Evolution of the Open Cloud Computing Interface

Boris Parák[1], Zdeněk Šustr[1], Michal Kimle[1], Pablo Orviz Fernández [2], Álvaro López García[2], Stavros Sachtouris[3] and Víctor Méndez Muñoz[4]

[1]*Department of Distributed Computing, CESNET z.s.p.o, Zikova 4, Prague, Czech Republic*

[2]*Instituto de Física de Cantabria (CSIC-UC), Avda. de los Castros s/n, Santander, Spain*

[3]*Greek Research & Technology Network, Mesogion Av. 56/11527, Athens, Greece*

[4]*Computer Architecture & Operating Systems Department, Universitat Autònoma de Barcelona, Bellaterra, Spain*

Keywords: Cloud, Standards, Architecture, Interoperability, Management, OCCI.

Abstract: The OCCI standard has been in use for half a decade, with multiple server-side and client-side implementations in use across the world in heterogeneous cloud environments. The real-world experience uncovered certain peculiarities or even deficiencies which had to be addressed either with workarounds, agreements between implementers, or with updates to the standard. This article sums up implementers' experience with the standard, evaluating its maturity and discussing in detail some of the issues arising during development and use of OCCI-compliant interfaces. It shows how particular issues were tackled at different levels, and what the motivation was for some of the most recent changes introduced in the OCCI 1.2 specification.

1 INTRODUCTION

The Open Cloud Computing Interface (OCCI) is a RESTful protocol and API for a variety of management tasks. OCCI was originally designed to create a remote management API for IaaS (Infrastructure as a Service) model-based services, allowing for the development of inter-operable tools for common tasks including deployment, autonomous scaling, and monitoring of virtual machines and related resources (Metsch and Edmonds, 2011b).

Since its publication in April 2011, OCCI has been adopted by a variety of cloud, cloud-like and cloud-adjacent platforms as the interoperability interface of choice. The strength of OCCI lies in its well though out and rather abstract core specification (Nyrén et al., 2011) resembling more a modeling language than a communication protocol. It gives OCCI its extensibility and wide-range applicability. Every other part of OCCI builds on top of the core specification by proposing various extensions targeting specific areas such as infrastructure management, monitoring, accounting, over-the-wire rendering, transport protocol, billing, and many others.

The OCCI standard specification currently (in version 1.1) consists of three separately published documents:

1. GFD.183 – OCCI Core (Nyrén et al., 2011)
2. GFD.184 – OCCI Infrastructure (Metsch and Edmonds, 2011b)
3. GFD.185 – OCCI HTTP Rendering (Metsch and Edmonds, 2011a)

However, the abstract and minimalistic nature of OCCI has its disadvantages. Most notably, significant parts of the standard require careful interpretation when creating a real-world implementation. This often leads to incompatibilities between implementations provided by different developers. This paper is an attempt to briefly introduce popular implementations of OCCI (Section 2), collect as much feedback from their developers as possible, describe most commonly encountered issues based on the aforementioned feedback (Section 3), and propose solutions wherever possible either by referencing the upcoming OCCI 1.2 standard (a set of not yet published documents, which have been, however, made available for public comment during 2015) or by suggesting future improvements (Section 4). Finally, Section 5 makes an overall statement on the suitability and maturity of OCCI as it stands today.

Parák, B., Šustr, Z., Kimle, M., Fernández, P., García, Á., Sachtouris, S. and Muñoz, V.
Evolution of the Open Cloud Computing Interface.
In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 2*, pages 339-346
ISBN: 978-989-758-182-3

2 IMPLEMENTATIONS

Over the last five years, OCCI gained multiple experimental and production-grade implementations in various states of usability. These implementations helped the overall evolution of the standard and contributed to the work being done by the OGF OCCI Working Group on the OCCI 1.2 standard. The following sections briefly describe the most prominent or widely used open-source OCCI implementations whilst also mentioning particular development challenges. This list is by no means all-encompassing; it is meant to provide a quick overview. The implementations listed here are also closely related to EGI and EGI Federated Cloud (del Castillo et al., 2015) due to existing affiliations of involved authors. Other prominent projects and implementations of OCCI not mentioned in this section include OCCIware (OCCIware Consortium, 2016), erocci (Parpaillon, 2016), and pyssf (Metsch, 2016). Readers are hereby encouraged to explore these as well.

