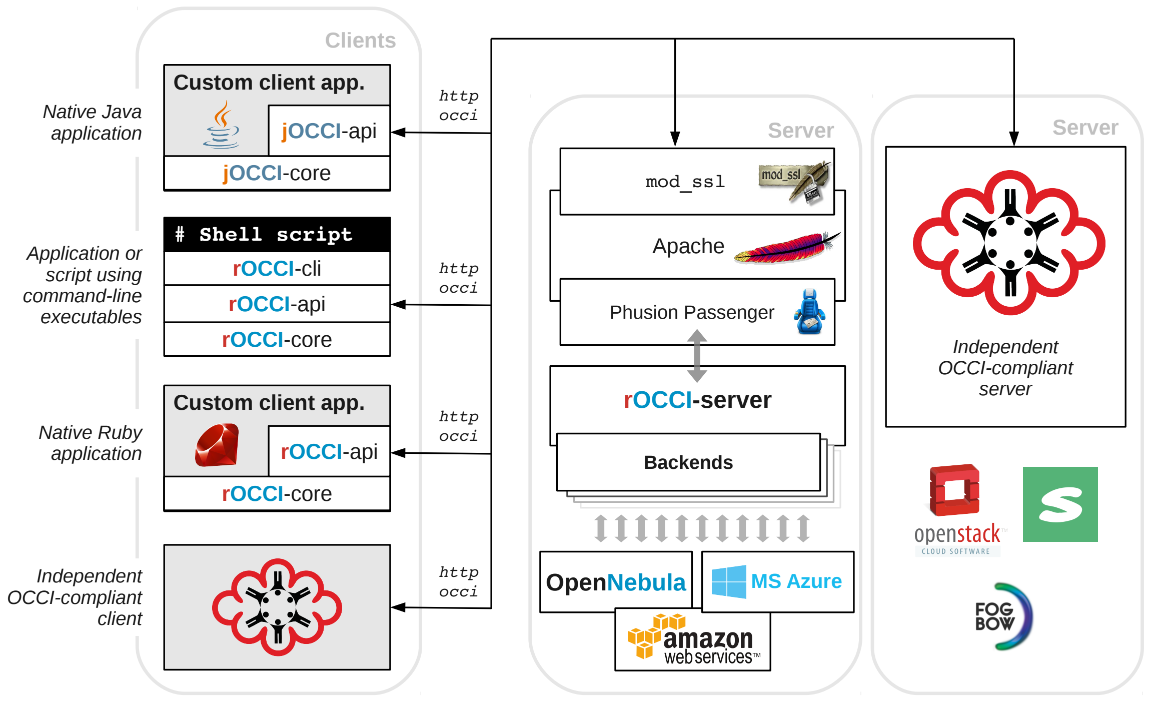

Figure 2: Various OCCI implementations on both the client and the server side, communicating over HTTP/OCCI

4. The new development: a custom client application built with Java, relying on the new jOCCI libraries. This has the same advantages as using the rOCCI libraries, plus the added benefit that Java is a language more often used (and known) in the community.

Multiple benefits are expected of the separation from rOCCI, and the decision to develop a completely independent client library:

- Multiple independent implementations are crucial to the development of open standards. Feedback from the development effort is helping to clear ambiguities and improve the next release of the standard.
- Independent implementations also make the results of (often automated) interoperability tests much more reliable, indicative of "real" interoperability rather than the same possible bug in the client side as well as the server side library (e.g., rOCCI) canceling each other out. In that manner, multiple side-by-side implementations help improve each other.
- One advantage stemming specifically from the fact that the jOCCI product team is very close to, and partly overlaps with, the rOCCI team, is that the new implementation can avoid design pitfalls found not in the development phase, but during the life cycle of the older product.
- With a native library, there is no need for emulation, cross-compilation or embedded runtime environments for other languages, greatly simplifying the development and release process of all components.

- What is more, a native library can rely more on language-specific design patterns and tools for native product maintenance and distribution. In the case of jOCCI, for instance, this means availability of OCCI libraries from the *maven* central repository.

All things considered, a separate Java library not only widens the choice for developers, but also helps improve other products through more significant test results, and allows the heavier rOCCI framework to shed its JRuby bindings.

5. Future Work

As far as it is possible to foresee, the bulk of future work will consist in implementing the upcoming OCCI v 1.2 specification. It is being released for public comment during April 2015, and there are major innovations:

- Proper *Attribute* definition in the standard – unifying the use of the term.
- Introduction of a new resource state. State `error` will be recognized on top of states defined by OCCI 1.1.
- Specification of JSON rendering. The use of JSON will be preferred from OCCI 1.2 on, replacing *text* rendering, which had various disadvantages as discussed in Subsection 2.1.
- Definition of the syntax and semantics of *pagination*, which can be used to request and download large lists or collections in parts.
- Introduction of new extensions to the OCCI standard, namely specifications for accounting and billing, SLA adherence and monitoring.
- Separation of rendering and protocol specifications into separate documents.

