Available online at www.sciencedirect.com
ScienceDirect
SoftwareX 5 (2016) 6–11
SoftwareX
www.elsevier.com/locate/softx

ooi: OpenStack OCCI interface

Álvaro López García*, Enol Fernández del Castillo, Pablo Orviz Fernández

Institute of Physics of Cantabria, Spanish National Research Council — IFCA (CSIC—UC), Avda. los Castros s/n, 39005 Santander, Spain

Received 27 August 2015; received in revised form 27 November 2015; accepted 15 January 2016

Abstract

In this document we present an implementation of the Open Grid Forum's Open Cloud Computing Interface (OCCI) for OpenStack, namely `ooi` (Openstack occi interface, 2015) [1]. OCCI is an open standard for management tasks over cloud resources, focused on interoperability, portability and integration. `ooi` aims to implement this open interface for the OpenStack cloud middleware, promoting interoperability with other OCCI-enabled cloud management frameworks and infrastructures. `ooi` focuses on being non-invasive with a vanilla OpenStack installation, not tied to a particular OpenStack release version.

Keywords: Cloud; OCCI; Standards; Interface

Code metadata

Current code version	0.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-15-00055
Legal Code License	Apache License, Version 2.0
Code versioning system used	git
Software code languages, tools, and services used	Python(>= 2.7)
Compilation requirements, operating environments	pbr, oslo.log, oslo.config, routes, OpenStack installation
If available Link to developer documentation/manual	https://ooi.readthedocs.org/
Support email for questions	ooi@lists.launchpad.net

1. Motivation and significance

The Open Grid Forum (OGF) has proposed the Open Cloud Computing Interface (OCCI) [2] as an open standard defining a RESTful API for managing cloud resources, developed as a joint effort between industry and academia.

The OCCI specification is being delivered as a set of complementary documents divided into three categories: the OCCI Core, the OCCI Renderings and the OCCI Extensions. At the time of writing this document the current version of the standard is OCCI 1.1, with OCCI 1.2 version being currently under development.

OCCI Core This is a single document [3] defining the OCCI Core abstract model. This model can be interacted with the renderings and is expanded by the OCCI extensions.

OCCI Renderings The OCCI Renderings describe how the OCCI Core model should be rendered. The current OCCI HTTP Rendering specification [4] defines how to interact with the OCCI core model and its extensions over a HTTP protocol based RESTful API. Multiple and different renderings may interact with the same instances of the OCCI Core protocol, thus not being limited to use a concrete rendering.

* Corresponding author.
E-mail addresses: aloga@ifca.unican.es (Á. López García), enolfc@ifca.unican.es (E. Fernández del Castillo), orviz@ifca.unican.es (P. Orviz Fernández).

http://dx.doi.org/10.1016/j.softx.2016.01.001

OCCI Extensions These specifications describe additions to the OCCI core model. The OCCI Infrastructure specification [5] contains the extension for the IaaS domain, defining the needed resource types, attributes and actions that can be taken on each resource type.

Several Mixin extensions have been developed so as to add additional functionality. Namely, the most relevant ones are the following:

Contextualization Extension Contextualization is the process of installing, configuring and preparing software upon boot time on a pre-defined virtual machine image. This Mixin extension allows to pass some data to the instance [6] that can be further fetched from inside the virtual machine by a software such as cloud-init [7] or Flamingo [8].

Key Pair Extension This Mixin extension allows users to inject a SSH public key for the authenticated access to the provisioned VM [6,9].

OCCI has been one of the first standards in the cloud ecosystem, providing the foundations for basic management tasks in Infrastructure as a Service (IaaS) providers and it can be easily extended easily so as to provide additional functionality. OCCI is a standard relevant for both cloud users and cloud providers as a way to provide an interoperable infrastructure, removing any kind of vendor lock-in.

Currently some OCCI implementations already exist for several cloud vendors or in the form of general frameworks that can be extended with several backends. In order to get an OCCI-enabled OpenStack [10] deployment, we only considered two candidates: rOCCI and OCCI-OS. Other implementations exist, but either they do not have recent activity in their codebase or they are too general frameworks that needed a lot of integration efforts (for instance for the authentication and authorization parts).

rOCCI [11] is one of the most notable projects implementing OCCI. It is a framework written in Ruby that aims to improve interoperability in the cloud by delivering an OCCI implementation that can be used by both at the server and at the client side. The rOCCI-server component makes possible to add an OCCI interface to some existing cloud stacks and vendors via one of the existing configurable backends, such as OpenNebula [12], Apache CloudStack [13], VMware [14] and Amazon EC2 [15]. It stands as a standalone server (rOCCI-server) that proxies the requests to the underlying cloud management framework. The rOCCI-cli on the other hand is the client component of rOCCI, making possible to interact with any OCCI-enabled framework.

