OCCI Walkthrough

Overview

THIS WALKTHROUGH DOCUMENT DOES NOT REFLECT THE CURRENT STATE OF THE SPECIFICATION. IT WILL BE UPDATED ON PUBLICATION.

The Open Cloud Computing Interface (OCCI) is an API for managing cloud infrastructure services (also known as Infrastructure as a Service or IaaS) which strictly adheres to REpresentational State Transfer (REST) principles and is closely tied to HyperText Tranfer Protocol (HTTP). For simplicity and scalability reasons it specifically avoids Remote Procedure Call (RPC) style interfaces and can essentially be implemented as a horizontally scalable document repository with which both nodes and clients interact.

This document describes a step-by-step walkthrough of performing various tasks as at the time of writing.

Getting Started

Connecting

Each implementation has a single OCCI end-point URL (we'll use http://example.com/) and everything you need to know is linked from this point - configuring clients is just a case of providing this parameter. In the simplest case the end-point may contain only a single resource or type of resource (e.g. a hypervisor burnt into the BIOS of a motherboard exposing compute resources, a network switch/router exposing network resources or a SAN exposing storage resources) and at the other end of the spectrum it may provide access to a global cloud infrastructure (e.g. the "Great Global Grid" or GGG). You will only ever see those resources to which you have access to (typically all of them for a private cloud or a small subset for a public cloud) and flexible categorisation and search provide fine-grained control which resources are returned, allowing OCCI to handle the largest of installations. You will always connect to this end-point over HTTP(S) and given the simplicity of the interface most user-agents are suitable, including libraries (e.g. urllib2, LWP), command line tools (e.g. curl, wget) and full blown browsers (e.g. Firefox).

Authenticating

When you connect you will normally be challenged to authenticate via HTTP (this is not always the case - in secure/offline environments it may not be necessary) and will need to do so via the specified mechanism. It is anticipated that most implementations will require HTTP Basic Authentication over SSL/TLS so at the very least you should support this (fortunately almost all user-agents already do), but more advanced mechanisms such as NTLM or Kerberos may be deployed. Certain types of accesses (such as a compute resource querying OCCI for introspection and configuration) may be possible anonymously (having already been authenticated by interface and/or IP address). Should you be redirected by the API to a node, storage device, etc. (for example, to retrieve a large binary representation) then you should either be able to transparently authenticate or a signed URL should be provided. That is, a single set of credentials is all that is required to access the entire system from any point.

Representations

As the resource itself (e.g. a physical machine, storage array or network switch) cannot be transferred over HTTP (at least not yet!) we instead make available one or more representations of that resource. For example, an API modeling a person might return a picture, fingerprints, identity document(s) or even a digitised DNA sequence, but not the person themselves. A circle might be represented by SVG drawing primatives or any three distinct points on the curve. For cloud infrastructure there are many useful representations, and while OCCI standardises a number of them for interoperability purposes,

an implementation is free to implement others in order to best serve the specific needs of their users and to differentiate from other offerings. Other examples include:

- Open Cloud Computing Interface (OCCI) descriptor format (application/occi+xml)
- Open Virtualisation Format (OVF) file (application/ovf+xml?)
- Open Virtualisation Archive (OVA) file (application/x-ova?)
- Screenshot of the console (image/png)
- Access to the console (application/x-vnc)

The client indicates which representation(s) it desires by way of the URL and/or HTTP Accept headers (e.g. HTTP Content Negotiation) and if the server is unable to satisfy the request then it should return HTTP 406 Not Acceptable.

Descriptors

In addition to the protocol itself, OCCI defines a simple key/value based descriptor format for cloud infrastructure resources:

compute	Provides computational services, ranging from dedicated physical machines (e.g. Dedibox) to virtual machines (e.g. Amazon EC2) to slices/zones/containers (e.g. Mosso Cloud Servers).
network	Provides connectivity between machines and the outside world. Usually virtual and may or may not be connected to a physical segment.
storage	Provides storage services, typically via magnetic mass storage devices (e.g. hard drives, RAID arrays, SANs).