Contrary to previous expectations, the currently available OCCI 1.2 public comment version has no specification for XML rendering, but JSON replaces that adequately. Also, the expected redesign of the filtering mechanism is not included in OCCI 1.2.

Aside of the implementation of OCCI 1.2, future work on the jOCCI library will involve its use in cloud interoperability testing:

- *Cloud Plugfests* [20] where various implementations of cloud service clients and servers are tested against each other for interoperability in pre-set scenarios.
- *EGI Federated Cloud product test suites* for automated testing of interoperability solutions used in the infrastructure.

6. Summary

An independent general-purpose client library for IaaS cloud management over OCCI (jOCCI) has been successfully developed in Java. It answers the demand by user tool developers, who often rely on Java to develop high-level interfaces for user communities. Besides its main purpose, i.e., supporting Java, it also helps establish the OCCI as a valid open standard with multiple implementations, helping improve other OCCI-compatible products on both the client side and the server side.

7. Acknowledgements

Development of the jOCCI library was partially funded by the EGI-InSPIRE project through the European Commission's 7th Framework Programme (contract # INFSO-RI-261323).

References

[1] The Storage Networking Industry Association, *Cloud Data Management Interface (CDMI)*, [Online] Available: http://www.snia.org/cdmi [Accessed: November 20, 2015].

[2] R. Nyrén, A. Edmonds, A. Papaspyrou, and T. Metsch, *OCCI specification*, OCCI-WG OGF, 2011. [Online] Available: http://occi-wg.org/about/specification. [Accessed: November 20, 2015].

[3] *Open Grid Forum*, [Online] Available: http://www.ogf.org [Accessed: November 20, 2015].

[4] The Distributed Management Task Force, *Cloud Management Initiative*, [Online] Available: http://dmtf.org/standards/cloud [Accessed: November 20, 2015].

[5] Amazon Web Services, *Amazon EC2*, [Online] Available: https://aws.amazon.com/ec2/ [Accessed: November 20, 2015].

[6] European Grid Infrastructure, *Federated Cloud*, [Online] Available: https://www.egi.eu/infrastructure/cloud/ [Accessed: November 20, 2015].

[7] R. Nyrén, A. Edmonds, A. Papaspyrou, and T. Metsch, *Open Cloud Computing Interface – Core*, OCCI-WG OGF, 2011. [Online] Available: http://ogf.org/documents/GFD.183.pdf [Accessed: November 20, 2015].

[8] T. Metsch and A. Edmonds, *Open Cloud Computing Interface – Infrastructure*, OCCI-WG OGF, 2011. [Online] Available: http://ogf.org/documents/GFD.184.pdf [Accessed: November 20, 2015].

[9] T. Metsch and A. Edmonds, *Open Cloud Computing Interface – RESTful HTTP Rendering*, OCCI-WG OGF, 2011. [Online] Available: http://ogf.org/documents/GFD.185.pdf [Accessed: November 20, 2015].

[10] Metsch, T.; Edmonds, A., *OCCI*, [Online] Available: https://wiki.openstack.org/wiki/Occi [Accessed: November 20, 2015].

[11] B. Parák, Z. Šustr, *rOCCI-server Repository*, [Online] Available: https://appdb.egi.eu/store/software/rocci.server [Accessed: November 20, 2015].

[12] GRNET, *snf-occi's documentation*, [Online] Available: http://www.synnefo.org/docs/snf-occi/latest/ [Accessed: November 20, 2015].

[13] B. Parák, Z. Šustr, *rOCCI-cli Repository*, [Online] Available: https://appdb.egi.eu/store/software/rocci.cli [Accessed: November 20, 2015].

[14] The Apache Software Foundation, *Apache HttpComponents*, [Online] Available: http://hc.apache.org/ [Accessed: November 20, 2015].

[15] IN2P3, *JSAGA*, [Online] Available: http://software.in2p3.fr/jsaga [Accessed: November 20, 2015].

[16] DIRAC Project, *The DIRAC Interware*, [Online] Available: http://diracgrid.org [Accessed: November 20, 2015].

[17] Barcelona Supercomputing Center, *COMP Superscalar*, [Online] Available: http://www.bsc.es/computer-sciences/grid-computing/comp-superscalar [Accessed: November 20, 2015].

[18] The Cloud4E Project, *Trusted Cloud Computing for Engineering*, [Online] Available: http://www.cloud4e.de [Accessed: November 20, 2015].

[19] SixSq, *Slipstream*, [Online] Available: http://sixsq.com/products/slipstream.html [Accessed: November 20, 2015].

[20] Cloud Plugfests [Online] Available: http://www.cloudplugfest.org/ [Accessed: November 20, 2015].