GWD-C-I
OCCI-WG
augusto@di.unipi.it

Augusto Ciuffoletti, Dept. Of Computer Science -
Univ. of Pisa
July 25, 2013

Describing a monitoring infrastructure with an OCCI-compliant schema

Status of This Document

Group Working Draft (GWD)

Document Change History

Febraury 1st, 2013: First revision (Augusto Ciuffoletti)

June 12, 2013: Second revision (Augusto Ciuffoletti):

- Set document label to indicate Informational
- Mixins may be associated to all Entity subtypes (not only Sensors or Collectors) for very simple layouts.
- For the same reason measurements may be included in REST resource representation.
- Example without HTTP rendering.
- Publishing moved from Collector to Sensor.
- Changed names of *mixins* (ToolSet to Metric, the others removing "Set")

June 21, 2013: Minor revision (Augusto Ciuffoletti):

- Modified definition of scope, focus on links, not attributes;
- Refined definition of value for metric attributes.
- Bugs fixed in examples

July 25, 2013: Extension (Augusto Ciuffoletti):

- the SLA appendix has been extended
- the "Related Works" section has been extended
- the UML class diagram has been added
- minor fixes

Copyright Notice

Trademark

Abstract

This document *provides information* to the Grid community about resource monitoring. It *describes* an OCCI Extension that allows to inspect the operation of functional resources; the provision of this API is considered as optional for the provider.

This document *presents* two further *Kinds*: the *Sensor Resource*, that processes metrics, and the *Collector Link*, that extracts and transports metrics. They are defined as OCCI types whose instances need to be specialized using OCCI *mixin*s. Using this API, the user is provided with a monitoring infrastructure *on demand*.

This document does not define any standards or technical recommendations.

One relevant target of this document is to provide a building block for the design of an API for Service Level Agreement (SLA): under this light, the API for the Resource Monitoring Infrastructure offers the tools to verify and implement the Service Level Objectives (SLO).

Contents

1 Introduction

This document describes an interface useful to define a monitoring infrastructure. It is based on the concepts introduced by the OCCI Working Group of the OGF, and it is intended to be a first step towards the definition of a protocol to measure service quality: its applicability extends to fault detection, billing, and the implementation of a server level agreement (SLA).

The purpose of this specification is that of giving the user the possibility to arrange a monitoring infrastructure in the way that best suits user's needs: notably, the existence of a standard specification enables the user to manage distinct cloud providers, possibly at the same time, using the same interface.

The importance of a configurable monitoring infrastructure emerges in many scenarios, starting from the simple case of the user that wants to monitor the activity of an array of servers, to composite use cases, where the user is in fact an intermediate service provider, that provides services to third party users in a multi-provider environment: in that case, the intermediate provider may decide to offer quality of service options that differ from that of the low level provider, thus needing to perform specific measurements on the infrastructure leased by the low level provider(s). The tools provided by an API that describes a monitoring infrastructure must be flexible to meet all degrees of complexity.

The communication of measurements inside and outside the monitoring infrastructure is another issue the framework must be flexible about. In fact, the amount of information that is produced by a measurement activity may range from negligible to "big data" dimensions, with various degrees of confidentiality. Also in this respect, guidelines must be as permissive as possibile, to give the provider the possibility to apply the solution that better fits the needs.

The management of the monitoring capabilities should also extend to the adaptive, and dynamic configuration of the components that contribute to the monitoring activity: the specification schema must give the user the possibility to explore the available functionalities in order to adaptively arrange a monitoring infrastructure, and to modify them according with changing needs.

One relevant fact about monitoring infrastructures is that it is extremely difficult to give a *detailed* framework for them that extends its validity to any conceivable use case or provider. The reason is that each of them exhibits local variants that do not fit a rigid approach. Also, the metrics that are used to evaluate the performance of the system are many, and subject to continuous changes due to the introduction of new

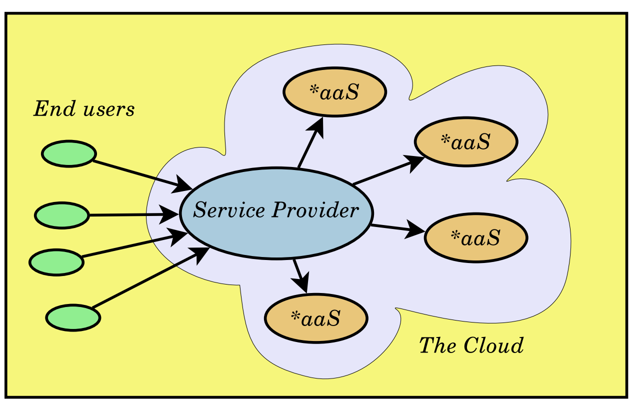

Figure 1: A multilayer and multiprovider scenario

technologies. Thus we have made an effort to introduce a generic schema that can be adapted so to effectively describe the relevant aspects of a monitoring infrastructure, but that does not interfere with details that depend on the specific environment.

The OCCI Core Model [OGF(2011a)] is well suited to support an extremely flexible framework, since it embeds the tools needed to avoid to get into provider specific details: this enables the specification of the abstract model, leaving to the user the task of making explicit the details, targeting a specific provider or technology. Furthermore, we claim that the specifications given in this document can find an application in environments other than computing infrastructures, since we abstract from the details that characterize cloud infrastructure resources.