OCCI-OS [16] is an implementation of OCCI for OpenStack, leveraging the Python Service Sharing Facility (pyssf) [17]. It consists on a new WSGI application that uses the internal OpenStack APIs.

rOCCI-server could be adapted to be used over an OpenStack installation, but the fact of being written in Ruby is an obstacle for reusing the existing OpenStack modules (e.g. authentication) already available.

On the other hand, OCCI-OS' WSGI application speaks directly to the OpenStack internal APIs. These APIs are not versioned and can be subject to change at any point in the development, leading to incompatibilities between the OCCI modules and the different OpenStack versions. As a result, the need of several OCCI-OS releases, each one aligned with its corresponding OpenStack API version, is a must. Changes in the internal OpenStack APIs happen even between minor releases, making impractical to update the code for each new version. Making OCCI-OS use the public APIs instead involves a complete refactorization of its codebase, as it leverages all the internal backends to accomplish the desired actions.

As an aim to overcome these architectural issues, we present in this paper `ooi` [1], a Python-based application designed to be easily integrated with the OpenStack core components.

2. Software description

2.1. Foreword on WSGI

The Python WSGI standard [18] proposes an interface between web servers and Python web applications so that it is possible for an application to handle HTTP requests using Python code. Among other things, it defines the WSGI application, server and middleware.

- The WSGI application object receives a representation of the HTTP request, processes it and returns a response that will be eventually sent back to the client.
- The WSGI server invokes the application for each request that is targeted to it. Therefore, an application receives the request from a server.
- The WSGI middleware receives a WSGI request, performs some logic on it, and sends it to the next WSGI middleware or application. Therefore, the WSGI middleware is seen as an application by a WSGI server, and as a server by a WSGI application.

It is then possible to chain several WSGI middlewares together, each one adding some additional functionality before actually passing the request to the final application. This appears as an analogy with *pipes* on UNIX systems, thus often using the term *pipeline* to refer to this chain of WSGI middlewares and applications.

Following this structure, OpenStack native API is a WSGI application that leverages several of such middlewares that perform additional functionalities like authentication (against the OpenStack Identity Component), and rate and size limiting, just to cite some.

In this context, instead of implementing `ooi` as a WSGI application, it has been developed as a WSGI middleware that proxies the OCCI requests and translates it to an appropriate OpenStack request. This is a key aspect of `ooi`'s architecture design that, unlike other solutions (OCCI-OS [16]) does not

