Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 68 (2015) 163 – 172

Procedia Computer Science

HOLACONF - Cloud Forward: From Distributed to Complete Computing,

Automated deployment of a microservice-based monitoring infrastructure

Augusto Ciuffoletti[a]

[a]*Dept. of Computer Science - Univ. of Pisa, P.le B. Pontecorvo, Pisa I-56122, Italy*

Abstract

We explore the specification and the automated deployment of a monitoring infrastructure in a container-based distributed system. This result shows that highly customizable monitoring infrastructures can be effectively provided *as a service*, and that a key step in this process is the definition of an expandable abstract model for them.

So we start defining a simple model of the monitoring infrastructure that provides an interface between the user and the cloud management system. The interface follows the guidelines of Open Cloud Computing Interface (OCCI), the cloud interface standard proposed by the Open Grid Forum. The definition is simple and generic and it is a first step towards the definition of a standard interface for Monitoring Services. It allows the definition of complex, hierarchical monitoring infrastructure by composing multiple instances of two basic components, one for measurement and another for data distribution,.

We illustrate how the monitoring functionalities that are defined through the interface are implemented as microservices embedded in containers. The internals of each microservice reflects the distinction between core functionalities which are bound to the standard, and custom plugin modules.

We describe the engine that automatically deploys a system of microservices that implements the monitoring infrastructure. Special attention is paid to preserve the distinction between core and custom functionalities, and the *on demand* nature of a cloud service.

A *proof of concept* demo is available through the Docker hub and consists of two multi-threaded Java applications that implement the two basic components.

Peer-review under responsibility of Institute of Communication and Computer Systems.

Keywords: on-demand monitoring; cloud computing; Open Cloud Computing Interface (OCCI); microservice; automated deployment; containers;

* Corresponding author. Tel.: +0-000-000-0000 ; fax: +0-000-000-0000.
E-mail address: author@institute.xxx

Peer-review under responsibility of Institute of Communication and Computer Systems.
doi:10.1016/j.procs.2015.09.232

1. Introduction

The steady growth of the computational power of computing devices is driving a similar trend in virtual resources: as a matter of fact, it is presently possible to implement virtual computers, networks and hard drives on top of servers with an acceptable loss in performance, but with a tremendous gain in flexibility. One of the notable effects of this evolution is the raise of the "cloud computing" hype.

Cloud computing leverages the presence of enterprises with powerful data centers that provide resources on demand: the provision mechanism relies on virtualization.

We distinguish two different approaches to virtualization:

- one uses the *hypervisor* technique, and envisions the realization of the whole operating system stack over a slice of the hardware resource [1];
- another uses the lower layers of the running operating system to implement one or more *containers*, that are isolated environments where to run an application [11].

Both types of virtualization are currently available from public cloud providers, and it is difficult for the user to determine which one is used. However they exhibit very different features, and they are indeed oriented to distinct evolutions: the choice between the two approaches is a trade-off between flexibility and efficiency. The hypervisor technique is less efficient, since it implements the whole OS stack, but, for the same reason, it is agnostic with respect to the host OS: i.e., you can have a virtual machine with a Linux operating system running on Windows. Instead the container is layered over the current OS kernel, and therefore it is far more efficient, but OS dependent.

The difference between the two approaches may be immaterial for the public cloud provider but the *container*-based approach is definitely more attractive for the designer that wants to implement (or develop) a distributed architecture: the limit of adopting a certain OS may be irrelevant, while performance issues are a key factor.

A *container*-based approach evolves towards the realization of complex but agile distributed architectures, composed of small and specialized services: the *microservice* approach [7] is a promising design paradigm that is tightly bound to (or merging with) the container technology.