Given the simplicity of the format it is trivial to translate between wire formats including plain text, JSON, XML and others. For example:

```
occi.compute.cores 2
compute.speed 3200
compute.memory 2048
```

Identifiers

Each resource is identified by its dereferenceable URL which is by definition unique, giving information about the origin and type of the resource as well as a local identifier (the combination of which forms a globally unique compound key). The primary drawback is that the more information that goes into the key (and therefore the more transparent it is), the more likely it is to change. For example, if you migrate a resource from one implementation to another then its identifier will change (though in this instance the source should provide a HTTP 301 Moved Permanently response along with the new location, assuming it is known, or HTTP 410 Gone otherwise).

In order to realise the benefit of transparent, dereferenceable identifiers while still being able to track resources through their entire lifecycle an immutable UUID attribute should be allocated which will remain with the resource throughout its life. This is particularly important where the same resource (e.g. a network) appears in multiple places.

New implementations should use type 4 (random) UUIDs anyway, as these can be safely allocated by any node without consulting a register/sequence, but where existing identifiers are available they should be used instead (e.g. http://amazon.com/compute/ami-ef48af86).

Operations

Create

To create a resource simply POST it to the appropropriate collection (e.g. /compute, /network or /storage) as an HTML form (supported by virtually all user agents) or in another supported format (e.g. OVF):

```
POST /compute HTTP/1.1
Host: example.com
Content-Length: 35
Content-Type: application/x-www-form-urlencoded

compute.cores=2&compute.memory=2048
```

Rather than generating the new resource from scratch you may also be given the option to GET a template and POST or PUT it back (for example, where "small", "medium" and "large" instances or pre-configured appliances are offered).

Retrieve

The simplest command is to retrieve a single resource by conducting a HTTP GET on its URL (which doubles as its identifier):

```
GET /compute/b10fa926-41a6-4125-ae94-bfad2670ca87 HTTP/1.1
Host: example.com
```

This will return *a HTTP 300 Multiple Choices response containing a list of available representations for the resource as well as a suggestion in the form of a HTTP Location: header of* the default rendering, which should be HTML (thereby allowing standard browsers to access the API directly). An arbitrary number of alternatives may also be returned by way of HTTP Link: headers.

If you just need to know what representations are available you should make a HEAD request instead of a GET - this will return the metadata in the headers without the default rendering.

Some requests (such as searches) will need to return a collection of resources. There are two options:

Pass-by-reference	A plain text or HTML list of links is provided but each needs to be retrieved separately, resulting in O(n+1) performance.
Pass-by-value	A wrapper format such as Atom is used to deliver [links to] the content as well as the metadata (e.g. links, associations, cahching information, etc.), resulting in O(1) performance.

Update

Updating resources is trivial - simply GET the resource, modify it as necessary and PUT it back where you found it.

Delete

Simply `DELETE` the resource:

```
DELETE /compute/b10fa926-41a6-4125-ae94-bfad2670ca87 HTTP/1.1
Host: example.com
```

Sub-resource Collections

(For want of a better name)

Each resource may expose collections for functions such as logging, auditing, change control, documentation and other operations (e.g. http://example.com/compute/123/log/456) in addition to any required by OCCI. As usual CRUD operations map to HTTP verbs (as above) and clients can either PUT entries directly if they know or will generate the identifiers, or POST them to the collection if this will be handled on the server side (using POST Once Exactly (POE) to ensure idempotency).

Requests

Requests are used to trigger state changes and other operations such as backups, snapshots, migrations and invasive reconfigurations (such as storage resource resizing). Those that do not complete immediately (returning HTTP 200 OK or similar) must be handled asynchronously (returning HTTP 201 Accepted or similar).

```
POST /compute/123/requests HTTP/1.1
Host: example.com
Content-Length: 35
Content-Type: application/x-www-form-urlencoded

state=shutdown&type=acpioff
```

The actual operation may not start immediately (for example, backups which are only handled daily at midnight) and may take some time to complete (for example a secure erase which requires multiple passes over the disk). Clients can poll for status periodically or use server push (or a non-HTTP technology such as XMPP) to monitor for events.