The approach followed in this document is similar to that found in the infrastructure document (GFD-P-R.184 [OGF(2011b)]): the monitoring capability is associated with a new *Kind* instance, the *Collector*, that is related with the OCCI Core Model *Link* type. The source of the *Collector Link* is the monitored resource that originates measurements that are delivered to the *target* resource. The role of a *Collector Link* instance is to indicate a specific monitoring technique applied on the *source*. The processing of the measurements and their delivery are described by the *Sensor kind*, that is related with the OCCI Core Model *Resource* type. A *Sensor Resource* instance collects metrics across a *Collector Link* and publishes aggregated metrics with a defined modality: for instance a *Sensor Resource* might produce the average load of an array of servers and publish it on a web page.

The three aspects of monitoring that we have thus outlined – namely production, processing, and publishing – are specified through the association of specific *mixin*s, primarily with a *Sensor Resource* or *Collector Link*. For the sake of simplicity, we introduce the possibility to associate such mixins to ordinary resources: this option is regarded as a tradeoff between simplicity and adherence to the REST approach, as explained in the appendix.

The encapsulation of technology dependent features into *mixin*s leaves the specific provider free to introduce specific supporting technologies, or to simplify the configuration with the provision of templates. To enable the discovery of such *mixins*, they are related with a *depends* relationship with well-known *mixin*s.

Although the interface based on *Sensor Resource*s and *Collector Link*s may describe very simple use cases with minimal effort, the designer is able to assemble complex, multilayer monitoring infrastructures using the same basic building blocks: for instance, a *Sensor Resource* can be used to aggregate a storage throughput using the input from three *Collector Link*s, one for the average response time, one for the mean time between failures, and another for network delay, and provide the results to an upstream *Sensor Resource* that aggregates the same results from other *Sensor Resource*s.

We point out that the interface is transparent to the existence of a standard for metric identifiers: if one exists, the interoperability of distinct monitoring infrastructures is certainly improved. We consider that the user that interacts with the monitoring infrastructures either knows about the identifiers used by the provider, or uses an interface (e.g., a SLA negotiation service) that translates provider specific identifiers into interoperable ones. This document highlights other similar standardization issues.

This document considers the monitoring activity as a time-related activity: as a matter of fact, the only characterization we introduce for a monitoring activity is the period with which samples are collected. This excludes from the range of applicability of this document those cases that envision the event driven inspection of a component. We consider that the availability of event driven inspectio has relevant use cases, but has aspects that differentiate from what is commonly expected from a monitoring framework.

The relevance of the timing pediod and its utilization varies from case to case: at one end there is an

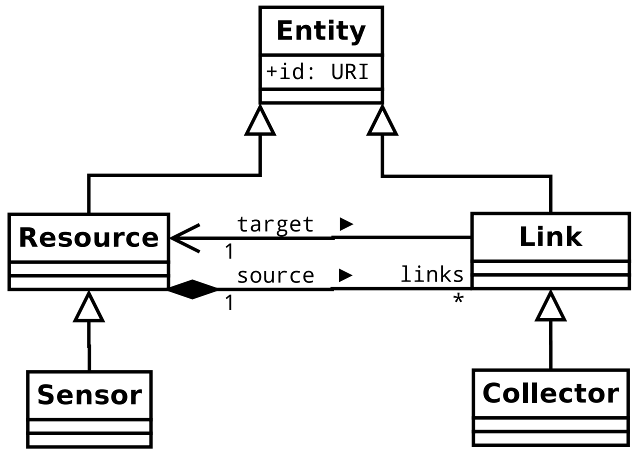

Figure 2: Class inheritance diagram for OCCI-monitoring

application that uses as a reference to compute the real timing, in a IGMP-like way. In other cases this indication has a real-time aspect: thus we complement the definition of the period with a granularity and an accuracy that binds the precision of the clock used to compute the period.

Similarly, the start of the monitoring activity is time-driven: the monitoring activity is scheduled as a task with a start and an end time. Similarly to the case of the period, the clock used to this timing may be subject to accuracy and granularity constraints.

The definition of the timing is dynamic, and can be modified while the monitoring is in effect. Thus the dynamic adaptation of the timing of the monitoring activity can be explicitely controlloed via the attributes of the monitoring resorces.

Summarizing, the specifications introduced in this document require that the conformant provider implements two *Kind*s: the Collector Link and the Sensor Resource. Three generic *mixin*s are also defined to enable the classification of *mixin*s that are specific for the provider: namely *Metric* to specify the production of measurement, *Aggregator* for their processing, and *Publisher* for their publication. The generic *mixin*s are used to identify and apply restrictions to provider-specific *mixin*s, for the sake of interoperability. Given that monitoring is recognized as a basic building block for the introduction of a SLA, we have dedicated an appendix to the discussion of how the API may contribute to this.

1.1 Terminology shortcuts

To distinguish an *Entity* instance from the related *Kind*, we will use the indeterminative article for the instance (e.g., “a *Resource*”), and the determinative article for the *Kind* (e.g., “the *Resource*”). The plural is reserved to instances (e.g., “the *Resources*”). In case of ambiguity we will use “*Entity* instance related with” or “*Kind*”.

Similarly, we will use the term *a <mixin id>* mixin to indicate a *mixin* that *depends* on the *<mixin id> mixin*. The provider ensures that *mixin*s inherit defined semantics from the *mixin* they depend on, as explained in the rest of this paper.

To disambiguate the usage of the term ”resource” we will use the term ”REST resource” when appropriate, and simply *Resource* when referring to the concept defined in the OCCI Core model.