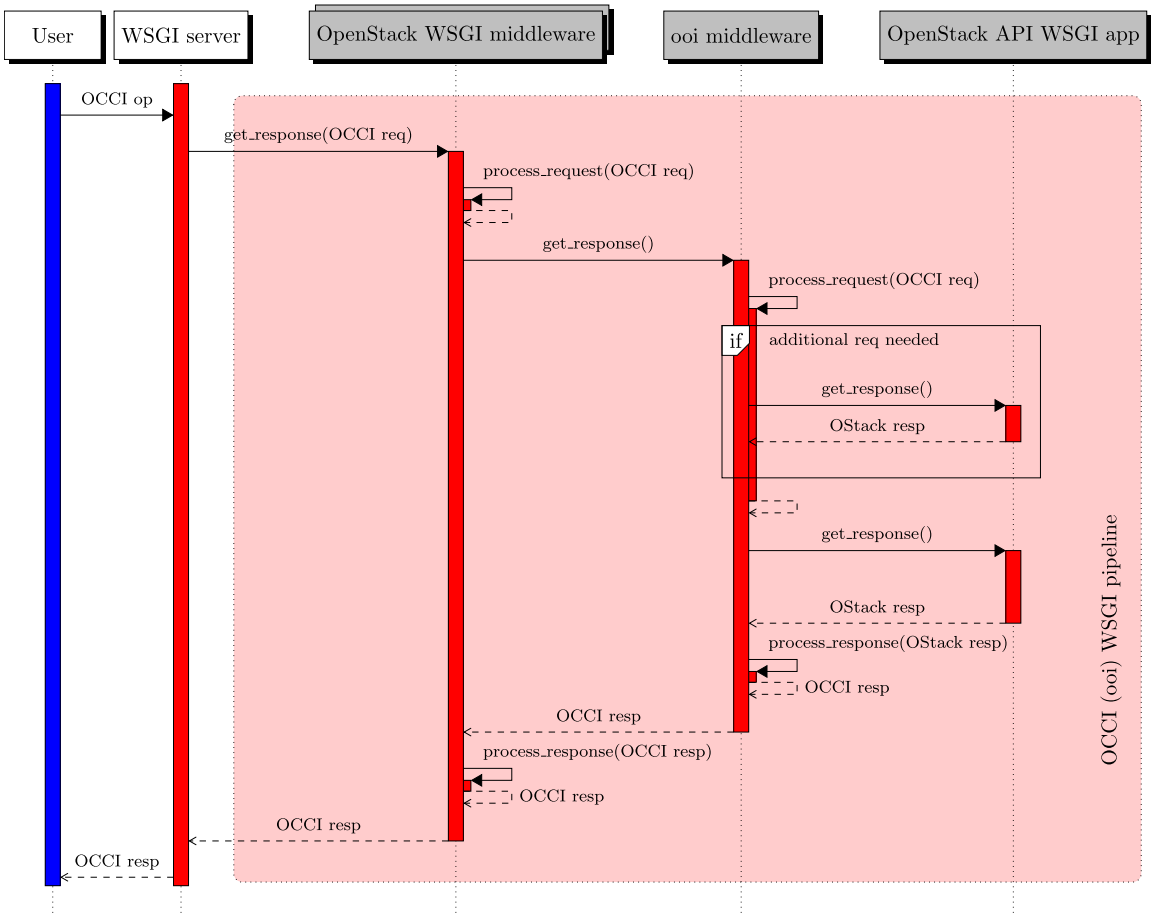

Fig. 1. `ooi` processing pipeline. This figure illustrates the sequence diagram for processing an OCCI request. The red shaded area represents the WSGI pipeline, whose components are depicted with grayed boxes. Solid arrows represent operations or method calls, dashed arrows represent data types, *OCCI op* is a request for an OCCI operation, *OCCI req* represents an OCCI request type, *OStack resp* is an OpenStack response, *OCCI resp* is an OCCI response, *OpenStack WSGI middleware* are the preceding and unmodified OpenStack WSGI default middlewares that are present in the pipeline. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

appear as a standalone WSGI application that calls OpenStack internal interfaces but rather makes use if its public API.

`ooi`'s workflow is shown in Fig. 1. The red shaded area represents the OCCI WSGI pipeline, whose components are depicted as gray boxes. As it is shown in the Figure, each of the WSGI middleware process the request and perform some operation with it (for example, authentication), then they call the next application or middleware in the pipeline, until the request gets downs to the final OpenStack API WSGI application. Then, the application will return a response, that will be processed back in reverse order by each of the WSGI middlewares until it gets up to the WSGI server.

Therefore, whenever a OCCI request arrives to the `ooi` middleware, this request is processed and translated into a new equivalent OpenStack request, based on its public API. If further information is needed so as to build the request, it is done transparently to the user. Whenever this transformation finishes, `ooi` passes down the corresponding OpenStack request to the OpenStack API WSGI application – the last step in the pipeline – and an OpenStack response is obtained. This response is processed again by the OCCI middleware, so that it is rendered back as a proper and valid OCCI response, and it continues its path upstream to the WSGI server.

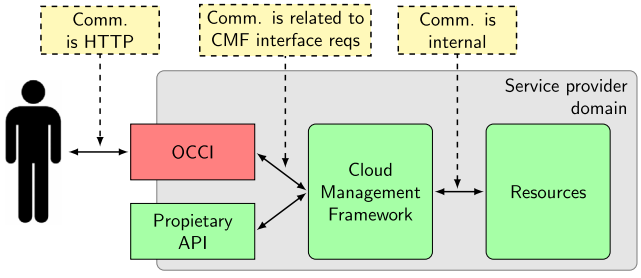

Fig. 2. Proposed OCCI's place in a provider's architecture according to the standard. Boxes in yellow explain the type of communication being made, green depicts the Cloud Management Framework components, red the OCCI interface. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

2.2. Interacting with OpenStack

The OCCI standard defines the API as a boundary interface that acts as a frontend to the internal management APIs, as shown in Fig. 2.

