Application level interface for a *Cloud Monitoring* service

Augusto Ciuffoletti

University of Pisa — Dept. of Computer Science

Abstract

We introduce a new paradigm, based on an extension of the Open Cloud Computing Interface (OCCI), for the *on demand* monitoring of the cloud resources provided to a user. We have extended the OCCI with two new sub-types of core entities: one to collect the measurements, and the other to process them. The user can request instances of such entities to implement a monitoring infrastructure.

The paradigm does not target a specific cloud model, and is therefore applicable to any kind of resource provided *as a service*. The specifications include only the minimum needed to describe a monitoring infrastructure, thus making this standard extension simple and easily adoptable. Despite its simplicity the model is able to describe complex solutions, including private/public clouds, and covers both infrastructure and application monitoring.

To highlight the impact of our proposal in practice, we have designed an engine that deploys a monitoring infrastructure using its OCCI-compliant descriptions. The design is implemented in a prototype that is available as open source.

Keywords: Monitoring of Cloud Resources, Monitoring-as-a-Service, Open Cloud Computing Interface (OCCI), RESTful service, Cloud Interoperability

1. Introduction

One of the most characteristic features of cloud computing is the *on demand* delivery of IT resources. The interface between the user, who issues resource requests, and the provider, who synthesizes and delivers them, is a critical component. Aside from the existence of a graphical user interface, users describe the features of the resources they need via the Application Programming Interface (API).

From a provider's perspective, an API should ensure the control of the kinds of services offered to the user, i.e. the resources it offers, their features, and how resources communicate with each other and with the outside. This means that APIs tend to diverge from one another, following the legitimate intent of the providers to differentiate and improve their services.

However, this is a reason of concern for the user. In their paper, published in 2010 [2], Armbrust et al. stated that *business continuity* and *data lock-in* were primary obstacles for cloud adoption. In both cases the existence of a uniform interface across providers helps, since being able to switch smoothly from one to another encourages cloud adoption. Therefore the interests of both the user and the provider find, to some extent, a point of convergence.

Since 2009 there has been a noticeable effort to standardize an API for cloud computing. A wiki reference to these developments is at `www.cloud-standards.com`, where the representatives active in each organization report their current progress. However major providers (Amazon, Microsoft, and Google) have been motivated to differentiate their products in order for them to become the *de facto* standard, also by participating in forums dedicated to designing a standard (such as the Distributed Management Task Forum, DMTF).

Somewhat in parallel with this trend is the evolution of the Open Cloud Computing Interface (OCCI) protocol, which is independent from providers and open to the contribution of academic and industrial research. OCCI provides an interoperable foundation for the cloud API, while allowing the widest range of options for the services offered. OCCI adoption alone does not guarantee interoperability, but is a fundamental step in the process.

The *Core* specification of the OCCI protocol [19] provides an abstract representation of the entities available to the user as cloud resources. It is a generic schema that defines the API of the service, and lays the foundations for further applications. For instance, the Infrastructure as a Service (IaaS) API is described in a companion document [20] that introduces specialized entities representing *Compute*, *Storage* and *Network* resources. The term *extension* is used to indicate a document that specializes the *Core* model to a certain field of application, as for the IaaS document referenced above.

The OCCI working group is developing an extension of the core API that allows the user to describe a monitoring infrastructure in the cloud. It has been recently proposed as an OGF document and can be found in the group repository at `https://redmine.ogf.org/projects/occi-wg/repository`. It is intended for those users that want control over cloud resources, including being able to measure how well these resources perform [22]. This perspective is not only related to the *pay-per-use* nature of cloud services, but is also of interest in private or federated environments. The evolution of *cloud sourcing* [18] towards business-critical applications [14] emphasizes this need, so that the availability of a customized cloud monitoring service (also known as *Monitoring-as-a-Service* [23]) is a discriminating factor in the selection of a provider.

In this paper we investigate the OCCI extension for *resource monitoring*. First we show how it fits in with the existing literature and current commercial solutions. Next we describe the *extension* and demonstrate its soundness with use cases and examples. Finally we describe the design and the development of an engine that deploys a monitoring infrastructure *on-demand*, thus providing a practical solution that we believe opens the way to industrial-grade solutions.

1.1. From cloud monitoring to monitoring-as-a-service

Cloud monitoring has been strongly influenced by an extensive study of distributed monitoring frameworks [13, 16, 21, 23, 27]. Following a commonly-adopted model, the provider is responsible for implementing the monitoring infrastructure, and the user *subscribes* to receive the data. However, a *Monitoring-as-a-Service* interface should allow the user to have control over the monitoring infrastructure, not just to subscribe to available data.

This claim is endorsed by a number of works that do not directly focus on the resources that are provided *on demand*. In [22] the authors illustrate how monitoring a business process requires a highly adaptive monitoring framework. From a different perspective, a plant automation [26] study illustrates a monitoring framework whose requirements include the interoperability of heterogeneous systems. An investigation into self-adaptive monitoring, which adjusts itself depending on the situation, is proposed in [17]. Although such works do not refer to an implementation *in the cloud*, they support the claim that monitoring frameworks cannot be limited to a *publish-subscribe* model.

In fact, all major cloud providers offer an API to configure resource monitoring and there is a clear trend towards giving the user the tools to customize this service. **Windows Azure** enables users to configure auto-scaling and alerts based on defined metrics. With **Google Cloud**, **Amazon AWS**, **Rackspace** users can define custom metrics, which can be used to produce alerts. In particular **Rackspace** allows a server to be polled from various geographical regions. **Stackdriver** offers a service that simplifies and improves the monitoring of AWS resources. With all these providers the service is limited to the production of measurements, with a limited range of asynchronous user-defined triggered actions. One step ahead is **Openstack** which has an ongoing *Monitoring as a Service* project named MONaaS. MONaaS is based on a preexisting service used for billing, and offers many ways to process data, like a cloud service that polls cloud resources and synthesizes aggregated metrics. **CompatibleOne** is a research project that follows a similar approach. It aims to provide the widest possible compatibility with existing cloud computing platforms, thus making it a unique tool to interact with all major providers. The interface embed tools to describe a monitoring framework, including a specialized *monitoring agent* that oversees the monitoring *probes* [16].