2.1 The rOCCI Framework

The rOCCI framework, originally developed by GWDG, later adopted and now maintained by CESNET, was written to simplify implementation of the OCCI 1.1 protocol in Ruby and later provided the base for working client and server components giving OCCI support to multiple cloud platforms while ensuring interoperability with other existing implementations (Parák et al., 2014).

At the time of writing this paper, the rOCCI framework has two published components: `rOCCI-core` and `rOCCI-api`. These serve as the base for two end-user products: `rOCCI-cli` and `rOCCI-server`.

The initial server-side component provided basic functionality and served as a proof of concept when it was adopted by the EGI Federated Cloud Task Force and was chosen to act as the designated virtual machine management interface (Wallom et al., 2015). This led to further funding from the EGI-InSPIRE project, development of a full featured client and a new `rOCCI-server` suitable for production environment.

`rOCCI-core` is a central component of the framework. It implements classes representing entities defined by the OCCI standard, provides parsing and rendering capabilities to/from multiple message formats. Currently supported, as both input and output formats, are plain-text and JSON which is based on an early draft of OCCI 1.2 JSON rendering (not yet published). It also introduces various helper classes such as `Collection` or `Model`, simplifying the handling and rendering of complex OCCI messages, provides advanced logging and attribute validation facilities. Using Ruby's meta-programming techniques, the core is able to "extend itself" with new classes and definitions provided dynamically at run-time. This works well with OCCI's inherent extensibility.

`rOCCI-api` builds on top of `rOCCI-core` and implements support for transport protocols and corresponding authentication methods. It also provides an extended set of various helpers simplifying the use of `rOCCI-core` and targeting the development of client-side applications and tools. HTTP (Fielding and Gettys, 2014) is currently the only supported transport protocol.

`rOCCI-api` also implements a pluggable authentication mechanism capable of fall-backs. Every authentication plug-in can declare a set of alternatives to be used in case of failure. This concept is capable of masking differences between various cloud frameworks implementing OCCI using their own authentication schemes.

`rOCCI-cli`, built on top of `rOCCI-api`, serves as an end-user client providing a shell-based interface. It allows users to interact with OCCI-compliant interfaces and perform basic operations and actions. Similarly, `rOCCI-server`, built on top of `rOCCI-core`, provides a server-side implementation supporting multiple cloud management frameworks or public cloud providers as its back-ends, exposing their resources via an OCCI-compliant interface. It currently supports OpenNebula (The OpenNebula Project, 2016) and Amazon Web Services EC2 (Amazon Web Services, Inc., 2016), with plans to implement support for Microsoft Azure (Microsoft, 2016).

The most notable challenges encountered whilst implementing parts of the rOCCI framework were, in no particular order, the plain-text rendering, the lack of clear authentication guidelines, missing parts of the attribute model, and vague (side-)effect descriptions for the infrastructure extension. The following paragraphs will briefly address each of these issues in greater detail.

Implementing a plain-text-based rendering of a non-trivial protocol is always a challenge. The unstructured nature of a plain text requires customized regular expressions or state machines for extracting relevant information. It also prevents the developer from using existing well-tested parsers or serialization and de-serialization libraries. Writing customized parsers is always time-consuming and prone to errors, which are difficult to discover.

The lack of clear authentication guidelines is a feature of the OCCI standard. It simply points the reader to mechanisms appropriate (and standardized) for the given transport protocol, which helps to keep the standard extensible. However, it makes the implementation of working OCCI-enabled components difficult, especially in a heterogeneous environment where a certain level of interoperability is desirable. This forces developers to implement complex fallback or discovery mechanisms, with wildly varying levels of success.