<table>
<tr><th>Model attribute</th><th>value</th></tr>
<tr><td>scheme</td><td>http://schemas.ogf.org/occi/monitoring#</td></tr>
<tr><td>term</td><td>collector</td></tr>
<tr><td>title</td><td>Collector Link</td></tr>
<tr><td>attributes</td><td>
<table>
<tr><th>name</th><th>type</th><th>mutable</th><th>required</th><th>Description</th></tr>
<tr><td>occi.collector.period</td><td>number</td><td>true</td><td>true</td><td>The time between two following measurements</td></tr>
<tr><td>occi.collector.periodspec</td><td>string</td><td>true</td><td>false</td><td>granularity, accuracy, exponent of period measument</td></tr>
</table>
</td></tr>
</table>

Table 1: Definition of the *Collector Link* Kind

2 Specification of the compliant server

The compliant server MUST define the following *Kind*s:

Collector Link that describes the extraction of measurements (see table 1);

Sensor Resource that describes how measurements are aggregated and used (see table 2).

In addition, the compliant server MUST define the following *Mixin* tags (see table 3):

Metric that is used as a tag for *mixin*s that describe a measurement activity;

Aggregator that is used as a tag for *mixin*s that describe a measurement aggregation function;

Publisher that is used as a tag for *mixin*s that describe measurement utilization.

These tags are used to define features that are common to the *mixins* that share the same tag. In principle, a *mixin* may be related with more than one of these tags.

2.1 The *Collector Link*

The *Collector Link* (see table 1) models the activity that extracts measurements from a source *resource* and the transfer of such measurements to another *resource*.

The OCCI attributes of the *Collector Link* define the timing of the monitoring activity. The execution rate is defined using three attributes: the rate itself (`period`), and an optional definition of the quality of the timing. This latter attribute (`periodspec`) contains a triple of numbers encoded as a string, that define the granularity with which the rate is measured, the accuracy of rate measurement, and the floating point exponent. By default `periodspec="undef, undef, 0"`. All time values are represented as numbers.

A *Collector Link* can be related ONLY with *metric mixins*.

2.2 The *Sensor Resource*

The *Sensor Resource* (see table 2) models the processing of the measurements, like their aggregation in composite metrics, as well as their effects outside the monitoring infrastructure, like their delivery to a billing office.

A *Sensor Resource* is characterized by OCCI attributes that define the rate with which new observations are produced, and by the scheduling times of its operation.

Model attribute	value				
scheme	http://schemas.ogf.org/occi/monitoring#				
term	sensor				
title	*Sensor Resource*				
	name	**type**	**mutable**	**required**	Description
attributes	occi.sensor.period	number	true	true	The time between two following measurements
	occi.sensor.periodspec	string	true	false	granularity, accuracy, exponent of period measument
	occi.sensor.timebase	number	false	true	The server time when the timestart and timestop are modified
	occi.sensor.timestart	number	true	true	The delay after which the session is planned to start
	occi.sensor.timestop	number	true	true	The delay after which the session is planned to stop
	occi.sensor.timespec	string	true	false	granularity, accuracy, exponent of time measurement

Table 2: Definition of the *Sensor Resource* Kind

Term	**Scheme**	**Title**
metric	http://schemas.ogf.org/occi/monitoring/collector#	A measurement tool
aggregator	http://schemas.ogf.org/occi/monitoring/sensor#	An aggregation algorithm
publisher	http://schemas.ogf.org/occi/monitoring/sensor#	A sensor Resource

Table 3: The *Mixin* tags defined for the monitoring API

The execution rate is defined using two attributes: the rate itself, and an optional definition of the quality of the timing. This latter attribute contains a triple of numbers encoded as a string, that define the granularity with which the rate is measured, the accuracy of rate measurement, and the floating point exponent. By default `periodspec="undef, undef, 0"`.

The activation of a *Sensor Resource* is controlled by two attributes that describe the scheduling of sensor activity: to schedule the execution of a sensor the user modifies the `starttime` with a value indicating how far in the future the instance is going to start its activity. A value of zero corresponds to the immediate start. The server sets the `timebase` attribute corresponding to the reference time of the start time.

All time values are represented as numbers. The `timebase` corresponds to Unix seconds, all timing values use a floating point notation. Also for time values there is a `timespec` attribute analogous to `periodspec`.

A *Sensor Resource* can be related ONLY with *Aggregator* and *Publisher mixins*.

A *Sensor Resource* is the target of *Collector Link* links, and all of them (*Sensor Resource* included) are collectively indicated as the *scope* of that *Sensor Resource*. The *scope* has the role to specify the visibility of measurements before they are delivered outside the monitoring infrastructure. Consider that a generic *Collector Link* instance belongs to exactly one scope, so that we can speak as well of *the* scope of a *Collector Link*.

<table>
<tr><th>Model attribute</th><th>value</th></tr>
<tr><td>scheme</td><td>http://acme.com/monitoring#</td></tr>
<tr><td>term</td><td>iostat</td></tr>
<tr><td>related</td><td>http://schemas.ogf.org/occi/monitoring#metric</td></tr>
<tr><td>attributes</td><td><table>
<tr><th>name</th><th>type</th><th>Description</th></tr>
<tr><td>com.acme.iostat.cpuutilization</td><td>String</td><td>metric</td></tr>
<tr><td>com.acme.iostat.what</td><td>enum(user,system)</td><td>control</td></tr>
</table></td></tr>
</table>

Table 4: Example – Definition of the `iostat` metric *mixin*

2.3 Features of the *mixin*s that have the *metric* tag

A *mixin* with a *metric* tag represents the availability of measurements from the associated *Entity*. We envision two distinct cases, the latter justified to allow extremely simple configurations (see Section 3):

- if the related *Entity* is a *Collector Link*, the measurement activity refers to the source *Resource* of the *Collector Link*;
- otherwise the measurement activity refers to the related *Entity* sub-type instance itself;

In principle, each provider may associate a different semantic to *mixin*s that share the same identifier, so here there is ground for further standardization. If the provider does not adhere to a defined standard, it MUST give an exhaustive documentation of the the monitoring tool associated with a certain *mixin*.