However there is a significant interoperability problem. If we consider the **EGI** project http://www.egi.eu, which gathers the European scientific grids into a unique infrastructure, the adoption of a common interface for cloud services is mandatory when monitoring a heterogeneous system. With the adoption of OCCI as the interface for all the IaaS offered by all partners in the project, EGI provided the way for a uniform test of all infrastructures in the SAM system. Although the EGI case regards cloud management, it reflects the same problem experienced by users who want to deploy a complex infrastructure in a multi-tenant environment, or in the case of hybrid public/private clouds.

Providers are thus investing considerable effort in implementing a flexible resource monitoring service. However, none of these systems is sufficiently flexible to do any more than trigger an action based on measurements, and each of them provides a distinct interface. The purpose of this paper is to go beyond this point.

An extension for monitoring infrastructures can be found in the road-map of the OCCI working group. One important purpose addresses those users that obtain resources from several providers, since they would benefit from a uniform interface for cloud monitoring. A proposed extension has been discussed within the OCCI working group [8], and is summarized in [5]. In [15] other authors extend OCCI monitoring by adding the capacity to recover from a deviation from a defined Service Level Agreement. The **CompatibleOne** project mentioned above includes the implementation of an API following the OCCI proposed extension. In [25] the authors suggest another extension of the OCCI core model which supports the whole life-cycle, from resource delivery to reconfiguration, including monitoring.

This paper aims to demonstrate the soundness of the proposal in [5]. This is done by comparing it against realistic cases, and by designing and implementing a prototype for a specific case.

The design tackles the challenge of synthesizing the monitoring infrastructure using its OCCI-compliant description. We believe that this result is original: among the commercial providers listed above, only **OpenStack** is figuring out a project for on-demand monitoring, while the others offer a limited range of triggered actions. It supports the proposed scheme, and gives a practical indication for a real scale implementation.

The prototype implementation has many limits: among others, it is not designed for performance, and there are no security measures in place. The rationale behind these options is to keep the design as transparent as possible, trading off code optimization for clarity. But, in combination, the design and the prototype are evidence that there is a real implementation of the abstract paradigm that is behind the schema proposed in [5].

The next section describes the model: first with an overview of the OCCI protocol, followed by a description of the monitoring extension, highlighting how it improves the state of the art.

2. A standard API for on-demand monitoring

The OCCI working group of the Open Grid Forum (OGF) has produced the specifications of an API for the communication between the user and the provider of a Cloud Computing service: the Open Cloud Computing Interface (OCCI). The aim of the OCCI working group is to foster the convergence towards a widely adopted standard.

A distinguishing feature of the OCCI proposal, which differentiates it from other similar standardization initiatives (such as TOSCA from the OASIS [3], or the CIMI from DMTF [1]), can be found in this bottom-up approach: the OCCI core model defines the foundations for an extensible family of standards.

The interface follows the Representational State Transfer (REST) paradigm [12]: the client and the server exchange *request* and *response* messages which operate HTTP verbs on *entities* that describe cloud resources. Each *entity* instance is associated with a *kind*. *Kind*s are arranged in a tree structure to form a sub-typing hierarchy.

The OCCI working group has defined two core *kinds*: the OCCI *resource* — standing for a *cloud* resource — and the OCCI *link*, which represents a relationship between two OCCI *resources*.

An *entity* can be associated with one or more *mixin*s in order to be further characterized by additional *attributes*, other than those defined by its *kind*. The provider is expected to define *mixin*s to give access to specific terms of the service: for instance, the provider might define a *mixin* for a given software package that may be associated with a *compute entity*.

Each *Attribute* is defined by: a **name** in a name-space, a **type**, the indication of its dynamic **mutability**, and whether its definition is **mandatory** or not.

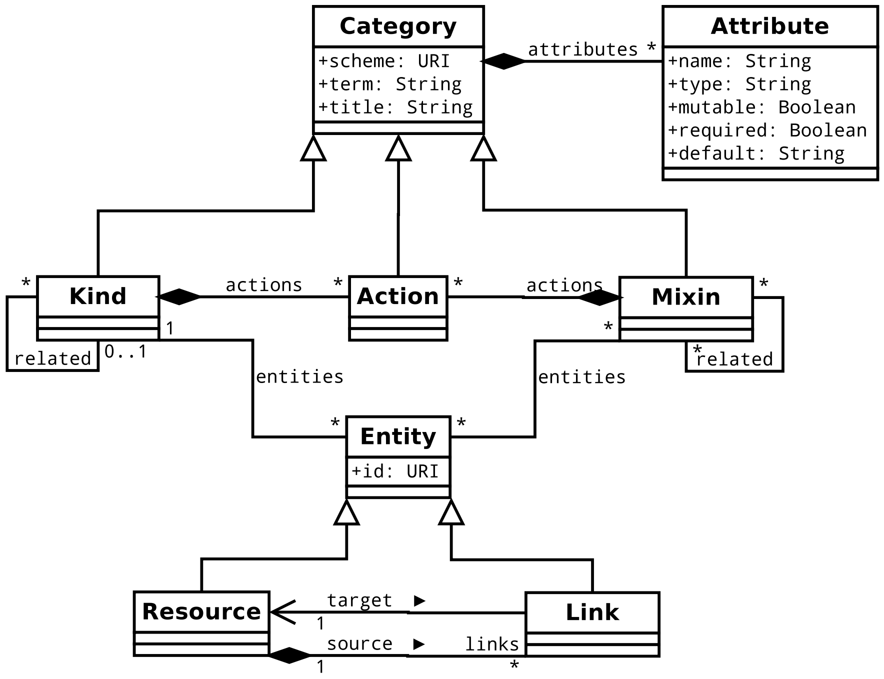

Figure 1: The OCCI core model – A UML class diagram Copyright © Open Grid Forum (2009-2011). All Rights Reserved

Figure 1 shows a UML class diagram that describes the OCCI model, and a full explanation can be found in [19].