To interact with OpenStack, `ooi` leverages its public API interfaces [19] rather than using the private API (Fig. 3). This architecture decision is motivated by the fact that OpenStack

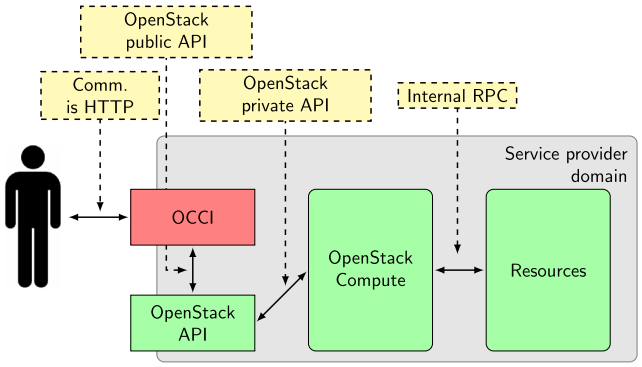

Fig. 3. OCCI place in a provider's infrastructure, following `ooi`'s architecture. Instead of using the private APIs, OCCI requests are translated to native OpenStack requests. Boxes in yellow explain the type of communication being made, green depicts the OpenStack components, red the OCCI interface. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

public API is versioned, whereas its private interfaces are not; hence there is no contract to maintain its signature between OpenStack releases.

This fact causes that any application using the private, internal interfaces may need to be adapted throughout OpenStack releases. On the other hand, changes in the public REST API are versioned (each change increases the minor version of the API), and the same version is supported across several releases. A given version of OpenStack public API is not subject to functionality or backwards incompatible changes, since that kind of changes will increase the version number.

The development work that involves supporting new major releases of OpenStack public API is alleviated by `ooi`'s modular architecture, making possible to plug additional modules without modifying substantial parts of the code. Moreover, several OCCI endpoints, supporting different OpenStack API versions, can co-exist in a single `ooi` installation allowing isolated environments to be used for different purposes. Thus e.g testing experimental API features can live together with the production endpoint without risks.

Currently, the supported OpenStack version is v2.1 [20]. However, it is possible to deploy `ooi` on top of the previous v2.0 API, since v2.1 is backwards compatible with the addition of strong API validation.

2.3. *ooi functionality*

`ooi` implements the OCCI 1.1 standard as described in Section 1. It implements the OCCI Core specification [3], the OCCI infrastructure extension [5] as well as the OCCI HTTP rendering [4]. Additionally, two extra extensions were implemented: the contextualization and SSH credentials.

During the development stages of `ooi` we have focused not only on following the standard, but also on remaining compatible with any other existing OCCI implementations. A comparison of the different OCCI implementations and the operations that can be performed in each of them is summarized in Table 1.

It is worth notice that OCCI does not mandate that all operations are actually supported by the respective backend (in this case OpenStack), therefore operations marked as N/A or marked as not implemented in Table 1 render the correct result as specified in the OCCI standard (that is, the HTTP 501 Not Implemented error code).

Table 1
OCCI feature comparison of the several implementations. The lack of features in rOCCI is due to the backend, not rOCCI itself, since it refers to the OpenNebula backend as there is no OpenStack backend available. N: not implemented or available, Y: implemented, P: partially implemented, N/A: not applicable (backend does not support it).

Query Interface	rOCCI	OCCI-OS	`ooi`
retrieve model	Y	Y	Y
filter	N	N	N
Infrastructure extension: compute			
query	Y	Y	Y
query and filter	N	N	N
create	Y	Y	Y
delete	Y	Y	Y
actions: start	Y	Y	Y
actions: stop	Y	Y	Y
actions: restart	Y	Y	Y
actions: suspend	Y	Y	Y
Infrastructure extension: network			
query	Y	N	Y
query and filter	N	N	N
create	P	N	Y
delete	Y	N	Y
attach to compute	Y	Y	Y
attach to compute	Y	Y	Y
detach from compute	Y	Y	Y
actions: up	N	N/A	N/A
actions: down	N	N/A	N/A
Infrastructure extension: storage			
query	Y	Y	Y
query and filter	N	N	N
create	Y	Y	Y
delete	Y	Y	Y
attach to compute	Y	Y	Y
detach from compute	Y	Y	Y
actions: online	Y	N/A	N/A
actions: offline	Y	N/A	N/A
actions: backup	Y	N/A	N/A
actions: snapshot	N	N	N
actions: resize	N	N/A	N/A
Contextualization extension			
contextualize compute	Y	Y	Y
SSH Key extension			
as an argument	Y	Y	Y
existing key	N	N	Y