The OCCI attributes of a *mixin* related with the *metric* tag are divided into two groups:

- Metric attributes: they correspond to the delivered metrics. We distinguish two cases for the semantic of the *String*:
  - it may represent the real value of the measurement (*here* mode), or
  - it may be a generic identifier, unique in the *scope* of the Entity it is associated with (*named* mode);

  For instance, a metric providing the CPU utilization of a *Compute Resource* may have an attribute named `cpuutilization`. Depending on the definition of the *metric* mixin, at a certain time the value may reflect the last measurement, or a local identifier for the data set;
- Control attributes: they control the operation of the measurement activity. For instance the `iostat` *mixin* implementing a cpu utilization tool may have a control attribute defined as (see figure 4)

  ```
  name=com.acme.iostat.what,
  type=enum{"user","system"}
  ```

  The role of the attributes is part of the specification of the specific *mixin*[1]

To enable interoperabilty, the provider SHOULD follow a defined standard for the naming of metric and control attributes, but its specification falls outside the scope of this document. Such naming MAY help the discovery of *mixin*s that are appropriate for a given task.

[1] In the definition of OCCI attributes we have sometimes omitted the model attributes **mutable**, **required** and **default** for typographical reasons.

<table>
<tr><th>Model attribute</th><th>value</th></tr>
<tr><td>scheme</td><td>http://acme.com/monitoring#</td></tr>
<tr><td>term</td><td>max</td></tr>
<tr><td>related</td><td>http://schemas.ogf.org/occi/monitoring#aggregator</td></tr>
<tr><td>attributes</td><td><table><tr><th>name</th><th>type</th><th>Description</th></tr><tr><td>com.acme.max.max</td><td>number</td><td>metric</td></tr><tr><td>com.acme.max.data</td><td>String</td><td>input</td></tr></table></td></tr>
</table>

Table 5: Example – Definition of the `max` aggregation *mixin*

2.4 Features of the *mixin*s that have the *aggregator* tag

A *mixin* with the *aggregator* tag is meant to implement the computation of an aggregated metric starting from raw metrics. It is intended to complement a *Sensor Resource*, but it can be associated also to a generic *Entity* instance, to meet very simple use case, as explained in the appendix.

In principle, each provider may associate a different semantic to *mixin*s that share the same identifier, so here there is ground for further standardization. If the provider does not adhere to a defined standard, it MUST give an exhaustive documentation of the the aggregation algorithm associated with a certain *mixin*.

The attributes of a *mixin* with the *Aggregator* tag are divided into three groups:

- Input attributes: they bind an input of the aggregating algorithm with one of the *named* metric attributes defined in the scope of the *Sensor*. The binding is implemented using the identifier associated to the specific *named* metric in the *metric* mixin. For instance, a *Sensor Resource* that computes the maximum CPU utilization for three *Compute Resources* may have an `input` attribute like the following:

  name=com.acme.max.input,
  input= "cpu-c1, cpu-c2, cpu-c3"

  where `cpu-c1`, `cpu-c2`, and `cpu-c3` are three identifiers defined in metric attributes in *Sensor Resource* scope. We do not introduce a syntax for such identifiers in this document.

- Control attributes: they control the operation of the aggregating function (for instance, the gain of an EWMA);
- Metric attributes: they correspond to the metrics delivered, and are defined like those of *Collector Link*s.

To enable interoperabilty, the provider SHOULD follow a defined standard for the naming of input, control and result attributes, but its specification falls outside the scope of this document. Such naming MAY help the discovery of *mixin*s that are appropriate for a given task.

2.5 Features of the *mixin*s with the *Publisher* tag

How data are delivered is defined by a *Publisher mixin*.

In principle, each provider may associate a different semantic to *mixin*s that share the same identifier, so here there is ground for further standardization. If the provider does not adhere to a defined standard, it MUST give an exhaustive documentation of the publishing mode associated with a certain *mixin*.

Examples of measurement delivery modes are: through a Unix pipe, on demand through a TCP connection, pushed through HTTP or UDP, persistently recorded in a database. However a *Publisher* can be associated

Model attribute	value
scheme	http://acme.com/monitoring#
term	tcp
related	http://schemas.ogf.org/occi/monitoring#publisher
attributes	**name** / **type** / Description: com.acme.tcp.in / URI / input; com.acme.tcp.source / URI / control; com.acme.tcp.port / number / control

name	type	Description
com.acme.tcp.in	URI	input
com.acme.tcp.source	URI	control
com.acme.tcp.port	number	control

Table 6: Example – Definition of the `tcp` publishing *mixin*

also with an activity outside the monitoring infrastructure, like triggering recovery strategies in case of failure.

The attributes of a *Publisher mixin* are divided into two groups:

- Input attributes: they are defined like those of the *Aggregator*;
- Control attributes: they determine the process used to publish input attributes.

To enable interoperabilty, the provider SHOULD follow a defined standard for the naming of input and control attributes, but its specification falls outside the scope of this document. Such naming MAY help the discovery of *mixin*s that are appropriate for a given task.