Besides the core standard which is designed to be long-lasting as valuable standards need to be, there are satellite documents that customize the core protocol to specific applications. For instance, the *IaaS Extension* defines the entities used to describe a cloud infrastructure, namely *Compute*, *Storage* and *Network Resources*, which are connected using *StorageLink* and *NetworkInterface link*s.

In this paper we consider an *extension* of the OCCI *core* model for a *Monitoring-as-a-Service* API: its class diagram is shown in Figure 2.

2.1. An OCCI extension for on-demand monitoring

The *Monitoring-as-a-Service* extension of the OCCI [5] is layered over the *core* model, so that it is applicable to a generic **aaS* without being associated with a specific type of cloud service. It shares an approach with the rest of the OCCI models which focuses on extensibility, rather than trying to be exhaustive. This contrasts with other standards, both open and proprietary, which tend to be exhaustive, for instance by enumerating the metrics for a given resource such as the CIMI from DMTF. The OCCI approach is complementary to exhaustive standards, since their content can be used to define OCCI **mixins**.

The monitoring extension defines a specialized *resource* — the *Sensor* — that embeds the capabilities to process the measurements and to use the results, but without participating in their production. Thus it is inappropriate to use a *mixin* to associate a *resource* with its current performance. Although straightforward, this solution introduces severe scalability issues, since the value of this attribute is frequently updated, and the user will periodically poll it. Therefore this approach is not flexible, does not scale, and is exposed to security issues. Our alternative consists of de-coupling the production and the processing of the measurements, without abandoning the simplicity.

This approach recalls the introduction of a *monitoring agent*, such as the ones in CompatibleOne and StackDriver. *Nagios*, one of the most popular monitoring tools on the market, follows a similar *separation of concern* paradigm, de-coupling production, processing and notification activities. The same concept is

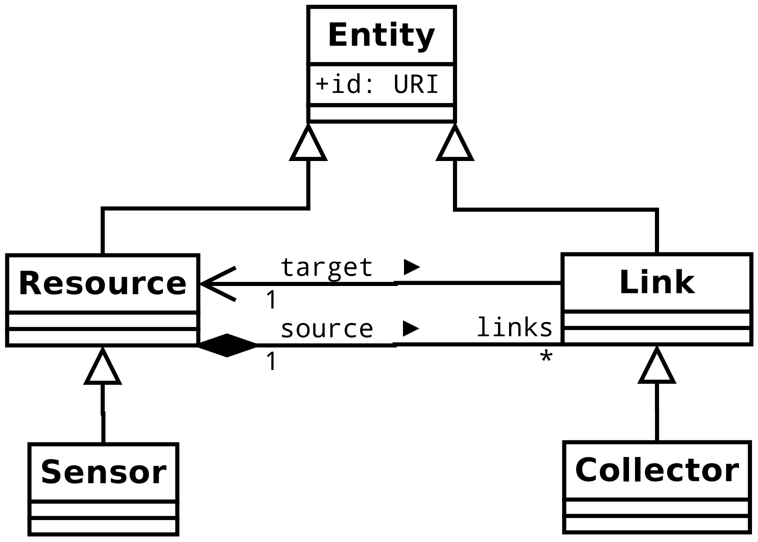

Figure 2: UML class diagram of the monitoring extension

present in the literature, for example [21], and we have successfully implemented a similar concept in a distributed network monitoring prototype [9, 11].

The attributes of a *Sensor* define only the timing of its activity (see Table 1). The specification of the period, a number that indicates the lapse between two subsequent measurements, is mandatory. The time interval during which monitoring takes place is indicated by a base-time, an ISO8601 date, and two time offsets measured from the base-time: one for the start and one for the end of the monitoring activity. These time offsets are optional, such as those that define the granularity of the clock and its accuracy which affect both the coordination of the monitoring and the precision of the time-stamps that are usually attached to measurements.

Model attribute	**value**			
scheme	http://schemas.ogf.org/occi/monitoring#			
term	sensor			
parent	http://schemas.ogf.org/occi/core#resource			
title	sensor resource			
	name	**type**	**mut.**	**req.**
attributes	occi.sensor.period	number	true	true
	occi.sensor.timebase	string	false	false
	occi.sensor.timestart	number	true	false
	occi.sensor.timestop	number	true	false
	occi.sensor.granularity	number	false	false
	occi.sensor.accuracy	number	false	false

Table 1: Definition of the *sensor* kind of the monitoring extension

The user defines the way measurements are aggregated and published by associating the *sensor* with the *mixins* that represent the desired activities: such *mixins* are either supplied by the provider or implemented by the user. The provider is thus free to determine a business strategy by defining the functionalities available: for instance by offering a simple metric for a large infrastructure, or exporting a detailed view of raw metrics. The provider can also allow the user to implement custom *mixins*.

The *Sensor* receives measurements from one or more probes that monitor cloud *Resources*: each of such associations is represented with an OCCI entity of *Collector* type. The *Collector* is a sub-type of the *link* type and, together with the *Sensor* type, they make up the OCCI monitoring extension.

In our model a *Sensor* is the source of several *Collectors* (see Figure 2), which means that the *source* sensor monitors several *target* resources. From a UML perspective, the *Sensor* is *composed of Collectors*.