In this perspective it is of great importance the availability of configuration tools for the *automated deployment* of containers hosting *microservices*: this to guarantee that an architecture can be cloned reliably – for development, testing, and operation. This need has a strong practical impact, and container technologies (like Docker, that we will consider in this paper) provide a basic tool (the Dockerfile in our case) that automates the deployment of a new container. The same need arises in the realm of hypervisor-driven virtualization:Vagrant, for instance, implements this feature with the Vagrantfile. The automated provision of an infrastructure composed of several *microservices* is a topic that has not been explicitly dealt with in the young field of *fine grain Service Oriented Architectures* (the term is attributed to Adrian Cockcroft), but there are tools under the *cloud computing* label that may fit the purpose: for instance, the aforementioned Vagrantfile may automate the deployment of a network of virtual machines across several cloud providers.

One fundamental step to obtain an *automated deployment* of a distributed, container-based infrastructure is the formal description of the cloud provision: it is a well studied topic for which standardization matters. And in fact many standards have been issued by major institutes (like NIST, DMTF, OGF and other) to allow cross compatibility of cloud provision specifications.

We explore a ground that is between cloud provisioning interfaces and *microservice* architectures, focusing on monitoring, and answer the question "How would you implement an *on demand* monitoring service?". We define a schema that describes an infrastructure composed of monitoring agents and probes: it conforms to the OCCI core schema, a standard for cloud interfaces promoted by the Open Grid Forum. After a formal description, we explain how it is used in practice. In essence,

we propose a *proof of concept* of a complete solution for the provision of an *on demand* monitoring service.

The reference architecture of the monitoring subsystem is composed by two entities: one that manages data, a sort of proxy, another that produces data. Such two components are simple, yet extremely variable from application to application: they are the ideal candidates for a micro-service based architecture. More precisely, the first one is represented as a stand-alone *microservice*, the other as a *probe* that may be embedded in the monitored resource: we do not make any assumption about the monitored resource, which may be a LAMP server as well as a networking device.

Next we show how the monitoring infrastructure is automatically deployed and configured. In particular we detail how the applications that implement the monitoring framework specialize their operation after downloading their description, rendered as a JSON document (the JSON rendering of OCCI entities is formally defined [10]) embedded in a web resource.

A prototype has been implemented and it is described in the last section: written in JAVA, it is designed to be expandable with monitoring tools that may be developed separately, and seamlessly integrated in the architecture as *plugin* modules.

The contribution of this paper is the description of all the steps in the design of an *on demand* monitoring service based on microservices: from the model that describes the infrastructure to the engine that deploys the probes. All steps are demonstrated in a *proof of concept* implementation. Results can be easely reproduced (and extended) using resources available on the Docker repository. The design of the plugins that implement metric probes and the management of the measurments falls outside the scope of the paper, even if the demo includes fully functional examples for the sake of completeness.

2. A reference architecture for a monitoring infrastructure

According with a widely applied and studied reference architecture [5,4,6] we introduce a component that is specialized in the management (not production) of the *measurements*, and in a probe, embedded in other components, that produces and delivers the *measurements*. In our terminology, we call *sensor* the former, and *collector* the latter.

The collector is a metering application: using appropriate probes it measures the relevant metrics for the architectural component for which it has a privileged access. For instance, in the domain of computing infrastructures, we may consider as typical the measurement of the workload for a processor, of the free space for a storage, of the number of forwarded packets for a networking device. But the range is unlimited as the number of resources types.

The sensor has a complex functionality. One of its functions is of receiving and processing the measurements coming from the collectors; in other words the sensor transforms raw metrics into custom ones. This activity is viewed as a function that takes as input one/more streams of measurements, and produces one/more streams on the output. As in the case of the *probes*, the range of functions is endless: anonymization, averaging, aggregation are related keywords.

Another function of the sensor is the publication of custom measurements, to make them available to other applications. For instance, the stream may be used to populate an SQL database, or sent to a dashboard. Again, this kind of functionalities cannot be enumerated, since they strongly depend on the application environment.

So we conclude that if we want be have a widely applicable reference architecture, we need to be generic about the measurement techniques, about the way in which data are processed, and how they are published. The user is in charge to design these functionalities, that should be seamlessly plugged in the respective component.

