Heterogeneous Cloud Systems Monitoring Using Semantic and Linked Data Technologies

Alessandro Portosa[†⋆], M. Mustafa Rafique[‡], Spyros Kotoulas[‡], Luca Foschini[†], Antonio Corradi[†]
[†]Dipartimento di Informatica – Scienza e Ingegneria (DISI), University of Bologna, Italy;
[‡]IBM Research, Ireland
Email: alessandro.portosa@studio.unibo.it; mustafa.rafique@ie.ibm.com; spyros.kotoulas;
luca.foschini@unibo.it; antonio.corradi@unibo.it

***Abstract*—Cloud businesses need comprehensive visibility on hardware and software components, their utilization and their configuration. In addition, they need to integrate such information with their asset management systems and publicly available information such as hardware specifications. In this paper, we present an approach for cloud management and monitoring based on a semantic layer that unifies different interfaces and representations, and makes all relevant information accessible from a single point. We show a proof-of-concept based on OpenStack and Linked Data technologies and evaluate it in terms of overhead, query execution times, and effectiveness in the data gathering phase. Our findings indicate that a semantics-based approach is indeed feasible and advantageous for providing uniform access across different cloud environments and levels.**

Keywords*—*Cloud computing; monitoring; * as a Service (*aaS); linked data

I. Introduction

Cloud computing systems are generally composed of diverse infrastructures that run on diverse software and hardware components. Depending upon the customers' requirements, these components are used to provide different service offers, such as Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS), Database as a service (DBaaS), etc. Software components in cloud systems often have diverse ownership, some being owned by cloud providers and others by customers. Even at the application level, it is possible that not all components have the same ownership, e.g., consider a typical scenario where a bare metal host runs an application server, which is owned and managed by the customer, communicates with a database (DB) server, which is instead owned and managed by the service provider (DBaaS).

With the increase in the use of cloud resources, one of the fundamental challenges faced by cloud providers is the lack of fine-grain monitoring tools to gather useful resource utilization information about the underlying system. The current state-of-the-art cloud monitoring frameworks provide resource monitoring either at the granularity of physical or Virtual Machine (VM) level. Often, they are unable to providing needed aggregated information, making it challenging and sometimes impossible for a provider to determine the resource utilization of either entire distributed services or particular processes and applications inside a VM.

Linked Data technologies have emerged as a way to integrate large-scale information in a flexible manner. Linked Data represents each entity as a unique, global identifier, which is a generalization of Uniform Resource Locators (URLs). Relationships between entities are also represented with such identifiers. A set of World Wide Web Consortium (W3C) recommendations standardizes semantics and representations: Linked Data enables the representation of an arbitrarily extensible information space with well-defined semantics.

In this paper, based on the versatility of Linked Data technologies, we propose an approach for heterogeneous cloud systems monitoring. The novelty of our approach lies in: (i) its ability to harvest information from multiple systems and at different level; and (ii) extensibility of the model to cater for the diverse modeling needs. We design and develop a framework called Semantic Monitoring Agents for Cloud Systems (SMACS) that enables efficient monitoring of hardware and software components in a cloud setup. We integrate SMACS with the widely-used OpenStack cloud computing platform [1] to elaborate its use and compatibility with conventional cloud computing platforms.

The rest of the paper is organized as follows. In Section II, we discuss the work that is closely related to our work presented in this paper. In Section III, we highlight the background technologies that enables SMACS. In Section IV, we present the design and architecture of SMACS. In Section V, we present an evaluation of SMACS and demonstrate its effectiveness. Finally, we conclude the paper in Section VI.

II. Related Work

Monitoring is a core functionality of any integrated network and service management platform. Cloud computing increases the complexity of monitoring infrastructures since it incorporates multiple physical and virtualized resources. Furthermore, it spans over several layers, from Infrastructure as a Service (IaaS) to Platform as a Service (PaaS) and Software as a Service (SaaS). In this section, we summarize the existing monitoring approaches that are closely related to the work presented in this paper.

Nagios [2] is a very popular server monitoring service to monitor the status of services and resources in data centers. Nagios adopts a centralized client/server architecture where a central server gathers monitoring information from remote nodes. Notwithstanding its widespread diffusion and different supported interaction modes, Nagios is more oriented to service status visualization rather than monitoring continuous changes of system resources, such as memory and CPU. Moreover, it is designed to monitor traditional large-scale IT infrastructures instead of cloud systems.