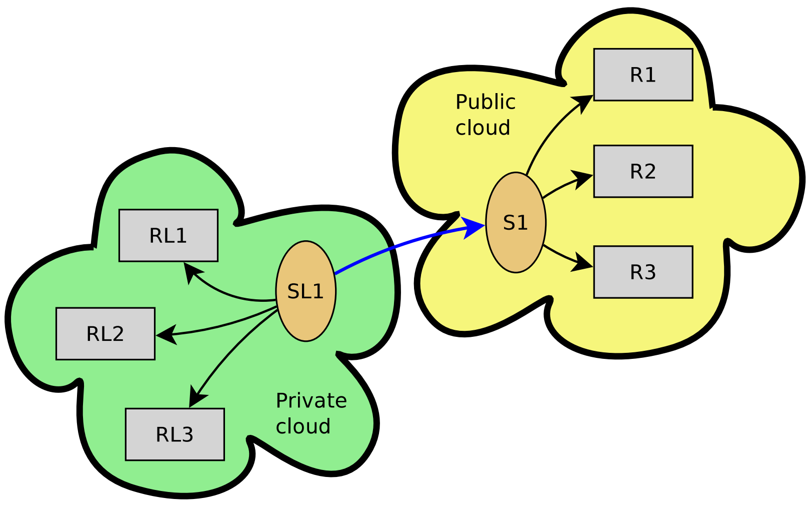

Figure 3: Layout of multi-layer monitoring in a hybrid cloud: the blue link is a cross-provider collector between the sensors SL1 and S1. RL1-3 and R1-3 are resources that are in the private and public clouds, respectively.

The *composed of* relationship is indicated with a diamond arrow in the UML diagram in Figure 2. Therefore if the sensor is deleted, all outgoing collectors are also removed, which is consistent with practice.

In line with the *Sensor*, the attributes of a *Collector* define only its timing (see Table 2), which may be either imported from the controlling *Sensor*, or customized.

The user finalizes the description of a *Collector* with *mixins*: typically each of which defines a metric and a way to measure it. This solution makes it possible to discover the association between a sensor and the resources that this sensor controls, a feature that is also implemented by RackSpace and CompatibleOne.

Model attribute	value
scheme	http://schemas.ogf.org/occi/monitoring#
term	collector
parent	http://schemas.ogf.org/occi/core#link
title	collector link
attributes	

name	type	mut.	req.
occi.collector.period	number	true	false
occi.collector.granularity	number	false	false
occi.collector.accuracy	number	false	false

Table 2: Definition of the *collector* kind of the monitoring extension

The descriptive power of this simple API goes beyond the *measure and notify* schema reported in section 1.1. Note that a *Sensor* is an OCCI *resource*; which is why it can be the target of a *Collector* from a second layer *Sensor*, thus representing a hierarchy of *sensors*. We call this a *multi-stage* monitoring infrastructure, which tends towards multi-tenant infrastructures.

To highlight the impact of this, consider Figure 3: the data collected by the first stage is used by sensor **S1** to control the amount of resources provided by the public cloud (aka auto-scaling). In turn, **S1** is the target of a collector from another sensor **SL1** which performs load balancing in the private cloud, out-sourcing some loads when needed.

The example shows how to address interoperability with a *Collector* linking two *Sensors* hosted by different providers. To play this role, the *Collector* needs to be configured with an appropriate *mixin* to export monitoring data. Since the probe runs on its premises, the provider owning the monitored resources is allowed to mask proprietary or sensitive data: which is a desirable property, as highlighted in section 3.

The two entities defined in the monitoring extension are able to describe very complex cases, including multi-tenancy [21, 24]. However they are *empty boxes* that rely on an assortment of mixins, whose attributes control the engine that implements the user request. Thus an extremely simple interface can be designed that has enough power to describe a complex monitoring infrastructure as a network of *Sensors* interconnected by *Collectors*, whose operation is defined in a second step, by leveraging the extensibility of the OCCI model. This description of a monitoring infrastructure is the converse of another approach that aims at the exhaustive definition of the resource features that need to be measured, overlooking the resources that are used to perform the measurements. Instead, we first define *how* monitoring is performed, and postpone the definition of *what* has to be measured.

The definition of *what* has to be measured is not hardwired into the standard, but defined by the provider using the extensibility of the OCCI schema, which is explored in the next section.

2.2. The monitoring pipe

The *mixins* play a fundamental role in the monitoring extension, allowing the user to define the functionality of the requested entities.

We categorize *mixin*s into three types based on their role: **metric**, **aggregator** and **publisher**. The three types are represented as *tags*, i.e. mixins without attributes, which is the canonical way to introduce typing among OCCI *mixins* (see Table 3). The three *tags* have the following meanings:

- **metric** – is a mixin of the *Collector* link that defines the probe applied to the target *resource*;
- **aggregator** – is a mixin of the *Sensor* resource that defines how raw measurements are treated in order to obtain the desired metric;
- **publisher** – is a mixin of the *Sensor* resource that defines how the metric is rendered or used.

Model attribute	**value**
scheme	http://schemas.ogf.org/occi/monitoring/#
term	metric
applies	http://schemas.ogf.org/occi/monitoring/#collector
scheme	http://schemas.ogf.org/occi/monitoring/#
term	aggregator
applies	http://schemas.ogf.org/occi/monitoring/#sensor
scheme	http://schemas.ogf.org/occi/monitoring/#
term	publisher
applies	http://schemas.ogf.org/occi/monitoring/#sensor

Table 3: The *Mixin* tags defined by the monitoring extension

The tags correspond to the three basic operations that are traditionally associated with a monitoring activity. Together they form a pipeline that processes one or more streams of measurements, from the moment they are produced until they are delivered [23, 10].