2.6 Constraints on the associations between instances and *mixin*s

The constraints on the association of *Sensor Resource* and *Collector Link* instances with the defined *mixins* are the following:

- a *Sensor Resource* MUST be the *target* of at least one *Collector Link*;
- a *Collector Link* can be associated ONLY with *metric mixin*s;
- a *Sensor Resource* can be associated ONLY with *publisher* and *aggregator mixin*s;

3 Addressing simple environments

We consider as a relevant issue the possibility to address simple use cases with a minimal effort. The provider should keep control over the application of simplified strategies, since in most cases they trade off efficiency for simplicity, so that the inappropriate application of a simple strategy may overload the provider infrastructure.

The basic tool available to arrange a monitoring activity with minimal effort is the *here* metric *mixin*(see Section 2.3). Using such a *mixin* the user bypasses measurement aggregation and publishing. However the resulting design is not fully REST compliant. An example is given in the appendix.

Another simplifying strategy consists in associating the *mixin* to *Entities* that are not those specified in this document, thus avoiding the presence of *Sensor Resource* and *Collector Link* instances. Certain monitoring specific *mixin*s can be applied directly to *Entities*: for instance an infrastructure *Compute* instance can take the role of a *Sensor* being associated with an appropriate *mixin*. In this case the provider looses control over measurement data, and cannot apply optimization and security strategies that are specific for monitoring data. An example is given in the appendix.

4 Conformance profiles

The definition of conformance profiles is appropriate because the provision of an interface for the management of a monitoring infrastructure is optional.

Profile 0 The *Collector Link* and *Sensor Resource Kind* s MUST NOT be implemented: attempt of instantiating such *Kinds* fails. In an HTTP rendering a POST and GET over the corresponding URI returns `404 Notfound`. The *Aggregator*, *Metric*, and *Publisher mixin*s MUST NOT be implemented: discovery fails. In an HTTP rendering a GET over the *mixin* returns `404 Notfound`;

Profile 1 The *Collector Link* and *Sensor Resource Kind* s MUST be implemented, and the user MUST be allowed to create new instances of such *Kinds*. In an HTTP rendering a POST or a GET over the corresponding URI return respectively `201` and `200`. In case of error, the server MUST NOT return `404 Notfound`. The *Aggregator*, *Metric*, and *Publisher mixin* MUST be implemented, and discovery is successful. The server MUST NOT allow to introduce *depends* relationships with the *Aggregator*, *Metric*, and *Publisher mixin*s. In an HTTP rendering, a POST over their URIs returns `405 Method Not allowed`;

Profile 2 The *Collector Link* and *Sensor Resource Kind* s MUST be implemented, and the user MUST be allowed to create new instances of such *Kinds*. In an HTTP rendering a POST and GET over the corresponding URI returns respectively `201` and `200`. In case of error, the server MUST NOT return `404 Notfound`. The *aggregator*, *metric*, and *publisher mixin*s MUST be implemented, and discovery is successful. The user MUST be allowed to introduce *depends* relationships with the *aggregator*, *metric*, and *publisher mixin*s. In an HTTP rendering, a POST over their URIs returns `200`.

5 Related works

The topic of Cloud Service Level Agreement has been extensively studied in a number of research projects, and there are results that have an impact on Cloud Monitoring: a report [**?**] of the European Union illustrates the results in the field. We have taken input from this activity in the design of this API.

In particular, we considered the need of taking into account the presence of *composite services* encompassing several providers for their implementation. Tightly related ot this is the need of representing multilevel monitoring infrastructures, a fact anticipated by the results of the SLA@SOI EU project. The results of the *Stream* project highlight how monitoring data may introduce *big data* issues, that need specific, flexible solutions on the provider's side: this is one of the reasons that induced the introduction of opaque *connectors* that hide sophisticated, rapidly evolving technologies. The IRMOS project puts an accent on timing requirements for multimedia services, that justify the attention we paid to define and qualify timing attributes. Many project made precise statements about the specific metrics that describe the Service Level for specific types of resource: the mPlane project addresses specifically the metrics for Network resources, while Cloud4SOA focusses on the relevance of an agreed set of standard metrics.

In the design of this API we also took advantage of the experience gained during the CoreGRID EU-project [Ciuffoletti et al.(2008)Ciuffoletti, Marchetti, Papadogiannakis, and Polychronakis], where we implemented a Grid monitoring infrastructure, in its turn inspired by many previous works (see the bibliography in the paper).

The reading of the CompatibleOne prototype [Marshall and Laisné(2012)] is an Open Source project aimed at developing an OCCI-compliant Cloud Infrastructure: its interface, that already covers monitoring and

SLA aspects, helped with a concrete view of a possibile interface. It has been enlightening concerning (among the rest) the need and possibility of modularizing the monitoring part.

The 2012 revision of the OCCI core model [OGF(2011a)] has been used as a reference.

6 Security Considerations

The API described in this document relies on the same mechanism as the basic OCCI API, of which it is an extension. In its turn, the OCCI API is designed according with a RESTFul model, a style of exposing a web service to the users.

The way this API is exposed inherits the security aspects of the RESTFul model, that can be summarized as follows:

- the web site MUST be protected to allow access only to authorized users, and to protect the content of the communication;
- the content uploaded on the web site by the user (using POST) MUST be protected;
- the content cached on third party sites not directly accessible by the user and by the provider (proxies etc.) MUST be protected.