Parts of the OCCI standard, especially the core specification, are rather abstract and the mechanisms described therein are difficult to implement. Developers are often left to their own devices, which leads to further discrepancies in interpretation between implementations. One such part is the attribute model, lacking detailed description and implementation guidelines.

Last but not least, effects or side-effects of particular operations or actions defined in various extensions of the OCCI standard often have vague descriptions. This makes aligning the behavior of implementations across multiple platforms very difficult. A unified behavior is a strong requirement for the user.

2.2 The jOCCI Framework

The jOCCI framework is a set of Java libraries implementing the OCCI standard. jOCCI currently consists of two client-side components, `jOCCI-core` and `jOCCI-api`, which together create a communication layer for OCCI clients and servers alike (Kimle et al., 2015).

`jOCCI-core` covers basic OCCI class hierarchy from both OCCI Core and OCCI Infrastructure and relations between them. 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, `jOCCI-core` also validates any OCCI request with respect to the declared OCCI model. This helps identify requests that would be rejected by server, even before they are sent.

`jOCCI-api` is a Java library implementing the transport layer functionality for rendered OCCI objects and queries. It is built on top of jOCCI-core and currently provides only HTTP transport functionality with a set of authentication methods and basic interfaces to simplify client-server communication.

The jOCCI library stack is currently used in the `jsaga-adaptor-jocci` (Rocca, 2016) project developed primarily for the Catania Science Gateway Framework (Fargetta, 2016). The aim of this project is to develop JSAGA (Reynaud and Schwarz, 2016) adaptor which will expose an interface for submitting grid jobs into automatically provisioned virtual machines in OCCI-compliant clouds. Another project utilizing jOCCI as the cloud-facing backend is the Karamel (Hakimzadeh, 2016) orchestration framework, heavily used in the bio-informatics community (Bessani et al., 2015). The so-called OCCI "launcher" adds support for the provisioning of on-demand virtual machines in OCCI-compliant clouds. The Karamel OCCI launcher is currently being developed and tested at CESNET.

As mentioned before, jOCCI is written in the Java programming language. This language was selected because of its popularity and demand by the community. Since Java is a strongly typed language, some of the concepts of the OCCI standard (e.g., mixins) were somewhat challenging to implement. Nevertheless, jOCCI's API was designed to meet the requirements of the OCCI standard whilst keeping Java's best practices in mind. For details, see Figure 1.

jOCCI currently implements version 1.1 of the OCCI standard. During the development of the libraries the authors encountered multiple caveats in this version of the standard. One of the problems is that the specification does not clearly state which attributes are internal and which should be available in rendering. Another problem comes from the plain text rendering, which is lacking the expressive power of other standard formats such as XML or JSON, making it difficult to render complex data structures.

Both `jOCCI-core` and `jOCCI-api` are distributed via Maven Central Repository utilizing well-known dependency management practices common in the Java developers community.

2.3 OpenStack OCCI Interface

`ooi` (López García et al., 2016) is an implementation of OCCI for the OpenStack Compute project, written entirely in Python so as to make exhaustive usage and profit from the already available OpenStack modules, such as authentication. `ooi` was designed to be easily integrated with the OpenStack core components, but with the aim of being independent from any OpenStack version or release.