Figure 4 shows a three-stage monitoring pipe that mimics the AWS CPU load measurement. First the metric is measured, then a robust estimator is applied, and finally the measurement is delivered as a stream of *UDP* datagrams. We use three distinct mixins, and the resulting attributes for the *sensor* and the *collector* are shown in Table 4. The attributes prefixed by `com.example` are those inherited from the mixins. They describe how the measurement is performed (only *user* CPU cycles), the *gain* of the moving average function (8), and the UDP address where filtered measurements are sent (`dashboard.example.com:8`). We have omitted the *channel* attributes that have not yet been introduced: the complete schemas of the *cpuload* and *udp* mixins are shown in Table 5

In contrast with the above case, where the user consumes the data from the monitoring pipe, there is a monitoring infrastructure where measurements are not published: an example of which is the auto-scale option. The provider enables the user to automatically control the amount of leased resources depending on the load, as in the case of a web server providing an increasingly popular service.

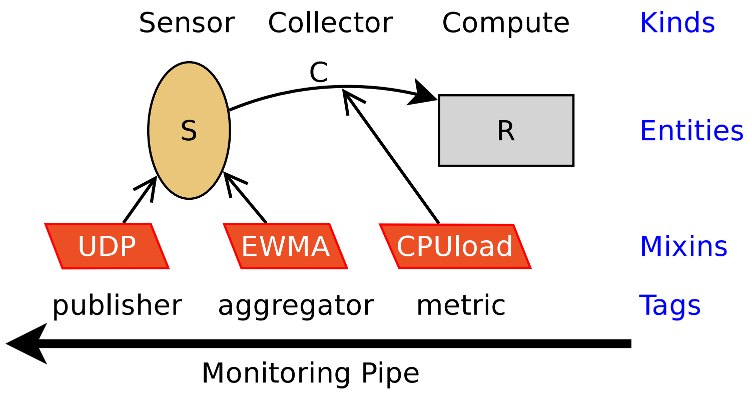

Figure 4: A 3-step monitoring pipe . **S**, **R** and **C** represent a sensor, a generic resource, and a collector that monitors the resource **R** under control of **S**, respectively. The orange diamonds represent the mixins that characterize the associated entities:the relative tags are indicated below each orange diamond.

name	**value**
occi.sensor.period	"60"
com.example.ewma.gain	"8"
com.example.udp.dst	"dashboard.example.com:8989"

`S` sensor attributes

name	**value**
occi.collector.period	"5"
com.example.cpuload.option	"user"

`C` collector attributes

Table 4: The attributes of the *sensor* S and of the *collector* C in Figure 4 (channel attributes are omitted)

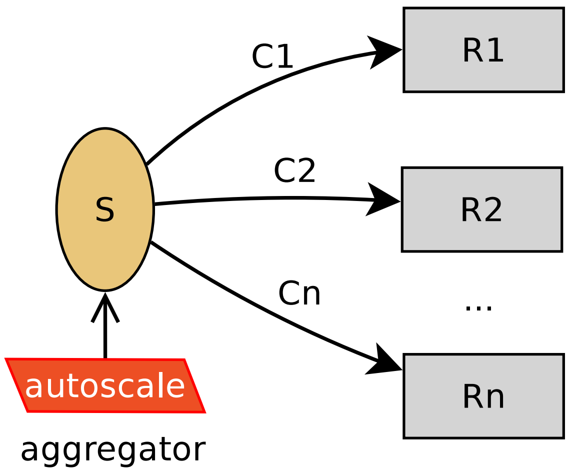

Figure 5: An opaque auto-scaling service . **R1..n**: generic resource; **C1..n**: monitoring collector; **S**: a sensor with associated an *autoscale* aggregator mixin.

If the provider wants to offer this kind of service as a *black box* functionality, the user does not need to detail the parameters used to evaluate the load, and obtains this feature with a yes/no check-box. The architecture that controls this feature is equally simple: a single *aggregator* mixin applied to a sensor, with *collector* links from the sensor to each of the leased resources (see Figure 5). The presence of the mixin together with the *kind* of the controlled resource tells the provider everything needed to implement the service.

When the user clicks on the the auto-scale option for a given resource, the cloud management engine creates the *Collector* links and the *Sensor* resource with an associated *auto-scale* mixin.

The monitoring pipe represents a flow of measures originating from *Metric* mixins associated with *collectors* and traversing one or more *sensors*. Data are processed at each step in the pipe, possibly merged or split, until they are exported or consumed. To represent this data flow we introduce mixin attributes called *channels*.

A *channel attribute* is an OCCI-attribute with a string value that has a special meaning: i.e., it is the label of a uni-directional data stream. Its direction, to or from the entity owning the *channel attribute*, is given in the mixin specifications.

Communication takes place between entities that share the same value for a *channel attribute*, and is restricted within a *scope* consisting of an OCCI *resource* together with the *links* that originate from it. In other words, *channel attribute* labels are relative to a *sensor*. This is consistent with the OCCI Core model [19], which defines a OCCI *resource* as *composed of* (in the UML class diagram) outgoing OCCI *links*, thus implying a strong relationship between a resource and the outgoing links.

2.3. An example

The example in Figure 6 shows a monitoring infrastructure that checks the state of a compute resource **R** and sends the results as UDP datagrams. It is composed of a *sensor* **S** and a collector **C**. The measurements are collected by two distinct *metric* mixins associated with the collector. They are then passed to an *aggregator* that checks the resource overload and forwards the result to a *publisher*, which periodically sends UDP packets to the dashboard. There are two flows of measures, one for the CPU load, another for the network traffic, which are merged into one periodic notification.