⋆A part of this research was conducted during a student placement at IBM Research, Ireland.

[3] proposes a hybrid pull-and-push approach to monitor cloud resources allows switching the communication strategy according to users' consistency and efficiency requirements. However, it focuses only on the interaction model (pull/push) and does not tackle the problem of scalable and timely data distribution. Another work [4] proposes distributed monitoring for load balancing in virtual networks based on network metrics gathered by agents deployed at each virtual interface. However, this work is limited to the network resources and does not incorporate other resources, such as CPU and memory.

The push-based approach in [5] monitors services in clouds with the goal of scaling web applications: a monitoring agent, installed in the web application, triggers opportune up-scale or down-scale operations. The main problem of this proposal is that it focuses mainly on web server resource usage at service level without incorporating hardware resources. Furthermore, it does not support any monitoring at the infrastructure level.

DARGOS [6] adopts a fully-distributed peer-to-peer publish/subscribe model, and a hybrid push/pull approach to disseminate resource monitoring information. It provides physical and virtual resources utilizations in the cloud while maintaining a very low overhead. Moreover, it is designed to be flexible and extendable to new metrics.

PCMONS [7] is an open-source pull-based cloud monitoring system. It relies on a monitoring server that receives information from multiple data integrators that pull cloud monitoring information. It relies on a single server to process the obtained monitoring information thus impairing its scalability.

An agent-based resource monitoring system is proposed in [8] for multi-tenant cloud environments where a cloud system is administered by cooperating entities via a one-to-many communications publish/subscribe communication support. Similarly, Lattice [9], adopts a publish/subscribe paradigm to disseminate monitoring information originating from cloud nodes. Lattice is also highly scalable and can perform complex monitoring tasks on large scale virtual networks.

Notwithstanding the recent advancements in cloud monitoring domain, we believe that extending these advancements with the use of standard semantic technologies, to ease interoperability and to foster higher expressiveness, may represent a big step forward in the cloud monitoring research area.

III. Linked Data for Cloud Systems Monitoring

Linked data has emerged as a paradigm for information integration across domains and systems [10]. It typically adopts a Resource Description Framework (RDF) [11], [12] representation. The basic RDF model dictates that information is represented as a set of labeled edges across nodes with unique, global identifiers. Typically, a standardized triple notation is used, consisting of RDF terms, generally referred as Subject-Predicate-Object triples. Data is usually stored in a data store (typically referred to as triple store or RDF store) and queried through the SPARQL query language.

The RDF terms can be URIs, Literals or BNodes. We leverage URIs in this work. URIs uniquely identify entities, properties or concepts: for example IBM has a unique URI, same as the concept Company. The same holds for properties. URIs may come from different domains, typically depending on the data publisher. Nevertheless, this does not mean that data owners may not use the URIs of others. In fact, they are encouraged to. For example, IBM may use URIs from DBPedia, so as to promote integration. Literal values are represented through typed literals (when no type is mentioned, type `String` is assumed).

We have chosen Linked Data as the data representation paradigm of our system for the following reasons. First, Linked Data (and associated schemas/ontologies) are arbitrarily extensible. This allows extensions to provide fine-grain information for various components: while all programs may report memory used, server components may additionally report network bandwidth. Moreover, web servers may also report request throughput. Second, the global semantics of URIs allow us to uniformly access information without relying on a specific (database) representation. Third, merging data from various Linked Data sources is very easy. In principle it entails either doing federated queries across the stores, without any schema mediation, or storing all information in the same RDF store, effectively taking the union of the triples from different systems. Schema mapping and consolidating entities from different sources is more complicated and is beyond the scope of this paper.

IV. Framework Design

It is typical to have a system based on a heterogeneous cloud environment with different monitoring mechanisms at different levels. In literature, a cloud systems is divided into the three main levels, i.e., Infrastructure as a Service, Platform as a Service and Software as a Service. Due to distinct characteristics of each layer, developing a cross-monitoring framework for all resources in the cloud-stack is very challenging. Therefore, required tools for monitoring operations are selected and installed according to specific needs at each layer. Moreover, the information content is represented to cater a specific tool, regardless of the fact that the semantic might be the same. The main challenge of a unified monitoring framework is to overcome the diversity of the systems to integrate all features in a synergistic way, especially for fully autonomous systems.

