Toward an Open Cloud Standard

Today's cloud ecosystem features several increasingly divergent management interfaces. Numerous bridging efforts attempt to ameliorate the resulting vendor lock-in for customers. However, as the number of providers continues to grow, the drawback of this approach becomes apparent: the need to maintain adapter implementations. The Open Cloud Computing Interface builds on the fundamentals of modern Web-based services to define a standardized interface for cloud environments while enabling service providers to differentiate their service offerings at the same time.

Andy Edmonds
Intel Labs Europe

Thijs Metsch
Platform Computing

Alexander Papaspyrou
Technische Universität Dortmund

Alexis Richardson
VMware

Any attempt to standardize the interfaces of commercial IT service offerings must cope with the tension between unification and differentiation. Although, from the standardization viewpoint, the agreed-on interface should be alike for all service providers, each vendor naturally strives for a way to expose its unique features and ensure customer retention.

In the domain of cloud computing, several current projects aim to provide a single API for the plethora of proprietary service provider interfaces. However, the most popular of these — libcloud (http://libcloud.apache.org) and DeltaCloud (http://deltacloud.apache.org) — follow a proxy/adapter pattern approach. This has a fundamental limitation: it introduces an additional layer of indirection into the system.

The Open Cloud Computing Interface (OCCI) addresses both the unification and differentiation aspects. It provides a unified and extensible API and offers discoverable capabilities. OCCI reduces the overhead of code and operational management by removing intermediate state management and reclaiming latency losses. Here, we describe OCCI's development and architecture, discuss current issues associated with the spread of proprietary cloud management APIs and approaches to harmonize them, and highlight current OCCI implementations and deployments in the cloud community.

Why a Cloud Standard?

Multiple cloud service and software providers exist, and they all have some kind of API. So why do we need another one?

We could have asked the same question about network software APIs before TCP became widely accepted as a lingua franca for networking. The answer at the time was that customers wanted to be able to buy from any vendor, possibly several at once, without

 1089-7801/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

having to change how their applications were written to use that vendor's software. For noncommercial users, better integration can lead to more effective collaboration.

Standardizing APIs is an integration and interoperability problem. One way to solve it is for the market to pick one vendor as the "standard," and for every competing system to then duplicate that vendor's API. Unfortunately, this approach has some problems. The first is asymmetry: this privileges a single vendor who can then dictate the terms for use of its API. In most cases, this means that whenever the vendor changes its API, everyone else must follow. But the vendor is under no symmetric obligation to cooperate — for instance, by warning others of changes. Worse, the vendor can introduce commercial and legal frictions, including fees and patents. The second problem is fitness: in early stage markets, arguing that a single vendor is fit for all common purposes is difficult, because use cases are still emerging.

In the TCP case, the community solved this problem by picking a technology specification that described real systems with broad cases that weren't vendor controlled. Furthermore, by choosing a suitable legal framework (the IETF), TCP users could have confidence that they wouldn't be sued. By enabling interoperable networking, TCP solved the integration problem.

The chief benefit was commoditization. The creation of an open marketplace for TCP software and solutions providers, as well as an ecosystem of add-on applications, drove down costs. So, not only did this solve the integration problem, but everyone could build better systems faster and bring new business to market at lower cost.

Born from a community of real cloud computing practitioners, OCCI aims to do this for cloud APIs. As with TCP and networking, HTTP has become the lingua franca for cloud APIs. OCCI builds on HTTP using the well-established and broadly accepted REST patterns.[1] Like TCP, it's completely open, and it can evolve and coexist with all open and proprietary APIs.

OCCI Overview

OCCI comprises a set of open, community-led specifications delivered through the Open Grid Forum (OGF) that deal with cloud service resource management. Since OCCI efforts began in April 2009, it has become one of the most promising APIs in cloud standardization.

OCCI's ambitious goal is to enable service providers to differentiate their service offerings through a standardized interface. During its first months, the OCCI working group took a top-down approach and evaluated many of today's available cloud APIs and interfaces. From there, OCCI underwent many development efforts from numerous contributors, which eventually led to real-world implementations and deployments.

Alongside these implementations, the OCCI working group continues to drive and extend the specification. This includes not only work on interoperability test suites and verification mechanisms but also collaborations with other standards organizations working on cloud-related specifications.

The working group is developing the OCCI specification around the ideas of integration, innovation, portability, and, at the core, interoperability. OCCI's modular approach allows for extensibility, flexibility, and the discovery of capabilities. Although it focuses on providing interoperable infrastructures, OCCI can be adopted into many cloud-related setups.[2]

As a unified, extensible API, OCCI is uniquely positioned in the area of cloud standardization, and the open and community-led effort operates similarly to the IETF: it not only uses the same open-minded concepts but also adopts many IETF-driven technologies, mainly surrounding the HTTP specification suite.

Architecture

OCCI is a boundary API that uses HTTP and the REST architectural style. It creates a standardized API for all kinds of service offerings and delivers an interoperable interface for many different services (see Figure 1).

Because OCCI lives on the boundary, service consumers must be able to discover what service providers offer. So, the working group designed the specification with three main goals:

- *Discoverability.* Service consumers can query the service provider to find out what capabilities are available. The information is self-describing and complete. If the service consumer is a broker, it can request that multiple service providers describe what's offered and then choose from among them.