The final representations of the two entities are shown in Table 6. The *metric* mixins (`CPUload` and `IPtraffic`, see the schemas in Table 5) are associated with two output channels: `CPUOut` and `IPTrafficOut`. The `2xTh` *aggregator* adds to the sensor two channel attributes: *ThresholdInput1* and *ThresholdInput2*. These two channels receive two input streams which are then processed differently. The connection between *CpuOut*

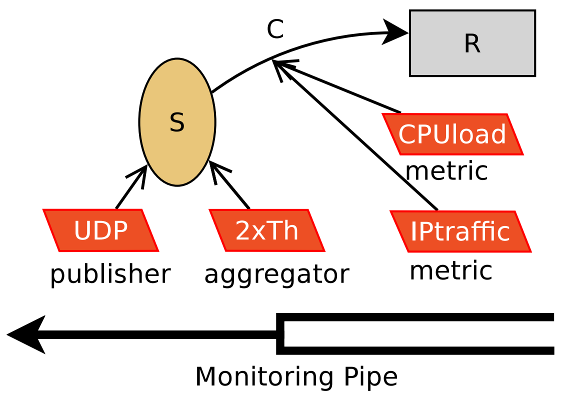

Figure 6: The 3-stage pipe of an overload alarm service: sensor **S** monitors **R** with collector **C**. The first stage of the (metric-aggregator-collector) pipe has two channels that are merged by the aggregator.

Model attribute	value
scheme	http://example.com/#
term	cpuload
related	http://example.com/#metric
attributes	

name	**type**	**mut.**	**req.**
com.example.cpuload.option	string	false	false
com.example.cpuload.out	string	false	true

Model attribute	value
scheme	http://example.com/#
term	iptraffic
related	http://example.com/#metric
attributes	

name	**type**	**mut.**	**req.**
com.example.iptraffic.out	string	false	true

Model attribute	value
scheme	http://example.com/#
term	2xth
related	http://example.com/#aggregator
attributes	

name	**type**	**mut.**	**req.**
com.example.2xth.thresholdinput1	string	false	true
com.example.2xth.check1	string	false	true
com.example.2xth.threshold1	number	false	true
com.example.2xth.thresholdinput2	string	false	true
com.example.2xth.check2	string	false	true
com.example.2xth.threshold2	number	false	true
com.example.2xth.thresholdout	string	false	true

Model attribute	value
scheme	http://example.com/#
term	udp
related	http://example.com/#publisher
attributes	

name	**type**	**mut.**	**req.**
com.example.udp.in	string	false	true
com.example.udp.dst	string	false	true

Table 5: The schemas of the four mixins in Figure 6

name	value
occi.sensor.period	"60"
com.example.2xth.thresholdinput1	"ch:1"
com.example.2xth.check1	"greater than"
com.example.2xth.threshold1	"90"
com.example.2xth.thresholdinput2	"ch:2"
com.example.2xth.check2	"lower than"
com.example.2xth.threshold2	"10"
com.example.2xth.thresholdout	"ch:3"
com.example.udp.in	"ch:3"
com.example.udp.dst	"dashboard.example.com:5656"

`S` sensor attributes

name	value
occi.collector.period	"5"
com.example.cpuload.option	"user"
com.example.cpuload.out	"ch:1"
com.example.iptraffic.out	"ch:2"

`C2` collector attributes

Table 6: The attributes of the entities in Figure 6

and *ThresholdInput1* is indicated by assigning the same *channel id* `ch:1` to the two attributes. The same happens for *IPTrafficOut* and *ThresholdInput2*.

These data are used by the cloud management engine to deploy the monitoring infrastructure. The next section outlines the design and the implementation of this component.

3. Deploying the monitoring infrastructure

The Cloud Management dynamically deploys the monitoring infrastructure following the description provided by the user: which is a similar step to the delivery of IaaS resources. Since the approach is new, we aim to show that it is actually feasible to implement it. This is also important in understanding how to map abstract OCCI entities onto real ICT components.

We suggest a design where the two entities of concern, the *Sensor* and the *Collector*, are implemented as applications that run on cloud resources. We do not target performance results, but highlight that the overall approach has a practical implementation, and that the ontology behind the defined API is sufficiently descriptive. In addition, to better illustrate our design, we implemented an open source prototype [4] using basic technologies, such as raw TCP communication and Java.

The Cloud Management engine obtains the description of the monitoring infrastructure requested by the user from a back-end interface of the Web server that implements the OCCI API. The retrieved entities are those introduced by the OCCI-monitoring extension: *Sensors* and *Collectors*. The Cloud Management engine then dynamically creates the applications that implement the entities.

3.1. The `Sensor` *Application*

The `Sensor` is a component that runs as an independent application on a server. For instance, in our prototype it is run inside a dedicated *Docker*.

The `Sensor` receives the URL from where it retrieves its own OCCI description, and spawns one `CollectorManager` thread for each *Collector* that has the *origin* on the *Sensor* itself. Each `CollectorManager` instructs and configures a `MetricContainer` running on the monitored resource, as explained in Section 3.2.

The `Sensor` thread also creates one `CollectorMultiplexer` thread, which opens a server-side TCP socket, which is the destination of the monitoring data from all originated collectors.

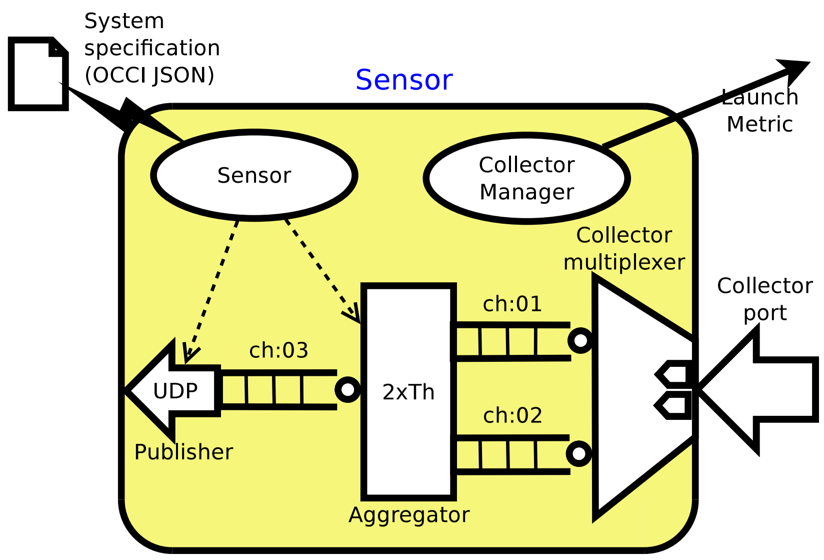

Figure 7: Layout of the implementation of a *Sensor*

The internal structure of a `Sensor` thread is shown in Figure 7.