The main motivation for this recent development stems from the need to overcome key architectural design issues found in the previous OCCI implementation, OCCI-OS (Metsch et al., 2016), namely the fact of using the internal OpenStack APIs directly. In contrast with the public APIs, the private ones are not versioned and are subject to change at any time in the development cycle, even between minor releases. Instead, `ooi` interacts with OpenStack lever-

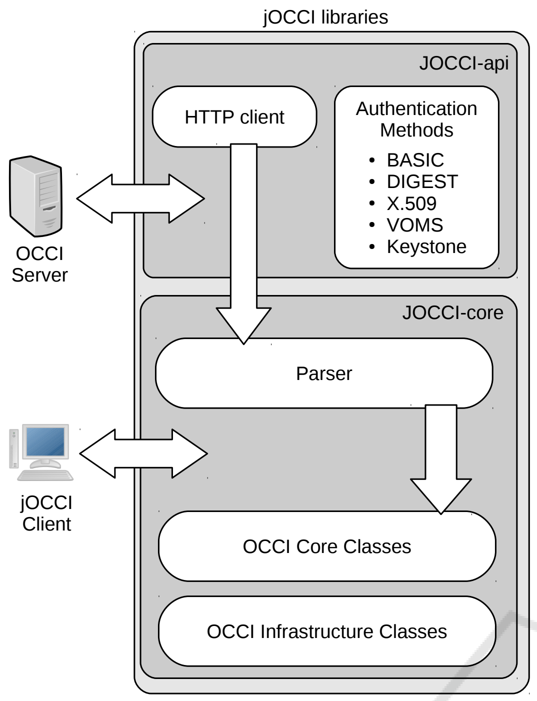

Figure 1: `jOCCI-*` architecture overview.

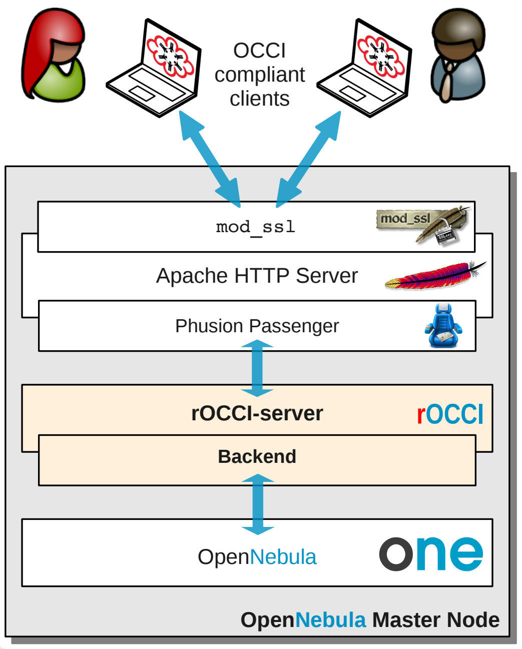

Figure 2: `rOCCI-server` in a typical setup with the OpenNebula Cloud Manager.

aging its public APIs to process any incoming OCCI request, translating it forth and rendering it back to get a proper and valid OCCI response. Unlike OCCI-OS, `ooi` has been designed as a WSGI middleware embedded in the OpenStack pipeline, located prior to the public API request processing and, accordingly, appearing as the first step once the API returns the response object.

At the time of writing this paper, `ooi` supports the OpenStack API version 2.1, but additionally it can be deployed on top of the previous and backwards compatible API version 2.0. Moreover, `ooi` allows the co-existence of isolated environments in terms of multiple OCCI endpoints mapped to different OpenStack API versions in the same installation, making it possible for providers to deploy several OCCI versions in different endpoints using the same deployment. In this regard, the current version of `ooi` implements version 1.1 of the OCCI standard, but the aforementioned design would make it possible to deploy several OCCI versions using the same installation.

The most relevant challenge faced when developing `ooi` was the implementation of text rendering for such a complex protocol. The parsing of plain-text structures representing more complex structures has been also identified as one of the major drawbacks in the other OCCI implementations, such as rOCCI (Section 2.1) and jOCCI (Section 2.2). In this regard, the adoption of JSON rendering in version 1.2 of the standard would be a step forward when compared with OCCI 1.1.

Since its release, `ooi` has been adopted in the EGI Federated Cloud as the reference implementation for the OpenStack providers.