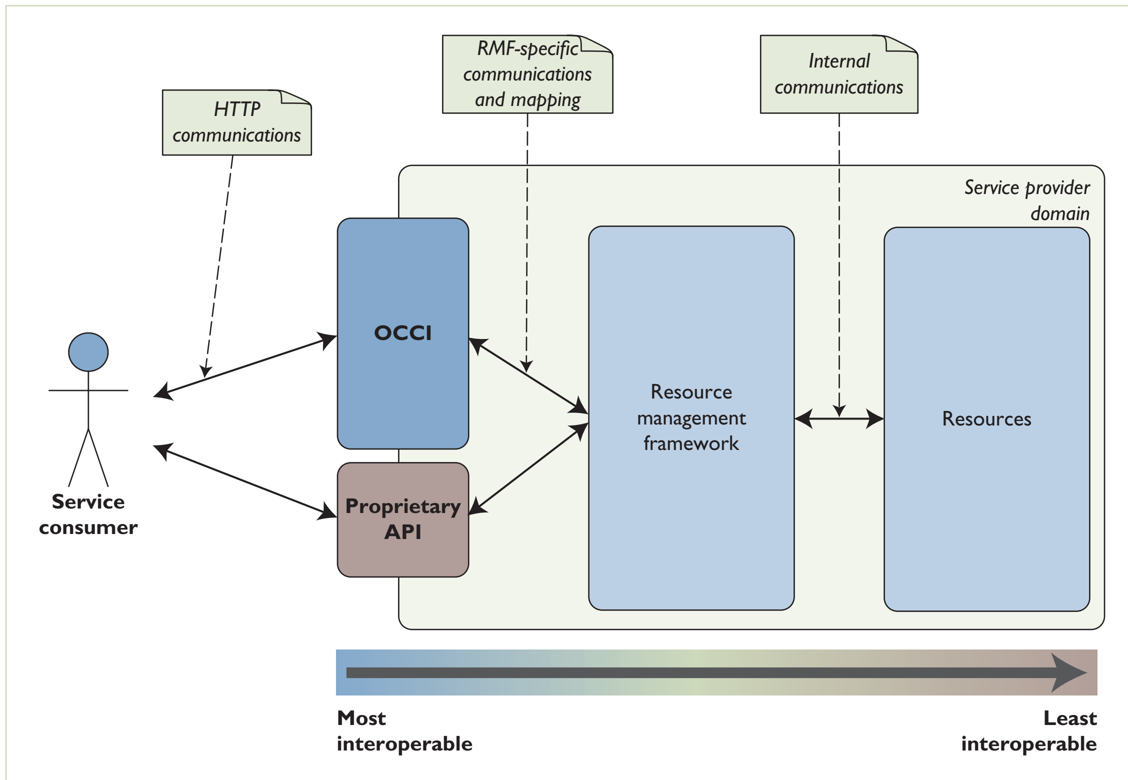

Figure 1. The Open Cloud Computing Interface (OCCI). As a boundary protocol, it helps decouple the proprietary resource management interface from the consumer side, introducing standard mechanisms for interaction over the HTTP protocol. OCCI is designed to coexist with proprietary APIs, yet expose them via standardized means as part of the protocol.

- *Extensibility.* Because cloud computing spans a broad set of offerings, from infrastructure to software as a service (IaaS to SaaS), the OCCI specification must be extensible. Currently, it specifies one extension for the IaaS domain, but the working group can add others, as can providers themselves. Service consumers must be able to discover the extensions available.
- *Modularity.* Because of its extensibility, OCCI must be modular. Indeed, even the OCCI specification itself is split into three documents: the first describes the *core model*, which serves as the foundation; the second describes an extension to this model for the IaaS domain; and the third describes a simple text-based HTTP RESTful rendering. Each document can be used individually, ignored, or replaced as the situation requires.

With these design constraints in mind, let's look at how the OCCI model is constructed and then relayed to and from a client.

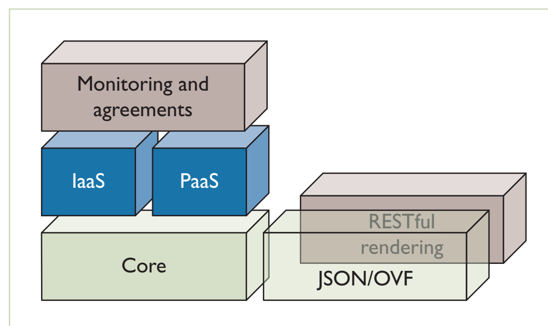

Figure 2. The Open Cloud Computing Interface (OCCI) model. OCCI takes a modular approach, allowing for self-description and extensibility.

Models

OCCI's foundation is the core model, which gives OCCI its self-description and extensibility features. Because the core model is constant, all extensions can build on it. Extensions can even be transitive (that is, extend other extensions), as long as the hierarchy has the core model at its root. Figure 2 shows OCCI's modular approach.

OCCI seeks to cleanly separate its model from the model's rendering (we use the term "rendering" in the sense of serialization here). Because the renderings provide a way to render extensions as well as the core model, service providers don't need to write new renderings when extensions are added or removed. Although OCCI initially focused on the IaaS domain, extensions have been written against the core model to represent grid computing, monitor agreements, and describe platform as a service (PaaS) domains.[2] These extensions don't affect the core model or the renderings.

Core. The core model's main objective is to introduce a type system through what OCCI calls *categories*: each resource in the OCCI namespace has a *type* that defines its capabilities (attributes, actions, and so on).[3] Categories are freely definable and uniquely identifiable using a scheme. They can relate to each other and thereby define a hierarchy.

Categories define only the type of resource. Extensions (such as the infrastructure extension described later) thus define subcategories that extend from the main categories themselves.

First, each resource instance will have one *kind* that's defined by a category in the OCCI model. This kind is immutable and specifies a resource's basic set of characteristics. This includes its location in the hierarchy, attributes, and applicable actions.