We stress that these security warnings are shared with any RESTful API.

The provider must ensure that a user defined *mixin* does not compromise the security of other services. The provider may attain this by restricting the functionalities associated to a *mixin* (the limit case is the provision of templates) or run the functionalities associated to a *mixin* in a protected environment (e.g., as a Unix user in a chroot jail). This issue is shared with the OCCI model.

Concerning the kind of monitoring infrastructure deployed using the *Sensor Resource* and the *Collector Link*, security aspects are managed using appropriate *mixin*s. For instance the *Collector Link* might be associated with a *mixin* describing a secure transport protocol, while the *Sensor Resource* might be configured to be accessible only from authenticated users. The provider SHOULD offer the user a set of predefined *mixin*s that introduce the appropriate level of security. User defined *mixin*s SHOULD be avoided for this kind of options.

7 Glossary

metric a metric is a mathematical representation of a well defined aspect of a physical entity

measurement a measurement is the process of extracting a metric from a physical entity, and by extension also the result of such process. The measurement seldom corresponds exactly to the value of the metric.

SLA *"An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement."* from *SLA@SOI Glossary*

Restful model *"REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations."* [Fielding and Taylor(2002)]

OCCI "*The Open Cloud Computing Interface (OCCI) is a RESTful Protocol and API for all kinds of management tasks. OCCI was originally initiated to create a remote management API for IaaS model-based* *services, allowing for the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring"* [OGF(2011a)]

OCCI *Kind* *"The Kind type represents the type identification mechanism for all Entity types present in the model"* [OGF(2011a)]

OCCI Link *"An instance of the Link type defines a base association between two Resource instances."* [OGF(2011a)]

OCCI *mixin* *"The Mixin type represent an extension mechanism, which allows new resource capabilities to be added to resource instances both at creation-time and/or run-time."* [OGF(2011a)]

OCCI *Resource* *"A Resource is suitable to represent real world resources, e.g. virtual machines, networks, services, etc. through specialisation."* [OGF(2011a)]

Sensor Resource The *Sensor Resource* is a *Resource* that collects metrics from its input side, and delivers aggregated metrics from its output

Collector Link The *Collector Link* is a link that conveys metrics: it defines both the transport protocol and the conveyed metrics.

8 Contributors

Augusto Ciuffoletti (corresponding author)
Dept. of Computer Science
L.go B. Pontecorvo - Pisa
Italy
Email: augusto.ciuffoletti@gmail.com

Andrew Edmonds
Institute of Information Technology
Zürich University of Applied Sciences
Zürich
Switzerland
Email: andrew.edmonds@zhaw.ch

Metsch, Thijs
Intel Ireland Limited
Collinstown Industrial Park
Leixlip, County Kildare, Ireland Email: thijsx.metsch@intel.com

Ralf Nyren
Email: ralf@nyren.net

Appendix - Examples: from simple to complex

The OCCI Monitoring API is able to meet the demands of a wide range of users. It is understood that an application that has a limited interest in monitoring (for instance to trigger human intervention to cope with a fault) wants an interface able to configure in a straightforward way a simple strategy. In contrast, the user that is faced with a complex infrastructure and tight quality requirements needs an expressive interface. On the provider side as well there is interest for the possibility of restricting the monitoring tools available to certain users to a restricted set, with limited capabilities. The challenge for the overall scheme is to cover the whole range, from simple to complex.

In this appendix we explore use cases starting from a very simple one, approached in a simplistic way. The involved *Entities* are described by listing their attributes; both model attributes, in bold, and OCCI attributes. We recall that model attributes are not discoverable by the client, while OCCI attributes are.

Case 1: too simple

We start from an extremely simple case, that the user wants to implement trading off efficiency for simplicity. We focus on the `iostat` *mixin* used above, that a user wants to use to be warned about the overload of a server `vm1`.

The simplistic, yet suboptimal, solution is to associate the server with the monitoring tool. At a certain point in time, when the cpuutilization is 78%, the state of a certain server might be the following:

Attribute	value
id	urn:acme:user529/vm1
kind	compute
mixin	iostat-here
occi.compute.architecture	x86
occi.compute.cores	4
occi.compute.hostname	vm1
occi.compute.speed	3
occi.compute.memory	250
com.acme.iostat.cpuutilization	78
com.acme.iostat.what	user

The provider declares to update the *metric* attribute `cpuutilization` every 10 minutes, and the user is happy with that latency. From time to time the user will download the REST resource associated with the *Compute Resource* and parse out the value of the attribute, that will be processed in user's premises.

Let's consider a possible implementation on provider's side. Whenever the user associates the `iostat` mixin to the *Compute Resource*, the Cloud Management Infrastructure will install and launch a script that performs the call to the `iostat` command, and then sends the data to the Cloud Management Interface. In its turn, the Cloud Management server will update the record describing the Compute Resource. At this time any cached content of the same record should become invalid.

It is a quite complex operation that hardly fits in a REST environment.

This solution, which is admissible for our API, is extremely simple for the user, but extremely inefficient and complex for the provider. Let us explore an slightly more complex alternative that exhibits a reasonable footprint.

Case 2: simple but effective

The user, in addition to the *metric mixin*, associates to `vm1` also a *publisher mixin*: for instance a `tcp` mixin. Its semantic is that the data is returned after a `connect` to a given TCP address, that we assume to be located on the same virtual machine.