2.4 OpenNebula OCCI Interface

As mentioned in Section 2.1, `rOCCI-server` acts as an OCCI-compliant interface for OpenNebula. It has been designed as a stateless proxy translating OCCI messages to native API calls for OpenNebula. Its main purpose is to hide platform-specific behavior from the user and create the illusion of a seamless OCCI-compliant service across multiple heterogeneous cloud platforms. As `rOCCI-server` is built on top of `rOCCI-core`, discussed in greater detail in Section 2.1, further description of its OCCI-related internals is omitted here.

For detailed description of `rOCCI-server`'s architecture and deployment, see Figure 2.

2.5 Synnefo OCCI Interface

The Synnefo OCCI interface (Athanasia Asiki, 2014) acts as an API middleware between the OCCI protocol and the Synnefo API (synnefo.org, 2015). Synnefo cloud software (synnefo.org, 2014) utilizes Ganeti (Guido Trotter, 2013) as a low-level virtualization layer to provide compute and storage cloud over an extended OpenStack API. The ˜okeanos IaaS (Vangelis Koukis, 2013), which provides compute and storage resources to the Greek and European academic communities, is the largest Synnefo deployment.

Synnefo (open source IaaS software) and ˜okeanos (IaaS service powered by Synnefo) are maintained and provided by the Greek Research and Technology Network (GRNET), which is the National Research and Education Network (NREN) provider of Greece. The principal role of GRNET is to operate the Greek Academic network, connect it with global academic communities and institutions, and provide them with cutting edge IT services and technology. GRNET is a key national level facilitator in the fields of distributed and large-scale research infrastructures including Grid, Cloud and HPC. It coordinates the Greek National Grid Initiative – HellasGrid and is a member of the EGI pan-european grid infrastructure.

The Synnefo OCCI interface features a distributed design of separate components (e.g., `snf-occi` and `astakos-vo-proxy`) which are connected via RESTful APIs. The rationale behind the aforementioned design choice is to ensure adaptability to the evolution of both OCCI and Synnefo, as well as robustness and security through the deployment on isolated nodes.

The main component of the interface is called `snf-occi`, a service that maps OCCI v1.1 requests to OpenStack/Synnefo. It is designed to run as a stand-alone service, connected to Synnefo through its RESTful API. Incoming (OCCI) requests are validated syntactically and corresponding users are authenticated. Synnefo credentials are retrieved through an API call to the `astakos-vo-proxy` component. The credentials are attached as headers to requests to the Synnefo API. Results are then reverse-mapped to be OCCI-compliant and returned as the response to the initial request.

`astakos-vo-proxy` acts as a user mapping agent with user creation and modification capabilities. Astakos is the authentication and policy (e.g., user quota) enforcement component of Synnefo. To facilitate the mapping between OCCI and Synnefo users, the proxy maintains an LDAP directory with the minimum user information needed for the mapping. Possible changes in the status of the supported user pool are reflected by frequently updating the directory with the assistance of a human operator.

Authenticating OCCI users who do not exist in the Synnefo user base was one of the most intriguing challenges while developing the interface. To tackle this issue, `astakos-vo-proxy` is equipped with the ability to create and modify Synnefo users and their quota policies. Every time a new but valid OCCI user attempts to access a Synnefo cloud through the OCCI interface, the proxy will create a new Synnefo user. On the other hand, if the user exists, the corresponding information is retrieved from the directory.

The mapping of OCCI users to Synnefo users constitutes a security challenge, because it allows users outside of the scope of a standard Synnefo deployment to be created on demand and also because it maintains a directory of sensitive user information and credentials. To protect `astakos-vo-proxy`, it must be deployed in a trusted and isolated environment.

A stable version of the `snf-occi` and the `astakos-vo-proxy` components are deployed for the ˜okeanos IaaS. Both components are deployed on separate virtual nodes powered by the said IaaS.