To design a unified cloud monitoring system, it is critical to map information in order to support all levels of heterogeneity while maintaining the original semantics with finer granularity. Moreover, a unique way and single access point to request data from the cloud environments is required. We develop SMACS as a monitoring system for the integration and aggregation of metrics data coming from various systems. To this end, SMACS relies on a *client/agent/server* distribution model and uses a *pull* interaction model with a *periodic* update strategy.

SMACS provides monitoring capabilities across the stack of cloud services (*public* and *private* IaaS, PaaS and SaaS) overcoming the heterogeneity of technologies. The schema of considered environments and resources is shown in Figure 1. In the following, we present and describe our view of a unified cloud monitoring architecture that uses semantic technologies to overcome the challenges of using different monitoring tools and techniques.

A. SMACS Architecture

SMACS is composed of six main components, namely, *Data Provider*, *Cloud Broker*, *Data Agent*, *Storage Consistency Maintainer*, *TDB* and *Resources Information Endpoint*. The interaction between these components as well as the overall architecture of SMACS is shown in Figure 2. From the functionality point of view, the proposed framework can be

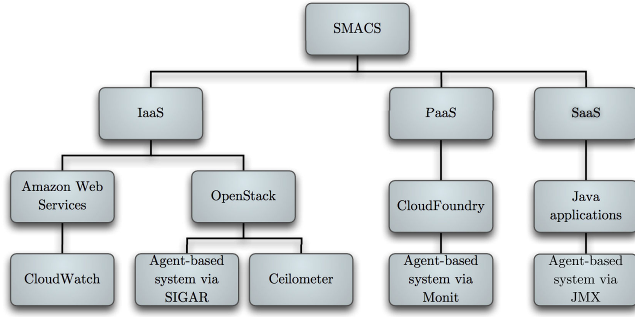

Fig. 1: Schema of considered environments and resources

split in the following two sections, where *Data Provider* acts as bridge between the two sections:

- A *gathering layer* (the left half) pulls the monitoring data from the cloud environment, using a deployed agent or a set of exposed cloud API.
- A *semantic layer* (the right half) provides the required level of abstraction to enhance the decoupling between the implementation details and the single access point.

Data Provider is the core component to compose the proposed architecture. It uses *Cloud Broker* component to communicate, when expected, with the cloud environments. In order to start gathering the data, *Data Provider* needs the IP address of the compute nodes and the VM instances, in addition to the VM status and availability zone, to decide if a monitoring action is required. An internal component, *Data Gatherer*, is responsible for starting the *Cloud Broker* and handling the received information. A *Data Worker*, when expecting the information from a Cloud Broker, starts communication with *Data Agent* by sending an asynchronous request via REST web services and returns back a set of information to the *Data Gatherer*, which stores the received information in the TDB, as better explained in the following.

Cloud Broker provides a bridge between the data provider and a cloud environment, such as OpenStack and Amazon Web Services. Due to the lack of a standard interface for the cloud management, extending SMACS to communicate with others IaaS requires the development of a new cloud broker for each adopted interface. For example, the Amazon EC2 API [13] is a de-facto standard and the OpenStack framework has adopted a compatible set of APIs, even though the APIs are still not fully compatible with each other. Similarly, the Open Grid Forum (OGF) and the Distributed Management Task Force (DMTF) have been proposed as standard interfaces for cloud computing management [14], [15]. *Cloud Broker* communicates with OpenStack and Amazon Web Services via REST APIs and forwards information regarding hypervisors, host, and instances (e.g., IP address, node name, VM status, etc.) to the *Data Provider*. It also leverages specific cloud monitoring APIs provided by Ceilometer service (OpenStack) and CloudWatch (Amazon Web Services).