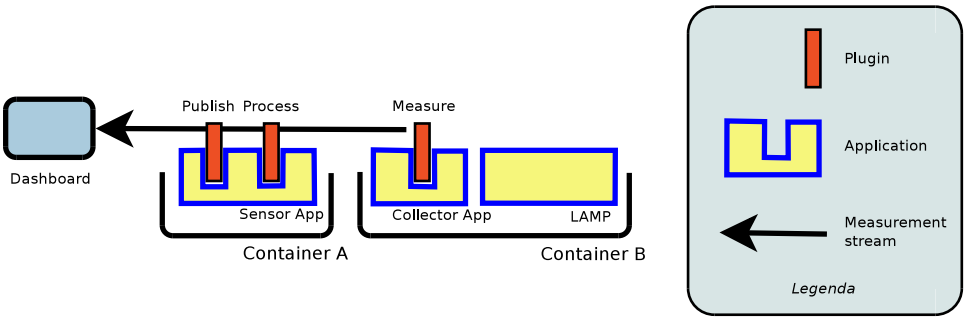

Fig. 1. A simple example using the reference architecture . The **Sensor App** implements a *Sensor Resource* hosted by a container. The **Collector App** implements a *Collector Link* hosted by the same container of the target resource. Red **plugins** implement *OCCI Mixins* of the indicated types associated with resource or link entities.

In summary, we have a reference architecture composed of one entity, the *sensor*, and of one relation, the *collector*. Their definition is generic, and their operation is specialized by the designer using appropriate plug-in modules.

There is another relevant concept in the reference architecture and it has to do with communication: it is the *stream* of measurements. It is characterized being:

- **discrete**, as composed by items representing a single measurement
- **time-related**, since each measurement is significant only if associated with a time reference.

The reference architecture is thus completely defined. An instance of a monitoring architecture is depicted in figure 1: it is a minimal system composed of a sensor, that averages a stream from a collector, and sends the output to a dashboard. The collector and the sensor are specialized with plugins that define their operation.

If we want this architecture to be deployed automatically, we need to formally define a language for its description: for this we define an extension of a standard interface model.

3. An extension of the Open Cloud Computing Interface

The Open Cloud Computing Interface (OCCI) is a standard for the description of cloud provisions, and gives the user the tools to define the resources that it wants to be instantiated in the cloud. It plays a fundamental role in a cloud computing architecture, since it defines how the user submits its requests and obtains feedback. The existence of a standard for this interface is of paramount importance for interoperability, and must at the same time be simple, to be easely understood by the user, and flexible, to allow extension and costumization.

The OCCI model is represented as a UML class diagram where cloud resources and their relationships are represented as two subtypes of the *Entity* class: the *Resource* and the *Link*, the latter representing a relationship between two *Resources*. The simplified diagram is included in figure 2

Using the *core* model[8] entities, a new document can describe further entities fitting specific provision types: the term used for this document is *extension*. For instance, an extension exists to represent an *Infrastructure as a Service* (IaaS) provision[9]. In that case three sub-types of the *Resource* kind are introduced: one for Compute entities, one for Storage entities, another for Networks. Two *link* sub-types are introduced to describe network interfaces, between a *compute* resource and a *network* resource, and for *storage*/*compute* relationships.

One fundamental role is played by *mixins*, another category of the OCCI UML model: they represent an additional characterization of an already instantiated entity. For instance, in the OCCI-IaaS

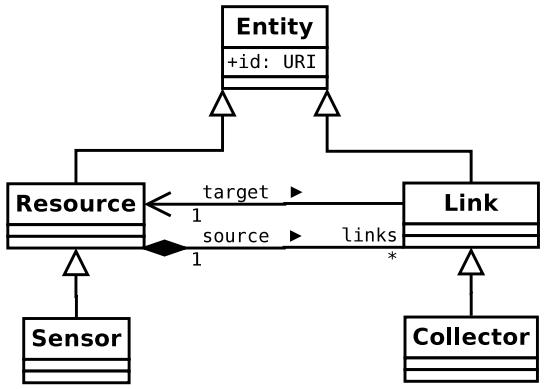

Fig. 2. The simplified UML diagram including the core OCCI classes (*Resource* and *Link*) and those in the monitoring extension (*Sensor* and *Collector*) .