3 ISSUES

This section collects input from Section 2, finds language-independent commonalities and draws conclusions on the usability of OCCI 1.1 in real-world implementations. It aims at identifying the most severe obstacles preventing further adoption of the OCCI standard among developers and service providers.

Issues outlined below were selected from Section 2 based on the following criteria:

- impact on future standard development `(1)`
- number of occurrences `(2)`
- subjective significance `(3)`

Formal Documents

The overall readability of documents formalizing the standard is a very important factor, especially in early stages of its adoption. In this area, a number of developers expressed concern. Specifically, the more abstract parts should be explained in greater detail, with practical examples if applicable.

In many cases, a single word has two or three different meanings depending on the current context. This should be avoided wherever possible by introducing new terminology or, at least, carefully aligned across all published documents to minimize inconsistencies, especially with regard to third-party extensions.

Design

With regard to the minimalistic and extension-based design of OCCI, no major issues were reported. There are certain interoperability challenges specific to this particular design; however, no explicit objections were raised by developers.

Core

In the core specification, attribute description and discovery was the most commonly reported problem. In OCCI 1.1, attributes are not properly specified and defined in the context of the OCCI (meta)model. This leads to implementation-specific solutions and differences in behavior.

In the next revision of the standard, attributes should be clearly defined in the OCCI (meta)model, including attribute properties and validation mechanisms.

Extension

When it comes to extensions, most developers have experience only with the Infrastructure extension. In this extension, vague descriptions of various operations and actions on resource instances were the most frequently reported issues. When the specification does not provide a clear description of effects and side-effects, these are left to implementation/developer-specific interpretation. It leads to issues with interoperability.

In the next revision of the standard, additional explanations and descriptions should be added wherever possible. However, it is understood that strict guidelines in this area would limit further proliferation and adoption of the standard due to platform-specific limitations.

Transport

No issues were reported for transport-layer specifications, currently represented only by the HTTP specification (Metsch and Edmonds, 2011a). Improvements were suggested for the parts dealing with authentication and authorization mechanisms; however, these are designated as “out-of-scope” by the standard.

Rendering

All reported issues were targeting the plain-text rendering (Metsch and Edmonds, 2011a), being the only currently published rendering specification. Issues ranged from relatively minor (parsing difficulties and performance) to distinctly major ones (inability to represent complex data types or advertise/discover necessary endpoint information).

The overall consensus on these issues is the need for new rendering formats with accompanying extensions/specifications outlining their use, including extensive examples.

4 TOWARDS OCCI 1.2

This section aims to address recent advancements towards the final OCCI 1.2 specification with regard to issues outlined in Section 3. The following subsections focus on the aforementioned issues one by one. Work on the OCCI 1.2 specification is performed by the OGF OCCI Working Group; this paper provides only an overview. The following list of changes is by no means complete.

Formal Documents

The OCCI documents underwent a significant refactoring. The OCCI 1.2 standard consists of seven separate documents covering core (Nyrén et al., 2016b), infrastructure (Edmonds et al., 2016), HTTP protocol (Nyrén et al., 2016a), text rendering (Edmonds and Metsch, 2016), JSON rendering (Nyrén et al., 2016c), PaaS (Platform as a Service) (Metsch and Mohamed, 2016), and SLAs (Service-Level Agreements) (Katsaros, 2016). With the addition of one profile document (Drescher et al., 2015) attempting to standardize available compute resource sizes. This should greatly improve readability and decouple unrelated extension specifications. At the time of writing the paper, the official documents are not yet published.

Design

No significant changes were made to the overall design of the OCCI protocol in order to maintain good backward compatibility with OCCI 1.1.

Core

Attribute description in the OCCI (meta)model has been appropriately updated and OCCI 1.2 clearly defines how to represent model attributes, including attribute properties. This change is based on existing implementations and should hence cover all required use cases such as attribute discovery or attribute value validation.