Data Agents are responsible for polling metering data and sending them to the *Data Worker* using REST web service. Furthermore, data agents are developed to be independent of a specific deployment environment, making them portable across different operating systems. The *Data Agent* component is stateless and is instantiated when a GET request on a

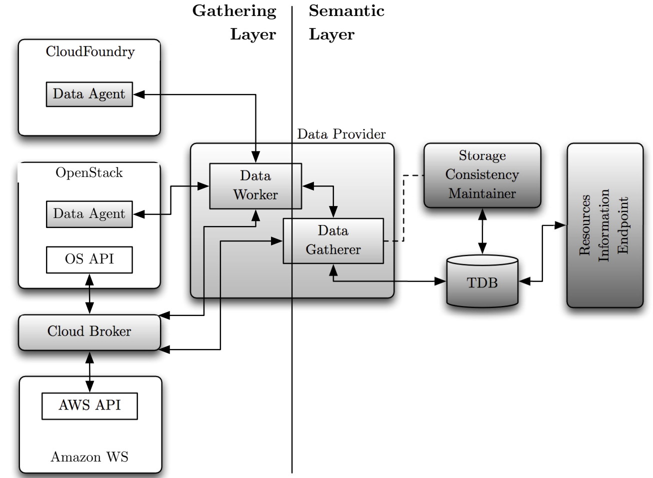

Fig. 2: Framework architecture

specific URI occurs. An agent is able to gather metering data (e.g., percent of CPU load, number of writes on a specific disk, etc.) for every monitored resource. It also produces a consolidated summary about all the monitored resources in a cloud environment.

Storage Consistency Maintainer maintains the consistency of TDB according to the specified configurations. It is started by the *Data Gatherer* during the initial system startup and it accepts parameters that influence the size and organization of the TDB data storage.

TDB (*Triples Database*) represents and builds the data abstraction. It provides a triples storage, built using Jena TDB, where RDF data are stored by *Data Gatherer* and periodically removed by *Storage Consistency Maintainer* according to a time-based filtering option.

Resources Information Endpoint acts as a front-end for the whole architecture. This component enables requesting information, interacts with the TDB and uses the gathered and stored data to satisfy such requests from the users.

B. Data mapping on RDF

The multiplicity of resources and the diversity of computing environments involved in our monitoring scenario require the use of different monitoring mechanisms. Therefore, SMACS relies on different services to enable heterogeneous monitoring capability. To this end, we model three types of agents. The first agent leverages cross-platform monitoring library to monitor the status and resources of cloud infrastructure. We integrate a set of native libraries provided by System Information Gatherer and Reporter (SIGAR) [16], which is a platform- and language-independent library for accessing operating system and hardware level information in Java. SMACS uses SIGAR to implement *Node Agents* that obtains on-demand real-time information about the infrastructure of the compute nodes and the running instances.

The second agent integrates a Java Management Extensions (JMX) technology [17]. JMX provides management and monitoring support for applications, system devices and objects using a client-server architecture. Most enterprise applications, such as Apache Tomcat, Apache web server, IBM WebSphere, etc. already provide interfaces for JMX technology.