The action category defines an *action* and its parameters. An action is a specific operation that can be executed on a resource.

A *mixin* is a way of dynamically adding or removing a resource instance's capabilities. We can think of mixins as a way to inherit additional capabilities by composition (OCCI uses the composite pattern to realize mixins). They are similar to the concept many modern programming languages incorporate that allows for bundling reusable features. Object-oriented programming languages, such as Scala and Ruby, support this concept out of the box. Each OCCI resource can have zero or more mixins. An OCCI mixin also has a set of capabilities such as a location in the URI hierarchy, applicable actions, and attributes. This means that a resource instance's capabilities can be altered over time. OCCI also leverages mixins as a templating mechanism. It defines templates as mixins that can be applied to resource instances, which then assume the template's characteristics. Last but not least, OCCI uses mixins to tag resource instances and so enable folksonomic organization.

Complementing the category class is the *entity* class, which represents a type's instances; entities can be either *resources* or *links*. Figure 3 shows all these elements and how they relate to each other.

The resource entity represents resources that are exposed to the service consumer. The resources that each entity represents can be abstractions from what the service provider exposes through his or her underlying resource management framework. Links create an association between resource entities. A link is a directed association between two resources, but because it derives from an entity, it's also a resource itself and, as such, is exposed as a URI in a RESTful interface. Each entity is of a certain kind and can be assigned multiple mixins. The kinds, mixins, and actions of a resource are all exposed through their category definitions.

Some might view setting up a type system with categories as a remake of MIME media types. However, they are in fact orthogonal to them, because the main difference lies in their purpose: MIME media types indicate how the data delivered is being rendered, whereas categories indicate what data is being rendered. (Remember that in OCCI, the model is decoupled from the rendering. Other renderings might not have MIME media types.) In fact, categories don't attempt to replace MIME media types, but rather complement them and broaden the usage model.

The category type system is more feature-rich than a system using MIME media types alone: categories are self-descriptive, discoverable through a *query* interface, and self-sufficient. A resource in the OCCI model can have multiple categories assigned, exposing several facets simultaneously. In combination with MIME media types, categories deliver a powerful system for resource metadata exposure that supports different renderings of the same information for any given resource type.

Within the query interface, service consumers can find all categories that are usable in a service provider's namespace. The query interface exposes all registered categories and their corresponding hierarchy and describes how each individual category is composed