2.4. *Performance comparison*

Even though it is not the original purpose of this new implementation, we found interesting to compare `ooi` performance against the existing OpenStack implementation – OCCI-OS

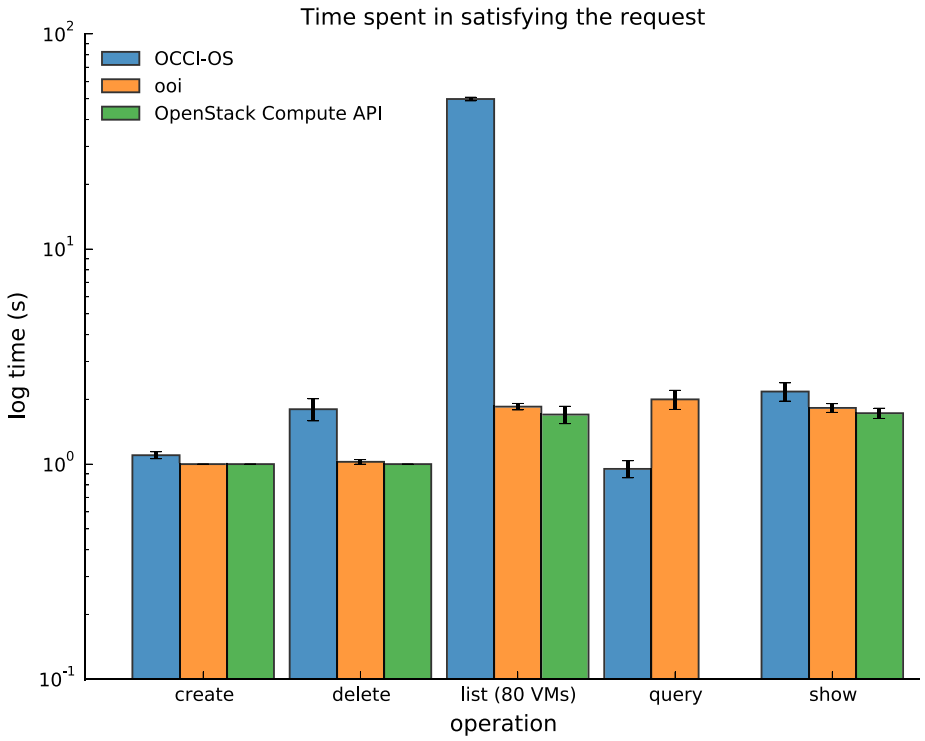

Fig. 4. Performance comparison between both OpenStack implementations, using a logarithmic scale in the Y-axis. The listing of the running instances has been made against an infrastructure running 80 instances. Standard error is represented in the error bars. Note that the *query* operation for OpenStack is not applicable, as it is OCCI specific and there is no equivalent in OpenStack.

– so as to ensure that our implementation does not impede the overall performance of the system.

For an accurate comparison to be made, we have deployed both `ooi` and OCCI-OS over the same OpenStack Compute controller and performed some basic operations using `ooi`, OCCI-OS and the native OpenStack API. The underlying server consists of an 8 core (Intel® Xeon® CPU E5-2640 2.00 GHz) virtual machine, with 16 GB of RAM. In order to eliminate any potential overhead introduced by a client tool (such as authentication or data verification), the operations have been made directly to the API using the corresponding HTTP method (i.e. GET, POST and DELETE in this case).

As it can be seen in Fig. 4, the results for the most common operations are similar, with the exception of listing a large number of VMs (for this comparison we have deployed 80 VMs). It is worth notice that there is no *query* operation or any equivalent in the native OpenStack Compute API, therefore it is not possible to show the results for such operation.