Attribute name	Description
timebase	timebase – a date
timestart	start time offset – a number of seconds
timestop	stop time offset – a number of seconds
period*	sampling period – a number of seconds
granularity	time granularity – a number of seconds
accuracy	time accuracy – a number of seconds

Table 1. Sensor attributes (* mandatory attributes)

Attribute name	Description
period*	sampling period – a number of seconds
granularity	time granularity – a number of seconds
accuracy	time accuracy – a number of seconds

Table 2. Collector attributes (* mandatory attributes)

extension, an *Operating System* mixin is used to describe the operating system installed on a given *compute* entity.

Our reference architecture for monitoring infrastructures is described as an extension of the OCCI *core* model. The *sensor* is a sub-type of the *resource* kind: this reflects the intuition that its activity is independent and distinguished from other activities. It receives raw measurement streams, and produces in its turn streams of data.

A *sensor* inherits the attributes of the parent type, the *resource*, and introduces new specific attributes that indicate the expected sampling rate, and the accuracy of the timing. This latter attribute makes explicit the *real-time* nature of the measurement process: for instance, when a sensor attaches a time-stamp to a measurement, we need to know how accurate is the time-stamp, as well as its granularity. The resulting list of *sensor* attributes is in table 1.

The *collector* is represented as a sub-type of the OCCI *link* kind: intuitively, it means that a given *sensor* monitors a *resource*. The sub-type of the monitored resource is unrestricted, and thus a *collector* may interconnect two sensors: this allows the existence of a *proxy* sensor, one that receives measurement streams and forwards them to other sensors.

Like in the case of the *sensor*, besides inherited attributes we have other attributes that specify accuracy and expected frequency. The list is in table 2.

Summarizing, with the instantiation of *collector links* and *sensor resources* the designer defines the topology of the monitoring infrastructure. Their functionality is described by *mixins*, that add new attributes to those of a *sensor* or *collector* instance. We introduce three types of mixins, as summarized in table 3: they are generic mixins, that are further specialized by custom mixins designed and documented by the provider or by the user.

Mixin name	Description
metric	mixin for collector links – a probe that performs a measurement
aggregator	mixin for sensor resources – a processor that aggregates measurements
publisher	mixin for sensor resources – an interface that publishes measurements

Table 3. Mixin types for monitoring

A special role among mixin attributes is played by those that define the endpoints of the streams; they are described as `<name:value>` pairs, where the *name* is used in the mixin specification to describe the operation of the mixin, and the *value* is the identifier of a *channel*: two endpoint attributes that share the same *value* are considered as connected through a channel. The specifications of a given mixin describe how communication is implemented, and the protocol used on the endpoint.

An exhaustive description of the above schema is in the OGF document dedicated to monitoring [3]. In figure 3 we show a simple monitoring architecture: the sensor contains three plugins, one for aggregation and two for publishing, while the collector contains two metric plugins, that implement the monitoring of the internet connectivity and of the workload of a generic virtual machine.

Using the standard OCCI-JSON syntax [10] in table 4 and 5 we give the JSON description of respectively the sensor and the collector. Note how the value of channel attributes is used to specify the data flow.

4. From the reference architecture to containers

We want to deploy a monitoring infrastructure in a system composed of a number of containers following a description provided by the user. In this section we illustrate an abstract solution and its prototype implementation.

We consider that containers are OCCI *resources*, and that static configuration options, as well as activities that are dynamically activated inside a container, are represented as OCCI *mixins*. Containers are service driven entities, closer to a PaaS than to an IaaS provision model: we consider that the provision of a service is represented as an OCCI *link* directed from the user of the service (the *client*) to the provider of the service (the *server*). This is a generalization of the model introduced in [12], which reflects commercial PaaS provisioning.