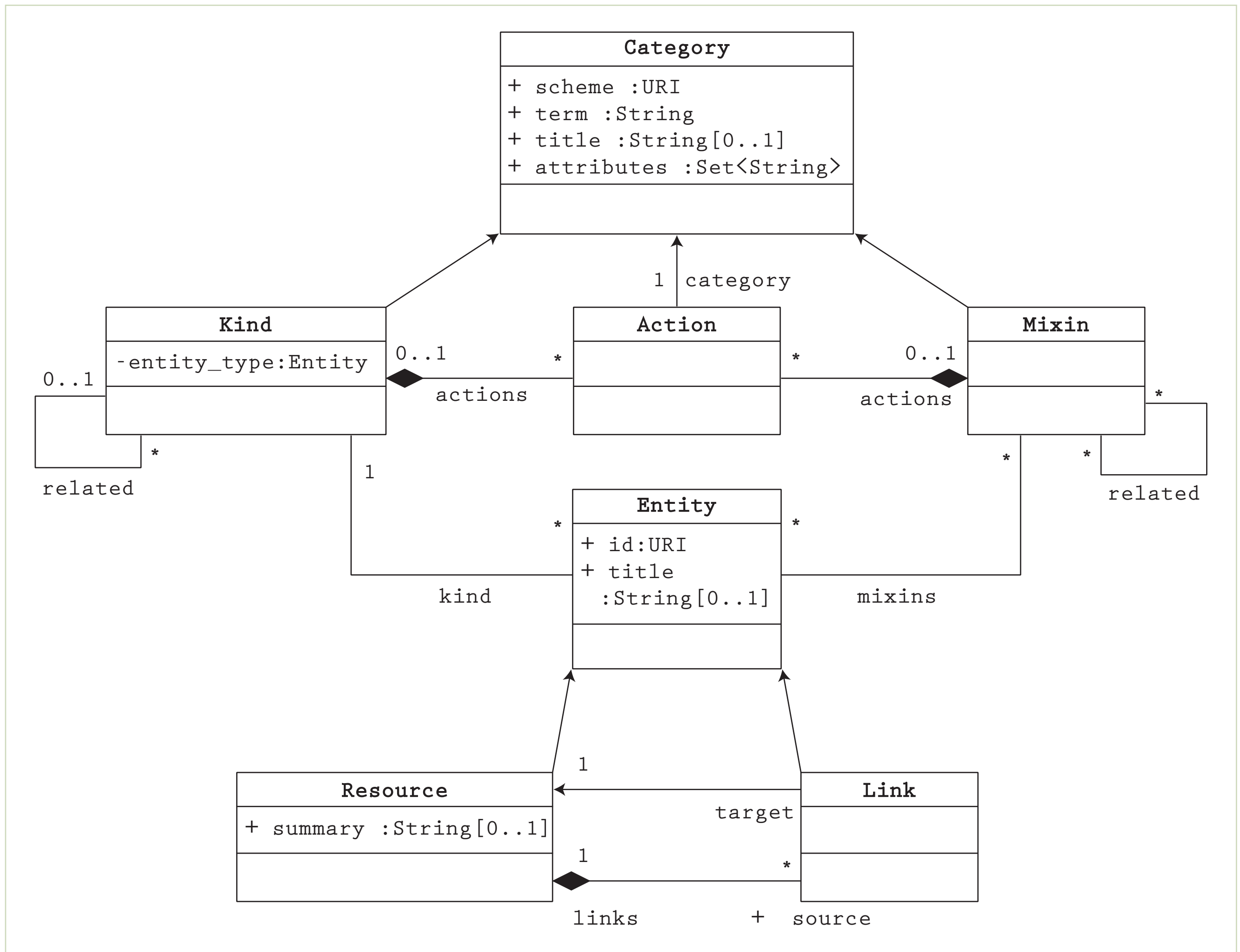

Figure 3. Different elements of the Open Cloud Computing Interface core model. Note the separation into core and meta models. The former describe the foundation of the OCCI type system, introducing the base `Entity` resource, whereas the latter comprises the descriptive part of the model that allows introspection into model instances.

(such as its capabilities and so on). Categories in the query interface can of course be filtered and searched for.

Infrastructure. The infrastructure model[4] is an extension to the core model that models entities in the IaaS domain. In the context of the core model, it describes a set of resources with their capabilities and how to link those resources together if required.

Through categories, the infrastructure specification describes three kinds, which can be used to create the resource instances of *compute*, *storage*, and *network*. When instantiated, any of the three will be accessible through the RESTful interface as URIs. Figure 4 shows how these entities relate to their corresponding parts in the core model.

The extension also further specifies links between resources. Links are necessary to represent associations between resources, such as a compute instance that links to an OSI layer 2 network device router or a storage resource — for example, a database, block device, or Cloud Data Management Interface (CDMI) end point. OCCI accomplishes interoperability with CDMI, for instance, through the linking mechanism.

Service consumers can apply mixins to some of the described infrastructural entities. Commonly, the infrastructural model's mixins are applied — especially in the case of service providers — to network-related entities to give them layer 3 and 4 networking capabilities. Otherwise, such entities would represent only layer 2 networking entities, which aren't particularly useful for internetworking.

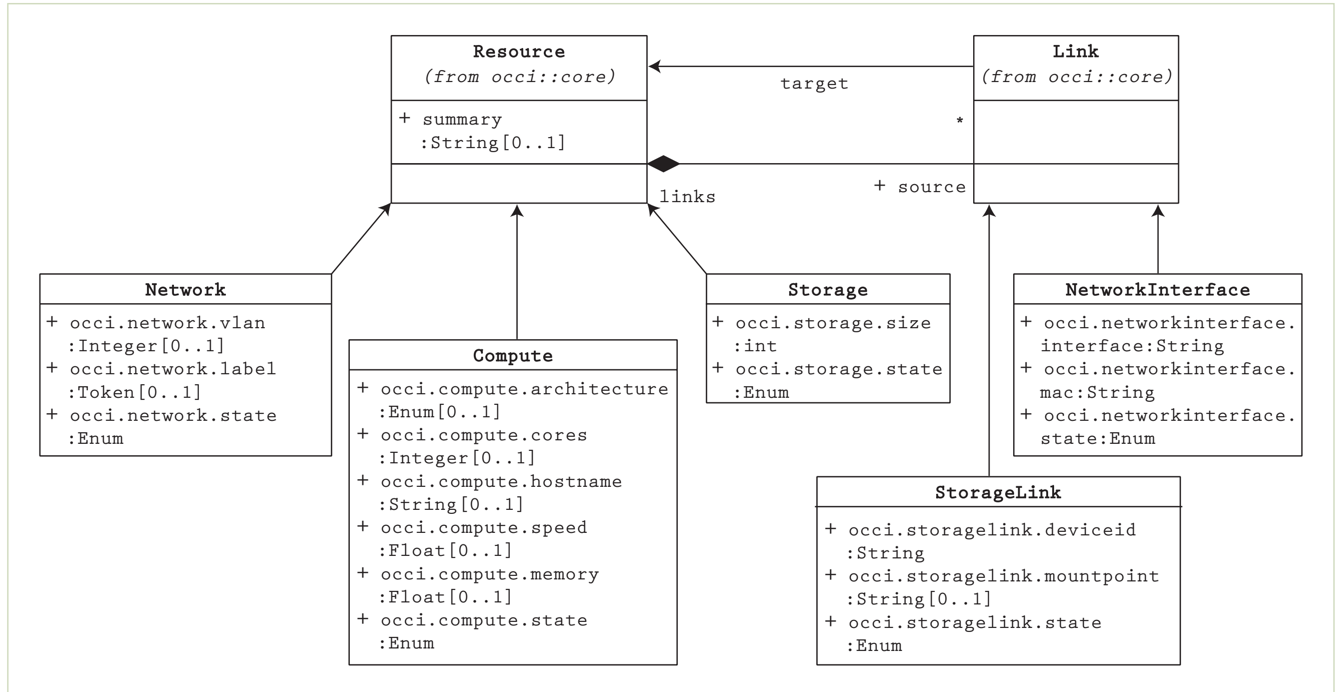

Figure 4. UML model of the infrastructure components in the Open Cloud Computing Interface. On the left side, the types denote a service provider's physical resources (such as a compute resource instance); on the right, ephemeral entities are described (for example, a storage mount point).

Figure 5 illustrates this concept. By dynamically adding an `IPNetwork` mixin to the `Network` resource, extra capabilities are added to the resource instance. Note that multiple mixins can be bound to a resource and that because links are themselves entities, we can add mixins to links as well.