Sensor *mixin*s are represented as distinct threads whose features depend on the user request. For instance, in our prototype we use Java reflection to dynamically instantiate a thread in the class corresponding to the mixin type indicated by the user. In Figure 7 the `2xTh` and `UDP` boxes represent threads associated with sensor mixins.

The implementation of the channels between threads representing sensor mixins (for instance `ch:03` in Figure 7) depends on the chosen language. In our prototype we use pairs of `PipedInputStream` and `PrintWriter`, collected in two `hash` objects indexed with the channel identifier, which is the string value of the *channel attribute*.

3.2. The collector and the `MetricContainer` *Application*

A *collector* is split into two distinct modules corresponding to its end-points, on the *sensor* and on the monitored resource, respectively.

The *Sensor*-side endpoint of a *Collector* is implemented by the `CollectorMultiplexer` thread. Messages in the TCP stream are encapsulated in JSON Objects that contain the channel identifier. Their content is routed to the destination mixin by the `CollectorMultiplexer`, which exposes a unique TCP port for all *Collectors* originating from the *Sensor*.

The resource-side end-point is implemented with the `MetricContainer`, a daemon thread running on the host where the measurement is carried out. The `MetricContainer` daemon is initially idle, and exposes an interface that the `CollectorManager` running on a `Sensor` uses to control the measurement tools, which are a pool of threads, each performing a separate measurement corresponding to a *metric* mixin attached to the *collector*.

More precisely, using the interface that the `MetricContainer` receives from the `Sensor`

- the description of the *metric* mixins that are to be activated on the resource, and
- the address of a TCP socket on the `Sensor` (call-back style)

In our Java prototype we make use of the RMI protocol for this interface, and use Java reflection to instantiate a new *metric* mixin of a given type.

In [6] there is a detailed description of a Docker-based demo based on the Java engine described above. The virtual architecture of the demo is illustrated in Figure 9, where the **Server** virtual machine responds

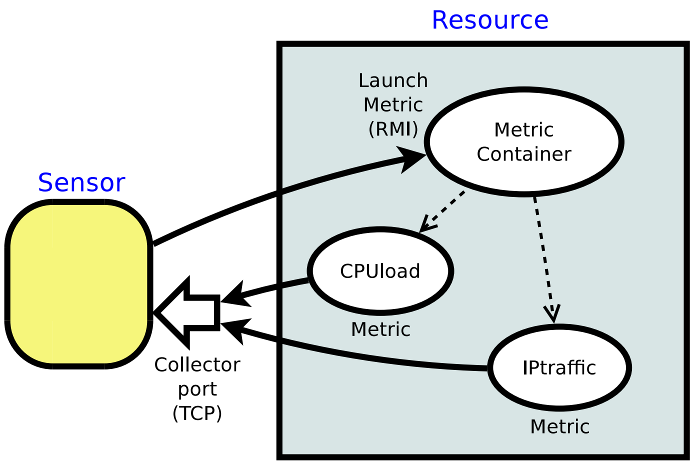

Figure 8: Layout of a *Collector*

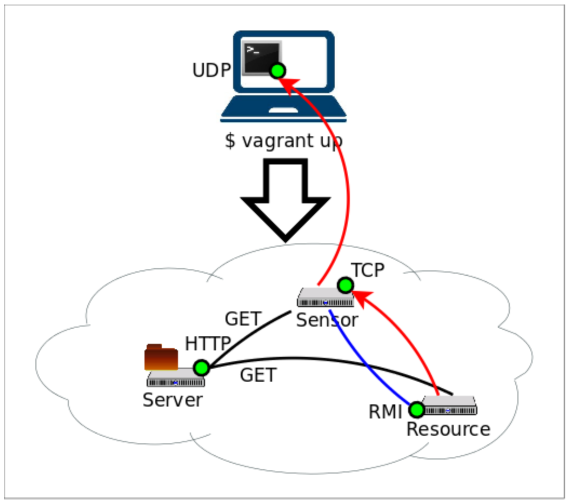

Figure 9: The virtual architecture deployed by the Docker-based demo. The green circles are Internet ports for the indicated protocol

to GET requests providing the JSON description of OCCI entities (black links). The **Sensor** configures the monitoring activity on the probe using the RMI interface on the *Resource* (green link). When the infrastructure is ready, monitoring results flow from the resource to the **TCP** port of the `CollectorMultiplexer` on the sensor, where they are filtered and forwarded to the **UDP** port of the dashboard (red arrows). The pre-loaded infrastructure is similar to the one in Figure 6, however the user can modify the entities recorded on the **Server** and add new resources to implement a different one. It is also possible to create new mixins for the *Sensor* and for the *Collector*. The Docker images are available from [7].

4. Conclusions

Resource monitoring is a useful aspect of a *cloud* service. In recent years it has evolved from the bare delivery of log-like measurements to a service that is controlled through a specific API. However the user interface with the monitoring system is rigid and provider-specific.

This paper addresses these two limitations with a new design paradigm that introduces:

- a flexible component-based framework for the design of the monitoring infrastructure;
- a representation of that framework using standard OCCI entities.

The adherence to the OCCI standard is a first step towards an open and interoperable interface for cloud monitoring. The interface is defined as an extension of the OCCI core model, where resource-specific details are unspecified, while timing plays a fundamental role. Since the model is resource-agnostic, it is a stable foundation for further developments.