Attribute	value
id	urn:acme:user529/vm1
kind	compute
mixin	iostat, tcp
occi.compute.architecture	x86
occi.compute.cores	4
occi.compute.hostname	vm1
occi.compute.speed	3
occi.compute.memory	250
com.acme.iostat.cpuutilization	cpustat
com.acme.iostat.what	user
com.acme.tcp.in	cpustat
com.acme.tcp.source	http://www.acme.com/user529-vm1
com.acme.tcp.port	4321

The provider declares that the latency of the data is less than 10 minutes. The user will poll the socket from time to time, and obtain the CPU utilization.

Let's consider a possible implementation on the provider's side. Upon association of the two *mixin*, the Cloud Management server will trigger the `iostat` script, and implement a pipe to trasfer the results to a TCP server listening on the indicated port. The configuration of the pipe is driven by the correspondence between the string (`cpustat`) associated with the *metric attribute* of the `iostat` *mixin*, and the *input attribute* of the `tcp` *mixin*.

In a similar way, not shown in this example, the user may associate also an *aggregator mixin* to `vm1` to process the raw measurements and obtain a filtered metric.

Note that the activity of the Cloud Management server is limited to the configuration of the two *mixins*. After that, the record of `vm1` will be updated with the presence of the two *mixin*s. There is no further activity on the side of the provider, and caches will remain consistent after that operation.

Widening the horizon

Consider that the user has allocated a pool of servers `vm1...vmn` and that she needs only the maximum CPU utilization in the pool. Instead of downloading all measurements and find the maximum in her premises, she prefers to delegate the task to the Cloud Management.

Our solution is to create one *Collector Link* instance in egress from each server, and to associate a `iostat` *mixin* to each of them, thus adding the possibility to control the timing of the measurements. There will be one addressable REST resource for each of them.

All the *Collector Link* instances will share the same destination, that might be one of the servers. There an *aggregator mixin* aggregates the data by computing the maximum each time a new value is received, and delivers the data to the TCP server described in the previous example. The following is the state of one of the collectors:

Attribute	value
id	urn:acme:user529/c1
kind	collector
mixin	iostat
source	urn:acme:user529/vm1
target	urn:acme:user529/master
occi.collector.period	600
com.acme.iostat.cpuutilization	cpustat1
com.acme.iostat.what	user

and this is the state of the master server that receives all measurements, computes the output value and delivers the result throufh a TCP socket:

Attribute	value
id	urn:acme:user529/master
kind	compute
mixin	max, tcp
links	urn:acme:user529/c1, urn:acme:user529/c2, urn:acme:user529/c3
occi.compute.architecture	x86
occi.compute.cores	4
occi.compute.hostname	master
occi.compute.speed	3
occi.compute.memory	250
com.acme.tcp.in	cpumax
com.acme.tcp.source	http://www.acme.com/user529/master
com.acme.tcp.port	4321
com.acme.max.data	cpustat1, cpustat2, cpustat3
com.acme.max.max	cpumax

The use of *Collector Link* has a number of advantages, that are all related with the fact that the activities associated with the *mixin* obtain a distinguished address in the system, and thus are REST resources on their own. For instance, multiple instances of the same monitoring tool may run on the same *Resource*.

Separation of concern

In the above example the measurements are processed on a generic computing resource: however, there are cases when monitoring data deserve a specific treatment, that can be hardly implemented inside a generic virtual machine. For instance when it has an heavy footprint, or it requires accurate timing, or it has effects that cannot be triggered by a generic virtual machine, or it is considered as confidential. If this is the case, then it is time to create an instance of a *Sensor Resource*.

For our example, we consider a user that wants to put in place a mechanism that instantiates a new server as soon as the maximum `cpuutilization` on one of the servers reaches a given threshold. For reasons related with system integrity, the provider does not allow to associate this activity to a generic *Resource*, but it implements this privileged operation in a *publisher mixin* that can be associated only with a *Sensor Resource*. The *mixin* that manages the generation of the new request is called *elasticpool*: for simplicity, we assume that it exposes a single attribute, the threshold, in the interval [0..1], but we may imagine that a realistic *mixin* may indicate a template for the new resource, a *Collection* where to include the resource, and a deallocation rule.

The layout of the system is now made of a number of *Collector Link*, one for each server in the pool, and one sensor that aggregates all results and allocates new *Compute* when needed.

Each of the collectors will be defined as follows:

Attribute	value
id	urn:acme:user529/c1
kind	collector
mixin	iostat
source	urn:acme:user529/vm1
target	urn:acme:user529/s1
occi.collector.period	600
com.acme.iostat.cpuutilization	cpustat1
com.acme.iostat.what	user

The *Sensor Resource* state is the following :

Attribute	value
id	urn:acme:user529/s1
kind	sensor
mixin	max, tcp
links	urn:acme:user529/c1, urn:acme:user529/c2, urn:acme:user529/c3
occi.sensor.period	600
occi.sensor.timebase	1371025907
occi.sensor.timestart	10
occi.sensor.timestop	3610
com.acme.tcp.in	maxcpu
com.acme.tcp.source	http://www.acme.com/user529/master
com.acme.tcp.port	4321
com.acme.max.data	cpustat1, cpustat2, cpustat3
com.acme.max.max	maxcpu

A A use case: OCCI Monitoring and SLA

The definition of a standard interface for the operation of a Service Level Agreegment (SLA) is regarded as a relevant step on the way of Cloud interoperability: the accomplishement of this task would end up improving the user experience, since the presence of guarantees about the performance figures that are considered relevant for a given application increments user confidence. The purpose of this appendix is to explore a possible roadmap towards a SLA API that uses the OCCI-monitoring API: this is to verify that the OCCI-monitoring API is sufficiently expressive for the task.