The categories hierarchy ensures that only mixins relevant to the resource instance in question can be added. This way, service consumers and providers are limited to adding a mixin to a resource instance that's in the mixin's hierarchy. This means that applying networking mixins to a `StorageLink` isn't possible. Because the hierarchy itself isn't limited, mixins can build on other mixins.

Figure 6 describes a simple portal service users access to run a MapReduce application. This application has also been used to explore interoperability in the cloud with regard to integrating OCCI, the Open Virtualization Format (OVF), and CDMI.[5] The service is described using OCCI's infrastructure extension model.

Service consumers can access all entities illustrated in Figure 6 through their own URIs. The links, compute, and network resource instances would be in the same provider's namespace (but aren't required to be), whereas the storage entities could be hosted through a CDMI-compatible interface. Note that the links are essential to the core model's ability to express associations.

Figure 7 shows a request for creating the virtual machine that represents the portal.

When creating the instance using HTTP `POST`, a category for the virtual machine's kind is present alongside the network association. This tells the service provider to connect the virtual machine to a certain service-provider-managed network.

Where OCCI Differs Architecturally

OCCI sits on the boundary between the resource management framework and the service consumer. It isn't a "Web-API-to-Web-API" proxy pattern, such as those used in DeltaCloud and libcloud. Although a proxy pattern offers more flexibility, it also affects latency and manageability: the proxy pattern is another level of indirection, so requests to or from a client incur an additional delay. Moreover, the software implementing the proxy pattern is another entity to manage and maintain. A proxy pattern implements support for various providers through drivers. If one provider changes its interface, the driver within the implementing

proxy software must also be updated. If support for multiple providers is present, the maintenance requirement again increases.

Using OCCI as the interface to the resource management framework removes the need for proxies, drivers, and even multiple drivers (see Figure 8). We should thus view proxies as a temporary solution to support cloud operators wishing to expose proprietary, legacy interfaces.

OCCI enables resource management through a standardized API that directly targets a resource management framework-specific API; this is quite different in intent from proxy-style frameworks. Also, using OCCI reduces the amount of indirection and abstraction required to get to the final target resource management framework.

Overall, OCCI enables system architecture optimization by bringing the API closer to the managed resources. It avoids additional dependencies and inefficiencies, and reduces the overall management and maintenance of system components. Given that proxies hold information about ongoing interactions, avoiding them further reduces additional state management.

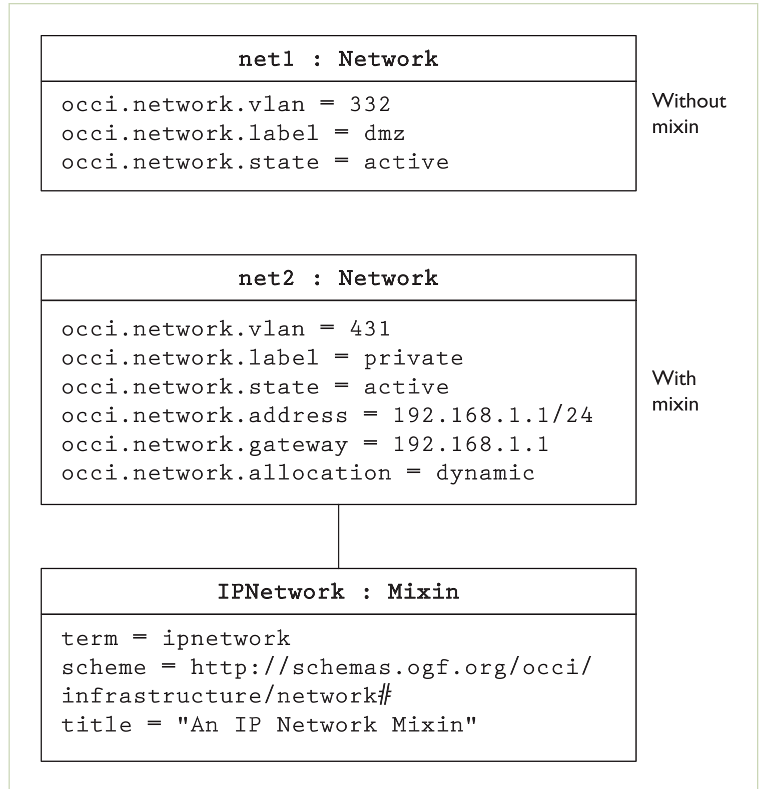

Figure 5. Relationships between different mixins. The upper resource, `net1`, depicts a physical network without layer 3 capabilities. The lower resource, `net2`, attaches an `IPNetwork` mixin that adds these capabilities – in this case, a network address and a gateway.

How OCCI Uses the Web

The OCCI HTTP specification[6] details how the core model and its extensions can be transported over the wire. When implemented and deployed, OCCI uses many of today's available HTTP features. It builds on the Resource Oriented Architecture (ROA) paradigm and uses REST to handle client and service interactions. Additionally, it defines some simple ways to filter and query the service provider.

Each entity (that is, resources and links) is exposed through URIs. Service consumers can use the normal set of HTTP verbs (`POST`, `PUT`, `GET`, and `DELETE`) to manage these resources, and can alter resource instances by updating their representation.

In this context, Tim Bray notes the idea of *controller functions* (see www.tbray.org/ongoing/When/200x/2009/03/20/Rest-Casuistry): although a RESTful approach would be to change a resource instance's attributes to initiate a state change, this doesn't always make sense. Like all requests that reflect an update of a resource instance using HTTP `PUT`, updating a resource should be *idempotent*. This means that repeated requests against a resource will always have the identical output result and effect on the system. Triggering operations such as shutdown, however, might lead to halting, killing, or suspending. Naturally, the result of the operation can't be identical to the request in such a case, due to the transition in state.