3. Conclusions

Standards in the Cloud cannot evolve without a rich ecosystem of available implementations. As we have stated in Section 1, the rOCCI framework has provided great OCCI support for several Open Source cloud management frameworks, but it lacks an OpenStack backend.

In this context OCCI-OS is an existing implementation for OpenStack, but it leverages its internal and private interfaces. As we have stressed in this document, it requires to be updated each time a new OpenStack version is released. This could lead to troublesome situations, as resource providers rely heavily on the availability of an updated and compatible OCCI interface before performing any OpenStack deployment upgrade.

Unlike OCCI-OS, `ooi` makes use of the public, versioned REST API (as explained in Section 2) to allow smooth, modification-free transitions across OpenStack releases. Resource providers can upgrade their infrastructure to the next release, with no `ooi` relevant compatibility concerns.

OCCI is the reference standard for some federated cloud infrastructures, such as the EGI Federated Cloud [21]. In such federated infrastructures, having a stable implementation of the OCCI interface for all of the Cloud Management Frameworks used – such as OpenStack – is a must. The implementation of yet another module providing OCCI support for OpenStack has the risk of non being adopted, as cloud providers may be reluctant to deploy another new component. However, we expect that `ooi` is adopted by cloud federations as it tries to address some of the shortcomings that resource providers have faced in the operation of deployment of OCCI enabled OpenStack clouds with the current available tools.

Acknowledgments

The authors want to acknowledge the support of the EGI-Engage (grant number 654142) and INDIGO-Datacloud (grant number 653549) projects, funded by the European Commission's Horizon 2020 Framework Programme.

References

[1] Openstack occi interface (ooi), 2015. URL https://launchpad.net/ooi.

[2] Open Grid Forum (OGF), OCCI Working Group, 2015. URL https://www.ogf.org.

[3] Nyrén R, Metsch T, Edmonds A, Papaspyrou A. Open cloud computing interface–core, Tech. rep., Open Grid Forum, 2010.

[4] Metsch T, Edmonds A. Open cloud computing interface-RESTful HTTP rendering, Tech. rep., Open Grid Forum, 2011.

[5] Metsch T, Edmonds A. Open cloud computing interface-infrastructure, Tech. rep., Open Grid Forum, 2010.

[6] Fernandez E. Occi contextualization extension, 2015. URL https://wiki.egi.eu/wiki/Fedcloud-tf:WorkGroups:Contextualisation#Contextualization.

[7] cloud-init, 2015. URL https://launchpad.net/cloud-init.

[8] Flamingo, 2015. URL https://github.com/tmrts/flamingo.

[9] F.-W. project, Occi key pair extension, 2015. URL https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/DCRM_OCCI_Open_RESTful_API_Specification#Key_Pair_Extension.

[10] OpenStack Foundation, OpenStack, 2015. URL http://openstack.org.

[11] Parak B, Sustr Z, Feldhaus F, Kasprzak P, Srba M. The rOCCI Project Providing Cloud Interoperability with OCCI 1.1; 2014. p. 1–15.

[12] Moreno-Vozmediano R, Montero RS, Llorente IM. Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures. Computer 2012;45(12):65–72. http://dx.doi.org/10.1109/MC.2012.76.

[13] Apache CloudStack, 2015. URL https://cloudstack.apache.org.

[14] VMWare, 2015. URL http://www.vmware.com.

[15] Amazon Elastic Compute Cloud (EC2), 2015. URL https://aws.amazon.com/ec2/.

[16] Occi—OpenStack, 2015. URL https://wiki.openstack.org/wiki/Occi.

[17] Metsch T, Smith C. Service Sharing Facility, 2015. URL http://pyssf.sourceforge.net.

[18] Eby P. Python web server gateway interface v1.0.1 (PEP 3333), sep 2010. URL https://www.python.org/dev/peps/pep-3333/.

[19] OpenStack project, OpenStack APIs, 2015. URL http://developer.openstack.org/.

[20] OpenStack project, OpenStack Compute API v2.1, 2015. URL http://developer.openstack.org/api-ref-compute-v2.1.html.

[21] Fernández-del Castillo E, Scardaci D, López García Á. The EGI Federated Cloud e-Infrastructure. Procedia Comput Sci 2015;68:196–205. http://dx.doi.org/10.1016/j.procs.2015.09.235. URL http://linkinghub.elsevier.com/retrieve/pii/S187705091503080X.