Coming to the entities that are specific for the monitoring, a *sensor* is implemented as a container hosting the sensor application: it is a *bus* application that implements *channels*. Mixins for aggregation and publication are plugged into the sensor application and interconnected using channels.

The *collector* is implemented as a servlet hosted by a container. It measures the metrics of the resource implemented by the container and forwards them to the sensor. For instance, a database (*resource*) may host a mixin that counts (*measures*) the number of queries per time interval (*metric*). Like the sensor application, the collector application is a *bus* hosting measurement plugins.

Let us see step by step how a sensor and a collector are deployed.

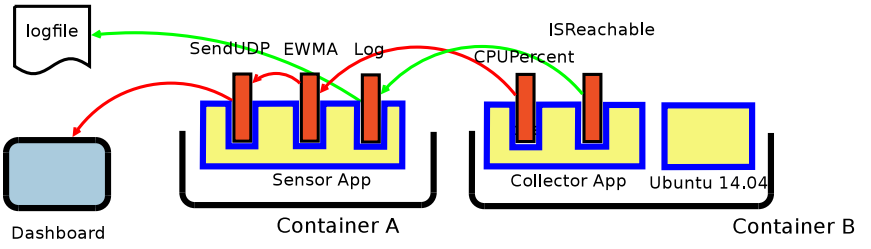

Fig. 3. A one-resource system with its monitoring: *Container B* is the monitored resource, *Container A* is the sensor

```
{
    "id": "urn:uuid:s1111",
    "kind": "http://schemas.ogf.org/occi/monitoring#sensor",
    "mixins": [
      "http://example.com/occi/monitoring/publisher#SendUDP",
      "http://example.com/occi/monitoring/aggregator#EWMA"
      "http://example.com/occi/monitoring/publisher#Log",
    ],
    "attributes": {
      "occi": {
        "sensor": {
           "period": 60, "timebase": 1386925386,
           "timestart": 600, "timestop": 3600,
           "networkInterface": "eth1"}
        },
      "com": {
        "example": {
          "occi": {
            "monitoring": {
              "SendUDP" : {"udpAddr":"192.168.5.1:8888","input":"c"},
              "EWMA" : {"gain":16,"instream":"a","outstream":"c"},
              "Log" : {"filename":"my/log/file","in_msg":"b"}}}}
      }
    },
    "links": ["urn:uuid:2345"]
  }
```

Table 4. JSON description of the sensor in figure 3. The `input` attribute of the `SendUDP` *publisher* mixin, and the `outstream` attribute of the `SendUDP` *aggregator* mixin are interconnected by the `c` channel.

The *sensor* obtains its OCCI description from the user interface. The plugins found in the description are launched as separate threads, and the channels between them are implemented.

The endpoints that receive measurements from the collectors are multiplexed across an *input port*. There is a exactly one *input port* for each collector.

The description of the sensor contains also references to originating *OCCI-links* (see the `links` attribute at the end of the JSON document in table 4), that represent the collectors. Once the sensor is ready to operate, it downloads the description of the collectors (see the example in table 5) to discover the monitored resources, and passes them the description of their monitoring activity through a dedicated *configuration port*. The sensor application discovers the *configuration port* by inspection of the OCCI document that describes the monitored resource.

The *collector* runs as a separate thread in the monitored container: its presence is configured with a specific mixin — the *metricContainer* — that controls the creation of an input port used by the sensors to pass the descriptions of monitoring activities.

Upon receiving a request across the *configuration port*, the collector launches the requested metric mixins as distinct threads. The measurement streams originating from metric threads are multiplexed across the *input port* of the sensor: each measurement is tagged with a *channel id*, and this allows the demultiplexing of channels on sensor side.