OCCI adopts this viewpoint through the notion of actions, triggered by the HTTP `POST` verb. Much like pushing a button that triggers a process in the background, an action leads to different state changes in a life cycle.

Actions within the OCCI model can have parameters and, as detailed in the core model, are exposed using a category definition. They are therefore discoverable and, as Figure 9 shows, can be associated with resource instances. Service consumers would use the request in Figure 9 to retrieve a service provider's single category (through the filtering mechanism) using the query interface.

The current OCCI HTTP specification leverages several IETF recommendations, especially the core HTTP RFC 2616. Other important specifications include URIs that can identify

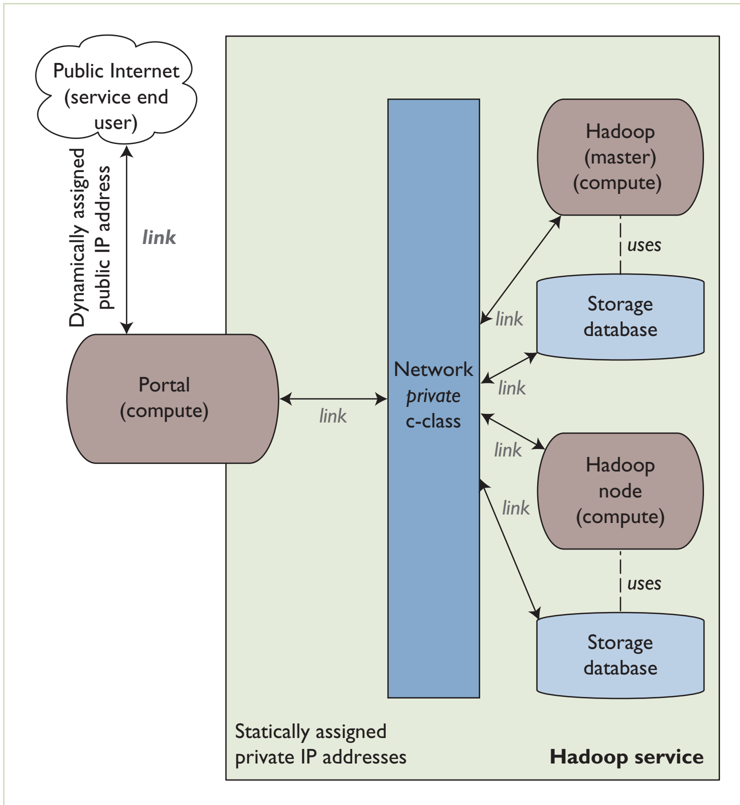

Figure 6. Possible Open Cloud Computing Interface application areas. Users can access this simple portal service to run a MapReduce application.

and handle resources (RFC 3986), well-known URIs that clearly define the query interface's entry point (RFC 5785), and HTTP Authentication (RFC 2617) to deal with authentication mechanisms.

Although OCCI is built on these specifications, service providers might choose to leverage other RFCs (RFCs 3280 and 5246 for security, for example) to offer clients an even richer API.

Ongoing efforts would give OCCI a more structured rendering, instead of the simple text rendering that evolved during the standard's creation, and the working group is attempting to reach consensus on these topics. A current draft describes how JavaScript Object Notation (JSON) can be used as a drop-in replacement (without needing to modify the core or infrastructure model) for the current rendering that the OCCI models and their extensions use.

The OCCI working group is currently investigating asynchronous behaviors associated with service offerings. This is useful for features such as notifying service consumers when monitoring is being used, and providing a constant stream of up-to-date information. A monitoring and agreement negotiation extension for

```
> POST /compute/ HTTP/1.1#
> User-Agent: curl/7.21.1 (i386-pc-solaris2.11) libcurl/7.21.1 OpenSSL/
    0.9.8o zlib/1.2.3 libidn/1.9
> Host: localhost:8888
> Accept: */*
> Content-type: text/occi
> Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure"
> X-OCCI-Attribute: occi.compute.speed=2
> Link: </network/123>; rel="http://schemas.ogf.org/occi/infrastructure#network";
        category="http://schemas.ogf.org/occi/infrastructure#networkinterface";
        occi.networkinterface.interface="eth0";
        occi.networkinterface.mac="00:11:22:33:44:55"
>

< HTTP/1.1 201 OK
< Content-Length: 2
< Content-Type: text/plain; charset=UTF-8
< Location: http://localhost:8888/compute/40675abc-c4ca-e6dd-ac7e-fa057cd5b164
< Server: pyssf OCCI/1.1
<
```

Figure 7. A compute resource instantiation over the Open Cloud Computing Interface using the HTTP rendering. All information is provided inline, such that the service provider can infer missing data and start up the corresponding machine with the requested properties.