Extension

Information regarding the use of OS and Resource template mixins has been updated and extended to give deeper insight into their intended purpose. Inconsistencies in other parts of the infrastructure extension have been corrected (removed useless attributes and unusable actions, adjusted vague wording wherever possible). However, key parts of the document outlining effects and side-effects of various actions remain unchanged to avoid overly restricting future implementations.

Transport

No significant changes were made to the transport-layer specification, aside from major document refactoring which separated HTTP protocol from the plain text rendering specification and minor clarification regarding the use of HTTP status codes to relay action results.

Rendering

To maintain backward compatibility with OCCI 1.2, no significant changes could be made to the plain text rendering of OCCI. However, to address issues raised by a number of developers, a new mandatory rendering was introduced as an extension – the JSON rendering. Using JSON (Crockford, 2006), including an example JSON Schema (Galiegue et al., 2013) for OCCI messages, should greatly simplify implementation and provide much needed reliability.

5 CONCLUSION

OCCI is a well-matured and well-accepted protocol, with a wide user base – mainly among academic users – and a number of independent implementations. Given that its authors were always striving for flexibility, discrepancies naturally had to occur in OCCI-compliant tools, especially in the early versions. However, the OCCI community was able to overcome that: firstly by being able to find common interpretation in problem areas, and secondly by being able to keep the standard evolving, answering not only to new needs but also to old aches of interoperable clouds.

ACKNOWLEDGEMENTS

In no particular order, authors would like to acknowledge their home institutions: CESNET, Masaryk University, Instituto de Física de Cantabria, GRNET, and Universitat Autònoma de Barcelona. They would also like to thank their colleagues from EGI and members of the EGI Federated Cloud. The credit for work on the OCCI standard and its continuous evolution goes to Open Grid Forum and OGF's OCCI Working Group.

This work is co-funded by the EGI-Engage project (Horizon 2020) under Grant number 654142.

REFERENCES

Amazon Web Services, Inc. (2016). Amazon Web Services - Elastic Compute Cloud. [Online] https://aws.amazon.com/ec2/. Accessed: March 10, 2016.

Athanasia Asiki, C. (2014). Synnefo OCCI Interface. [Online] https://code.grnet.gr/projects/snf-occi. Accessed: March 10, 2016.

Bessani, A., Brandt, J., Bux, M., Cogo, V., Dimitrova, L., Dowling, J., Gholami, A., Hakimzadeh, K., Hummel, M., Ismail, M., et al. (2015). Biobankcloud: a platform for the secure storage, sharing, and processing of large biomedical data sets. *the First International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH 2015)*.

Crockford, D. (2006). The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational).

del Castillo, E. F., Scardaci, D., and Álvaro Lopéz García (2015). The egi federated cloud e-infrastructure. *Procedia Computer Sceince*, (68):196–205.

Drescher, M., Parák, B., and Wallom, D. (2015). OCCI Compute Resource Templates Profile rev. 2. [Online] https://goo.gl/puR6JG.

Edmonds, A. and Metsch, T. (2016). Open Cloud Computing Interface – Text Rendering rev. 1.2. [Online] https://goo.gl/puR6JG.

Edmonds, A., Metsch, T., and Parák, B. (2016). Open Cloud Computing Interface – Infrastructure rev. 1.2. [Online] https://goo.gl/puR6JG.

Fargetta, M. (2016). Catania Science Gateway Framework. [Online] http://www.catania-science-gateways.it/. Accessed: March 10, 2016.

Fielding, R. and Gettys, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230.

Galiegue, F., Zyp, K., and Court, G. (2013). Json schema: core definitions and terminology. draft-zyp-json-schema-04.

Guido Trotter, T. (2013). Ganeti: Cluster Virtualization Manager. *USENIX;login*, (3).

Hakimzadeh, K. (2016). karamel. [Online] https://github.com/kamalhakim/karamel. Accessed: March 10, 2016.

Katsaros, G. (2016). Open Cloud Computing Interface – Service Level Argeements rev. 1.2. [Online] https://goo.gl/puR6JG.