The resulting architecture for the sample monitoring infrastructure is in figure 4. It implements the monitoring of a plain host, producing two different streams: one that delivers a filtered average (the filter is an Exponentially Weighted Moving Average, abbreviated with EWMA) of the workload to a dashboard (in red), another that logs the reachability of a reference host (in green).

```
{
  "id": "urn:uuid:2345",
  "kind": "http://schemas.ogf.org/occi/monitoring#collector",
  "mixins": [
    "http://example.com/occi/monitoring/metric#CPUPercent"
    "http://example.com/occi/monitoring/metric#IsReachable"
  ],
  "attributes": {
    "occi": {
      "collector": {"period": 3 }
    },
    "com": {
      "example": {
        "occi": {
          "monitoring": {
            "CPUPercent" : {"out": "a"},
            "IsReachable" : {
                "hostname":"192.168.5.2",
                "maxdelay":1000,"out":"b"
            }
          }}}}},
  "actions": [],
  "target":"urn:uuid:s1111",
  "source":"urn:uuid:c2222"
}
```

Table 5. JSON description of the collector in figure 3. The `a` channel interconnects the `out` attribute of the `CPUPercent` mixin is with the `instream` attribute of the `EWMA` mixin in the sensor.

4.1. The prototype

We have implemented a prototype that demonstrates the applicability of the above abstract control flow. We have used plain Java as the application programming language (a Ruby version is on the way), and Docker for the containers. OCCI descriptions follow the OGF standard[8] and the transport of OCCI documents is delegated to HTTP. The code is circa 1000 lines of Java, detailed comments included, and privileges readability to efficiency.

The prototype implements:

- the Docker container that hosts the sensor;
- the servlet associated with the *metricContainer* mixin, to be run in a generic resource that is the target of a collector link;
- a small number of plugins, those needed to implement the sample infrastructure in Figure 3

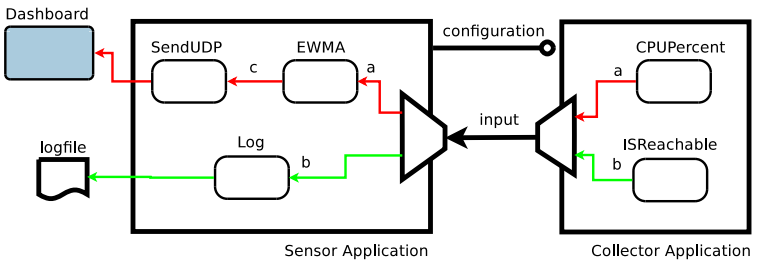

Fig. 4. JSON description of the monitoring system in figure 3

The *channels* that connect aggregator and publish mixins within a sensor are implemented as Unix pipes, while measurements from metric mixins are multiplexed across a TCP connection — that implements the *input port* — from the monitored resource (client side) to the sensor (server side).

The *configuration port* is implemented as an RMI interface on the monitored resource.

A running version of the prototype is available on the Docker hub as occimon-live[1]: the interested reader finds instructions for an *on premises* deployment of the sample infrastructure used in the example. The implementation of the proof of concept is not a complex task and can be carried out without applying to exhotic libraries in a couple of weeks. With limited effort it is possible to add new plug-ins and a more complex infrastructure.

5. Discussion: open problems and non-problems

We have formally defined an interface for the specification of a monitoring infrastructure as a system of *microservices*, and we have prototyped the entities that collaborate for its automatic deployment. The schema is based on the Open Cloud Computing Interface, following the guidelines of the OCCI monitoring document.

We consider the adherence to a *standard* as a primary feature, since it warrants that our conclusions can be effectively reused in other contexts, and may facilitate the convergence to an open and portable architecture. We immersed our schema in an ecosystem composed of *microservices*, which is a rapidly evolving paradigm that may have a strong impact in the design of future distributed systems.