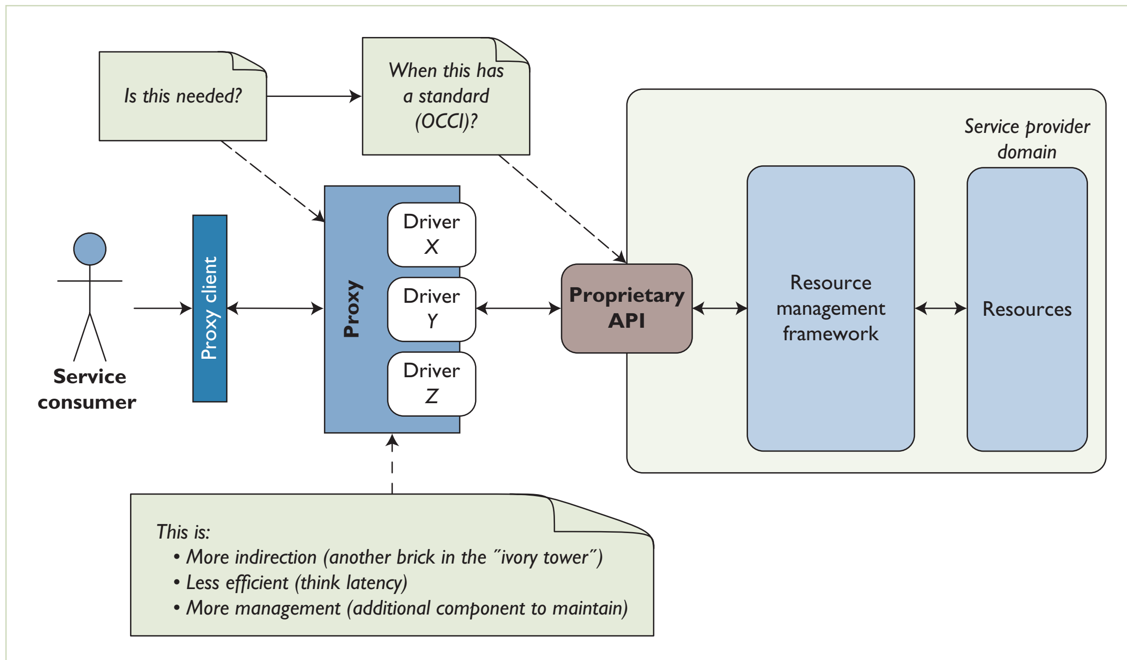

Figure 8. Open Cloud Computing Interface as a replacement for proxy-based API approaches. Here, we see the overhead of an additional software (or even middleware) layer in the process, adding overall latency and, more importantly, maintenance costs per additional driver.

```
> GET /.well-known/org/ogf/occi/ HTTP/1.1
> User-Agent: curl/7.21.1 (i386-pc-solaris2.11) libcurl/7.21.1 OpenSSL/0.9.8o zlib/1.2.3 libidn/1.9
> Host: localhost:8888
> Accept: */*
> Content-type: text/occi
> Category: compute;scheme="http://schemas.ogf.org/occi/infrastructure"
>

< HTTP/1.1 200 OK
< Content-Length: 592
< Etag: "1fb0432a8222fb441a6cbf5e6acb02b701a2ed94"
< Content-Type: text/plain
< Server: pyssf OCCI/1.1
Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure#"; class="kind";
          title="A compute instance"; rel="http://schemas.ogf.org/occi/core#resource";
             location=/compute/;
          attributes="occi.compute.architecture occi.compute.cores occi.compute.hostname
                      occi.compute.speed occi.compute.memory occi.compute.state";
          actions="http://schemas.ogf.org/occi/infrastructure/compute/action#start
                   http://schemas.ogf.org/occi/infrastructure/compute/action#stop
                   http://schemas.ogf.org/occi/infrastructure/compute/action#restart
                   http://schemas.ogf.org/occi/infrastructure/compute/action#suspend"
<
```

Figure 9. Filtering resources through the query interface. A service consumer would use this request to retrieve a service provider's single category (here, `compute`) to discover its capabilities and find the location of its instances.

OCCI is also under development so that service providers can offer service-level agreements (SLAs) to their customers.

Impact and Implementations

To be successful, standards must both be grounded in reality, taking their requirements from real-world use cases, and respect core tenets of successful standardization activities. One such activity, the Advanced Message Queuing Protocol (AMQP),[7] defined a successful standard to be a collective effort and a fully defined, open, royalty-free, unpatented specification that enables anyone to implement a compatible service; be cited in an organization that can protect these features; and have real-world implementations and live deployments.

The OCCI working group is a collective of stakeholders from industry and academia. All members work together under the intellectual property rights protection that the OGF offers. OCCI is clearly defined, royalty free, and lets anyone implement the service. Numerous OCCI implementations — many of them open source — are available, including Eucalyptus (www.eucalyptus.com), OpenNebula (http://opennebula.org), OpenStack (http://openstack.org), and libvirt (http://libvirt.org). Other OCCI-related software is also available to the community.

Various deployments hosting live systems use OCCI. For example, SARA's HPC Cloud system (www.sara.nl/services/cloud-computing) offers high-performance computing resources to scientists from areas such as geography, ecology bio-informatics, and computer science. Currently, the system comprises 608 cores and 4.75 Tbytes of RAM distributed so that each node has 10 Tbytes of local storage.

The OpenStack infrastructure management framework shares several of OCCI's ideals and has many early adopters (including Dell, Rackspace, AT&T, and Hewlett-Packard). OCCI can provide interoperability for not only the various OpenStack deployments but also deployments of other infrastructure management frameworks, such as OpenNebula. A superb example is the European Grid Infrastructure (EGI) federated environment (more than 1,400 cores), which uses multiple infrastructure and management frameworks but harmonizes them using OCCI. Another interesting use of OCCI that impressively demonstrates its flexibility is within the CompatibleOne project (www.compatibleone.org), which uses OCCI as the core of its architecture, not only to provision IaaS-type instances but also to broker between many service providers.