As inthe case of OCCI-monitoring, we propose to distinguish between those that are semantic aspects that it is difficult (or inappropriate) to include in a recommendation, and the skeleton that supports their representation. In the former category we include the specific metrics used to evaluate the service level, the legal terms of the contract, and the logic used to decide whether, at a certain time, the system conforms to the SLA. We want to make as little assumptions as possible about these aspects, so to leave the provider the widest range of options.

Instead we focus of the general structure of the provision of a Service Level Agreement service, and on the basic concepts that have been characterised for it: the Service Level Objective, and the negotiation.

The term Service Level Objective [?] indicates a function defined on system metrics: for instance, a SLO may return a boolean to indicate that the average response time of a database is below a given threshold. The SLO is referenced in the SLA statement and its failure of fulfillment can be used as a trigger for actions.

The concept of SLO is tightly adherent to that of the *Sensor Resource*, as defined in this document: computation of a function on collected metrics. The presence of an explicit *Collector Link* resource for metric acquisition introduces the capability to cross inter-provision boundaries: when the measured resource and the SLO sensor are provided by distinct entities, the *metric* mixins available for the leased resource may include, for instance, encryption, anonymization, or authentication. A specific adapter can be implemented inside the *metric* to suit the expectation of the sensor for a given metric. All these aspects, that entail a complex, policy driven management, are totally hidden from user's perspective: the user instantiates a *metric* mixin for a given resource, and attaches it to the SLO.

The (re)negotiation of a SLA requires a totally different approach: primarily, the activities related to the SLA are not triggered on a time basis. Since a SLA is provided to the user as a kind of service, it looks appropriate to define a new resource kind for it.

A SLA is a contract between a user and a provider that come in two basic flavors:

- The *provider* offers a SLA: the providers offers the user the ability to monitor the conformance to SLA contract
- The *user* offers a SLA: the provider offers the User the tools to implement resource monitoring (and possibily a SLA) to meet a SLA on the operation of the user (that is possibly a service provider in its turn).

Disregarding the organizational differences between the two cases, we see that they can be treated with the same abstraction of a SLA resource: a set metric collectors, possibly inter-provider, are built according with SLA statement, and SLA actions are triggered by sensor values.

This enlightens some of the main characteristics of an SLA resource:

- the SLA resource configures and links to a number of sensors, each representing a SLO
- the SLA links a SLO value with an actions: for instance, the raise of a ticket, the application of a different billing plan, the modification of the SLO etc.

The negotiation loop (refrained in Figure) envisions the following states, that follow the definition of the IaaS infrastructure:

1. SLA instantiation: the user defines the SLOs and submits an SLA service request
2. the provider discovers the metrics needed to monitor the SLOs, and the sensors are generated
3. the provider computes the cost associated to the SLO
4. in a multi-provider environment, the provider may generate sub-SLAs
5. the SLA is submitted to the user for acceptance (SLA offer)
6. the actions associated to the triggers are defined: both the user and the provider contribute to this step
7. the SLA becomes effective:

From this picture we see that two concepts are not already covered by the OCCI-monitoring API: the *cost* and the *trigger/action* association.

As for the cost, it appears to be associated to the SLO, each contributing with a component of the total cost. The fact that the total cost may not be exactly additive is modeled in a global cost function that is specific for the SLA. Here the relevant fact is that the cost may be regarded as asoociated with the *Sensor Resource*, thus being implemented as a mixin for a *Sensor Resource* resource itself.

The *trigger/action* association is tightly connected with the existence of a SLA: we put forward the idea of a resource SLA, containing as attributes the administrative details of the contract (like contractors, global cost function, location etc.) with a number of OCCI links that relates the SLA with the SLOs: the mixins associated with each link describe an action associated with the output of the sensor.

B Intellectual Property Statement

C Disclaimer

D Full Copyright Notice

E References

[Ciuffoletti et al.(2008)Ciuffoletti, Marchetti, Papadogiannakis, and Polychronakis] Augusto Ciuffoletti, Yari Marchetti, Antonis Papadogiannakis, and Michalis Polychronakis. Prototype implementation of a demand driven network monitoring architecture. In *Proceedings of the CoreGRID Integration Workshop*, Hersonissos (Greece), April 2008. URL `http://www.di.unipi.it/~augusto/papers/cur_2008_a.pdf`. Available through www.slideshare.net.

[Fielding and Taylor(2002)] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002. ISSN 1533-5399. doi: 10.1145/514183.514185. URL `http://www.cs.helsinki.fi/group/java/s12-wepa/resurssit/principled-design-of-the-modern-web-architecture.pdf`.

[Keller and Ludwig(2003)] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1): 57–81, March 2003. doi: 10.1023/A:1022445108617.

[Marshall and Laisné(2012)] Jamie Marshall and Jean-Pierre Laisné. *CompatibleOne Resource Description System*, 2012. URL `http://www.compatibleone.org/bin/download/Download/Software/CordsReferenceManualV2.11.pdf`.

[OGF(2011a)] OGF. *Open Cloud Computing Interface - Core*. Open Grid Forum, June 2011a. URL `http://ogf.org/documents/GFD.183.pdf`. Available from www.ogf.org.

[OGF(2011b)] OGF. *Open Cloud Computing Interface - Infrastructure*. Open Grid Forum, June 2011b. URL `http://ogf.org/documents/GFD.184.pdf`. Available from www.ogf.org.