The OCCI monitoring model describes a component-based framework which is one step towards a principled design of on-demand monitoring services. It has a light design footprint for simple scenarios, but with the ability to scale to complex, multi-tenant environments. We have shown through examples that our model, based on only two additional kinds of entities, can address the challenges of monitoring a hybrid cloud, while providing a straightforward description of an elastic Web server.

To provide concrete evidence, we designed an engine that transforms the representation into a real deployment, and demonstrated its soundness with a Docker-based prototype. Our goal is not performance but simplicity, with a design that we believe would be useful as a guideline for production-grade implementations.

5. Bibliography

[1] Cloud infrastructure management interface (CIMI) model and RESTful HTTP based protocol an interface for managing cloud infrastructure, October 2013.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[3] Tobias Binz, Uwe Breitenbcher, Oliver Kopp, and Frank Leymann. Tosca: Portable automated deployment and management of cloud applications. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Advanced Web Services*, pages 527–549. Springer New York, 2014.

[4] Augusto Ciuffoletti. Java implementation of a cloud monitoring manager. `https://bitbucket.org/augusto_ciuffoletti/occi-monitoring`.

[5] Augusto Ciuffoletti. A simple and generic interface for a cloud monitoring service. In *CLOSER - 4th International Conference on Cloud Computing and Services Science*, pages 143–150, 2014.

[6] Augusto Ciuffoletti. Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science*, 68:163 – 172, 2015. 1st International Conference on Cloud Forward: From Distributed to Complete Computing.

[7] Augusto Ciuffoletti. On demand monitoring using OCCI: the demo. `https://hub.docker.com/r/mastrogeppetto/occimon-live/`, 2015.

[8] Augusto Ciuffoletti. *Open Cloud Computing Interface - Monitoring.* Open Grid Forum, June 2015.

[9] Augusto Ciuffoletti, Tiziana Ferrari, Antonia Ghiselli, and Cristina Vistoli. Architecture of monitoring elements for the network element modeling in a grid infrastructure. In *Proc. of Workskop on Computing in High Energy and Nuclear Physics*, La Jolla (California), March 2003.

[10] Augusto Ciuffoletti, Yari Marchetti, Antonis Papadogiannakis, and Michalis Polychronakis. Prototype implementation of a demand driven network monitoring architecture. In Sergei Gorlatch, Paraskevi Fragopoulou, and Thierry Priol, editors, *Grid Computing - Achievements and Prospects*, volume 9, chapter 8, pages 85–97. Springer, 2008.

[11] Augusto Ciuffoletti and Michalis Polychronakis. *Architecture of a Network Monitoring Element*, volume 4375 of *Lecture Notes in Computer Science*, chapter 2, pages 4–14. Springer, 2007.

[12] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[13] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems. *Network, IEEE*, 7(6):20–30, Nov 1993.

[14] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing the business perspective. *Decision Support Systems*, 51(1):176 – 189, 2011.

[15] M. Mohamed, D. Belaid, and S. Tata. Monitoring and reconfiguration for OCCI resources. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 539–546, Dec 2013.

[16] Jess Montes, Alberto Snchez, Bunjamin Memishi, Mara S. Prez, and Gabriel Antoniu. Gmone: A complete approach to cloud monitoring. *Future Generation Computer Systems*, 29(8):2026 – 2040, 2013.

[17] Audrey Moui and Thierry Desprats. Towards self-adaptive monitoring framework for integrated management. In Isabelle Chrisment, Alva Couch, Rmi Badonnel, and Martin Waldburger, editors, *Managing the Dynamics of Networks and Services*, volume 6734 of *Lecture Notes in Computer Science*, pages 160–163. Springer Berlin Heidelberg, 2011.

[18] Mirella Muhic and Bjrn Johansson. Cloud sourcing next generation outsourcing? *Procedia Technology*, 16:553 – 561, 2014.

[19] OGF. *Open Cloud Computing Interface - Core.* Open Grid Forum, June 2011. Available from www.ogf.org. A revised version dated 2013 is available in the project repository.

[20] OGF. *Open Cloud Computing Interface - Infrastructure.* Open Grid Forum, June 2011. Available from www.ogf.org.

[21] Javier Povedano-Molina, Jose M. Lopez-Vega, Juan M. Lopez-Soler, Antonio Corradi, and Luca Foschini. Dargos: A highly adaptable and scalable monitoring architecture for multi-tenant clouds. *Future Generation Computer Systems*, 29(8):2041 – 2056, 2013.

[22] T. Reimer and Qing Tan. Ecosystem for business driven it management. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–6, May 2014.

[23] Michael Smit, Bradley Simmons, and Marin Litoiu. Distributed, application-level monitoring for heterogeneous clouds using stream processing. *Future Generation Computer Systems*, 29(8):2103 – 2114, 2013.

[24] D. Tovarnak and T. Pitner. Towards multi-tenant and interoperable monitoring of virtual machines in cloud. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 436–442, Sept 2012.

[25] Salvatore Venticinque, Alba Amato, and Beniamino Di Martino. An OCCI compliant interface for IaaS provisioning and monitoring. In Frank Leymann, Ivan Ivanov, Marten van Sinderen, and Tony Shan, editors, *3rd International Conference on Cloud Computing and Services science (CLOSER)*, pages 163–166. SciTePress, 2012.

[26] M. Vierhauser, R. Rabiser, P. Grunbacher, C. Danner, S. Wallner, and H. Zeisel. A flexible framework for runtime monitoring of system-of-systems architectures. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 57–66, April 2014.

[27] Deqing Zou, Wenrong Zhang, Weizhong Qiang, Guofu Xiang, Laurence Tianruo Yang, Hai Jin, and Kan Hu. Design and implementation of a trusted monitoring framework for cloud platforms. *Future Generation Computer Systems*, 29(8):2092 – 2102, 2013.