In addition to fulfilling the four key points required for a successful standard, many global, coordinating standards activities have expressed interest in OCCI:

- The US National Institute of Standards and Technology (NIST) has listed and noted OCCI in its cloud computing program strategic efforts, particularly in the area of Standards Acceleration to Jumpstart Adoption of Cloud Computing (SAJACC).
- The Standards and Interoperability for e-Infrastructure Implementation Initiative (SIENA) has named OCCI a key recommendation for cloud standards in its "European Roadmap on Grid and Cloud Standards for eScience and e-Government" (www.sienainitiative.eu/Repository/Filescaricati/8ee3587a-f255-4e5c-aed4-9c2dc7b626f6.pdf). Integrating OCCI with CDMI and OVF has been recommended for future eScience and e-government platforms.
- The UK government's G-Cloud initiative came out of the UK's cabinet office in response to the growing interest in cloud computing within the government. Several reports, among them the *Technical Architecture Workstrand Report*, recommend using OCCI (www.cabinetoffice.gov.uk/sites/default/files/resources/08-G-CLOUD-TechnicalArchitectureWorkstrand-Report.pdf).
- The German Federal Ministry of Economics and Technology has identified OCCI as the leading standard for cloud computing in terms of both maturity and impact in its recently published report, "The Standardization Environment for Cloud Computing" (www.bmwi.de/English/Navigation/Service/publications,did=476736.html).
- As the major stakeholder for e-infrastructure for the European research area, EGI has adopted OCCI as the flagship standard for infrastructure management within its overall eScience platform vision. To this end, EGI integrates OCCI with other standards toward a federated IaaS ecosystem profile. Complementing this activity in the US, FutureGrid is also considering the using OCCI.

OCCI is one of the first and most mature efforts to bring standardized protocols and interfaces to the cloud. It can evolve and coexist with all open and proprietary APIs, and it encompasses an evolving world of cloud resources. Furthermore, the OCCI team has actively collaborated with other open standards initiatives such as the Distributed Management Task Force (DMTF) and the Storage Networking Industry Association (SNIA). The output of these collaborations is critical to forging ahead in the world of cloud standards and demonstrating that enough intersecting and complementary standards exist to realize a standards-based, open, and interoperable cloud.

The OCCI community offers an API and code that implements that API along with compliance and verification testing suites, but the adjoining communities are providing real implementations for different infrastructure management frameworks backing the API. Work is under way to define additional extensions and refinements to the specification, with a focus on business-related requirements such as audit and billing. Along with this work are other efforts (such as FI-ware; www.fi-ware.eu) that use the specification to expose service differentiators.

OCCI isn't just a specification — it represents a collective effort to create one of the first standards in the cloud space. OCCI's extensibility features offered through its core model, extensions, and mixins can be added to other kinds of interfaces and in general be useful for other Internet standards. Thus, we believe the future is bright for broadly interoperable cloud computing.

Acknowledgments

We thank all past and present contributors of the Open Cloud Computing Interface working group. Special thanks to John Kennedy (Intel Labs Europe), Winston Bumpus (VMware), and the *IEEE Internet Computing* reviewers for their valuable insights.

References

1. R. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," doctoral dissertation, Univ. of California, Irvine, 2000.
2. A. Edmonds, T. Metsch, and A. Papaspyrou, "Open Cloud Computing Interface in Data Management-Related Setups," *Grid and Cloud Database Management*, vol. 1, Springer, 2011, pp. 23–48.
3. M. Behrens et al., "Open Cloud Computing Interface — Core," *OGF Document Series*, A. Edmonds et al., eds., recommendation track no. 183, Open Grid Forum, June 2011.
4. M. Behrens et al., "Open Cloud Computing Interface — Infrastructure," *OGF Document Series*, T. Metsch and A. Edmonds, eds., recommendation track no. 184, Open Grid Forum, June 2011.
5. A. Edmonds, T. Metsch, and E. Luster, "An Open, Interoperable Cloud," *InfoQ*, July 2011, www.infoq.com/articles/open-interoperable-cloud.
6. M. Behrens et al., "Open Cloud Computing Interface — RESTful HTTP Rendering," *OGF Document Series*, T. Metsch and A. Edmonds, eds., recommendation track no. 185, Open Grid Forum, June 2011.
7. J. O'Hara, "Toward a Commodity Enterprise Middleware," *ACM Queue*, vol. 5, no. 4, 2007, pp. 48–55.

Andy Edmonds is an applied researcher in the Intel Innovation Open Lab and Cloud Services Lab. His research interests include distributed and system architectures, virtualization, service-oriented architectures, and cloud computing. Edmonds has a research master's degree in distributed systems from Trinity College Dublin. He currently cochairs the Open Grid Forum's Open Cloud Computing Interface working group. Contact him at andy@edmonds.be.

Thijs Metsch is a senior technical consultant at Platform Computing. His work has given him a deep knowledge of distributed systems design, grid, and cloud computing technologies. Metsch is a graduate engineer in Information Technology from the University of Cooperative Education Mannheim, Germany. He cofounded and currently cochairs the Open Grid Forum's Open Cloud Computing Interface working group. Contact him at tmetsch@gmail.com.

Alexander Papaspyrou is a researcher in computer science at Technische Universität Dortmund, Germany. His interests are in infrastructure capacity planning, adaptive resource management, mobile applications, and standardization. Papaspyrou has an MS in computer science from Technische Universität Dortmund. He currently cochairs the DCI Federation Working Group, which develops profiles for federation use cases in on-demand infrastructures. Contact him at alexander@papaspyrou.name.

Alexis Richardson is a senior director for the VMware Cloud Application Platform. He cochairs the Open Cloud Computing Interface working group and helped create the Advanced Message Queuing Protocol (AMQP). Prior to joining VMware, he founded several companies, including RabbitMQ. Contact him at arichardson@vmware.com.