The choice of a classical language and of outdated communication patterns (pipes, TCP Sockets, RMI) contrasts with the definition of an advanced environment. It is however justified since we want to suggest an architecture, and not to develop a product: it is therefore pointless for us to optimize implementation, our first goal being the delivery of *readable* code. For the same reason we implemented only the mixins that are strictly needed for the demo: the user is free to add more. Those included in the demo are examples that are included only for the sake of completeness.

Regarding the sensor/collector schema, one challenging use case is that of a system consisting of an array of hundredths of micro-servers: the SWARM[2] clustering system is an example. Here the apparent problem is in the definition of the OCCI monitoring infrastructure: one distinct collector for each micro-server would not solve the problem. However, consider that the same issue would emerge for the description of the system itself: one distinct OCCI resource for each micro-service would be prohibitive. We conclude that to address this use case there should be a syntax to express a *collection* of similar resources: using that same syntax we might describe a collection of similar collectors, and convey the monitoring streams across the same sensor.

There are two features that limit the power of the sensor/collector schema: one is that a single collector cannot feed more than one sensor. In other words the output of a collector cannot be multicast to more than one sensor: this is intrinsic in the fact that a collector is an OCCI link. However a resource may be the target of several collectors, possibly connected with different sensors: this preserves the possibility to have multiple monitoring activities (as sensors) attached to the same resource.

Another feature that limits the applicability of our schema is the tight relationship with timing. The schema is designed to cope with *periodic* measurements, and it is not prepared to manage (but may produce) asynchronous alarms. This is our response to the intent of keeping the schema as simple as possible: asynchronous alarms need a completely different management, and a dedicated scheme looks more appropriate.

[1] https://registry.hub.docker.com/u/mastrogeppetto/occimon-live/
[2] https://github.com/docker/swarm

The time driven approach highlights a delicate issue concerning virtualized environments: clock accuracy and, in general, real time. In our case, it means how accurate are the time-stamps attached to measurements, and how precise is the period between measurements. This may or may not be an issue, and in fact the accuracy attributes are optional in our model: however, when it is the case, clock accuracy is supported.

In this respect, operating system virtualization, represented in our work with Docker, has a significant advantage against the hypervisor approach. In fact, the container shares the same clock with the hosting machine, and inherits the same characteristics: if the hosting machine is not a virtual machine in its turn, clock synchronization is obtained with the appropriate accuracy using well known protocols (like NTP or PTP/IEEE1588). This is not true in the case of hypervisors, and the problem is actively investigated[2].

Our further steps with this work will probably go in the direction of implementing a set of plugins tailored on a specific use case, suggested by one of the institutes and agencies that are active in the definition of standards for on-demand cloud monitoring.

References

1. Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *In SOSP (2003*, pages 164–177.
2. Timothy Broomhead, Laurence Cremean, Julien Ridoux, and Darryl Veitch. Virtualize everything but time. In *in OSDI10*, pages 1–6, 2010.
3. Augusto Ciuffoletti. *Open Cloud Computing Interface - Monitoring*. Open Grid Forum, June 2015.
4. Augusto Ciuffoletti and Yari Marchetti. *A Distributed Infrastructure for Monitoring Network Resources*, volume 7 of *Horizons in Computer Research*, chapter 10, page 17. Nova Publishers, Hauppage, USA, June 2012.
5. 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.
6. Gianluca Iannaccone, Christophe Diot, Derek McAuley, Andrew Moore, Ian Pratt, and Luigi Rizzo. The CoMo white paper. Technical Report IRC-TR-04-17, Intel Research, 2004.
7. Sam Newman. *Building Microservices: designing fine-grained systems*. O'Reilly, 2015.
8. 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.
9. OGF. *Open Cloud Computing Interface - Infrastructure*. Open Grid Forum, June 2011. Available from www.ogf.org.
10. OGF. *Open Cloud Computing Interface - JSON Rendering*. Open Grid Forum, June 2015. Available from www.ogf.org.
11. Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.
12. Sami Yangui and Samir Tata. An OCCI compliant model for PaaS resources description and provisioning. *The Computer Journal*, 2014.