Kimle, M., Parák, B., and Šustr, Z. (2015). jOCCI – general-purpose OCCI client library in java. In *ISGC15, The International Symposium on Grids and Clouds 2015*. PoS.

López García, Á., Fernández del Castillo, E., and Orviz Fernández, P. (2016). ooi: OpenStack OCCI interface. *SoftwareX*, (xxxx):1–6.

Metsch, T. (2016). Service sharing facility. [Online] https://github.com/tmetsch/pyssf. Accessed: March 10, 2016.

Metsch, T. and Edmonds, A. (2011a). Open Cloud Computing Interface – HTTP Rendering. GFD-P-R.185.

Metsch, T. and Edmonds, A. (2011b). Open Cloud Computing Interface – Infrastructure. GFD-P-R.184.

Metsch, T., Edmonds, A., and López García, Á. (2016). OCCI Interface for OpenStack. [Online] https://github.com/stackforge/occi-os. Accessed: March 10, 2016.

Metsch, T. and Mohamed, M. (2016). Open Cloud Computing Interface – Platform rev. 1.2. [Online] https://goo.gl/puR6JG.

Microsoft (2016). Microsoft Azure: Cloud Computing Platform. [Online] https://azure.microsoft.com/. Accessed: March 10, 2016.

Nyrén, R., Edmonds, A., Metsch, T., and Parák, B. (2016a). Open Cloud Computing Interface – HTTP Protocol rev. 1.2. [Online] https://goo.gl/puR6JG.

Nyrén, R., Edmonds, A., Papaspyrou, A., and Metsch, T. (2011). Open Cloud Computing Interface – Core. GFD-P-R.183.

Nyrén, R., Edmonds, A., Papaspyrou, A., Metsch, T., and Parák, B. (2016b). Open Cloud Computing Interface – Core rev. 1.2. [Online] https://goo.gl/puR6JG.

Nyrén, R., Feldhaus, F., Parák, B., and Šustr, Z. (2016c). Open Cloud Computing Interface – JSON Rendering rev. 1.2. [Online] https://goo.gl/puR6JG.

OCCIware Consortium (2016). Occiware project. [Online] http://goo.gl/M1rZKv. Accessed: March 10, 2016.

Parák, B., Šustr, Z., Feldhaus, F., Kasprzak, P., and Srba, M. (2014). The rOCCI project – providing cloud interoperability with OCCI 1.1. In *ISGC14, The International Symposium on Grids and Clouds 2014*. PoS.

Parpaillon, J. (2016). Occi compliant rest framework. [Online] https://github.com/erocci/erocci. Accessed: March 10, 2016.

Reynaud, S. and Schwarz, L. (2016). JSAGA. [Online] http://software.in2p3.fr/jsaga/dev/index.html. Accessed: March 10, 2016.

Rocca, G. L. (2016). jsaga-adaptor-jocci. [Online] https://github.com/csgf/jsaga-adaptor-jocci. Accessed: March 10, 2016.

synnefo.org (2014). Synnefo White Paper. [Online] https://goo.gl/LDvTsf. Accessed: March 10, 2016.

synnefo.org (2015). Synnefo API. [Online] https://www.synnefo.org/docs/synnefo/latest/api-guide.html. Accessed: March 10, 2016.

The OpenNebula Project (2016). OpenNebula Cloud Management Framework. [Online] http://www.opennebula.org/. Accessed: March 10, 2016.

Vangelis Koukis, Constantinos Venetsanopoulos, N. (2013). okeanos: Building a Cloud, Cluster by Cluster. *IEEE Internet Computing*, (3):67–71.

Wallom, D., Turilli, M., Drescher, M., Scardaci, D., and Newhouse, S. (2015). Federating infrastructure as a service cloud computing systems to create a uniform e-infrastructure for research. In *IEEE 11th International Conference on e-Science, 2015*. IEEE.