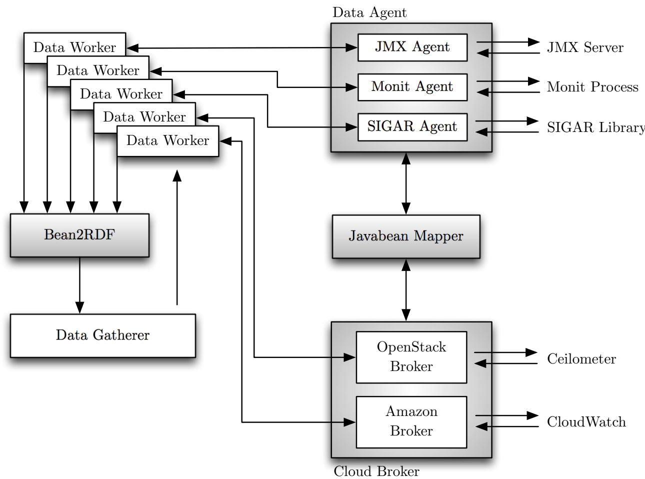

Fig. 3: RDF mapping process

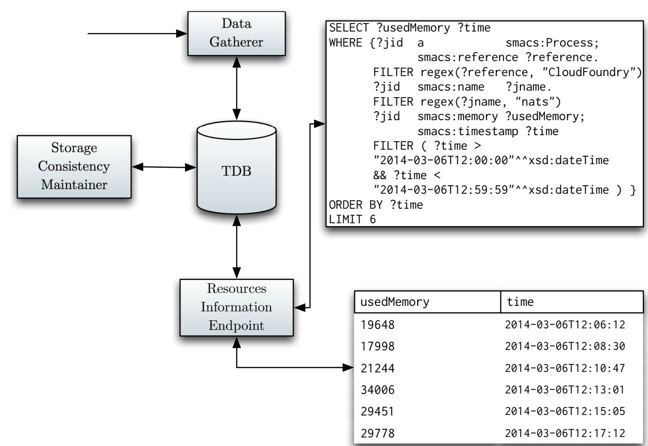

Fig. 4: Semantic layer integration

In SMACS, a JMX client connects with the JMX agent to perform management and monitoring operations on the target applications and resources.

Although there is no direct interface to monitor CloudFoundry, it provides state information on jobs through the Monit daemon of BOSH. Our third agent dialogues with CloudFoundry, executing locally (since it must be deployed inside the monitored CloudFoundry installation) the command `monit status` that provides status and metrics about all the CloudFoundry processes. Once results from the command have been obtained, the *Data Agent* maps the metrics using common objects which are translated later to RDF.

As shown in Figure 3, *Data Gatherer* starts the process of pulling the data from the *Data Agent* and maps it into RDF. This approach is fundamental to provide a unique interface where different information can be retrieved homogeneously. *Data Gatherer* creates a hierarchy of entities, where the leaf nodes represent *Data Workers* that start communication with specific *Data Agents* and *Cloud Brokers*, and process the received information.

Information received from a *Data Agent*, which uses a JSON encoding, are mapped in a set of JavaBeans. The aim is to built a corresponding set of RDF starting from these classes and store the new converted data in the TDB. To this end, we use an existing tool, JenaBean[1], that transforms common object oriented code into Semantic Web Languages. JenaBean is a powerful and flexible RDF API to persist JavaBean. Its approach is not driven by the Web Ontology Language (OWL) [18] ontology or RDF schema, but it is instead class-driven. This kind of binding enables saving JavaBean as RDF with no mapping file or template. Every bean class is coded with a set of annotations to extend the semantic information (RDF property and namespace), providing a more fine binding between object and RDF.

When a response is received, or if a connection timeout, the *Data Worker* returns back the metering data to its parent in the hierarchy. Once all *Data Worker* complete their tasks, a highly parallel merging phase starts. In this phase, the metering data is translated in to RDF, and each RDF is also merged with other RDFs. Every RDF shares the same schema, the base of the URIs and each resource keeps a reference to its parent resource. This enables creating a network of metrics, where the whole information gathered from different sources is linked together. In this way, the top of the hierarchy returns a unique RDF object that includes linked metrics for the specific monitoring round. Finally, the *Data Gatherer* gets the unique RDF representing the whole set of the retrieved information.

[1] http://code.google.com/p/jenabean/

C. Semantic layer Integration

The integration of a semantic layer in SMACS architecture allows a better decoupling between the user's view of the framework and the implementation details. Figure 4 shows the integration of semantic layers in SMACS and the main components involved. The *Data Gatherer* extracts the triples from the RDF existing from the *Data Worker* hierarchy and adds them to a specific TDB graph. If the relative graph does not exist, an empty one is created on demand and is inserted as usual.

The TDB is organized in graphs, where the name of each graph depends on its creation date. A graph for each time frame is created on demand and the triples are stored in the right graph depending on when a specific resource is monitored. This modeling approach enables efficient data retrieval and processing by isolating the monitored resources and the associated data.

Storage Consistency Maintainer leverages the concept of a time window, where the size of the window is chosen by the *Data Provider*. In effect, the *Data Provider* is also responsible to start the execution of the *Storage Consistency Maintainer*, and providing it with the information about the width of the time window. *Storage Consistency Maintainer* extracts a list of the named graphs and checks if there exists any graph that are outside the time window (the graphs are named using date time format).

The *Resources Information Endpoint* has been integrated as a SPARQL endpoint, to provide an efficient access to the stored data. The endpoint is closely tied to the TDB that acts as the information container, and the data can be queried using the SPARQL syntax, which allows access to the corresponding information by using clauses, filters and functions [19] with a high level of expressiveness. Finally, the results from the query execution are shown in a tabular format. We note that the

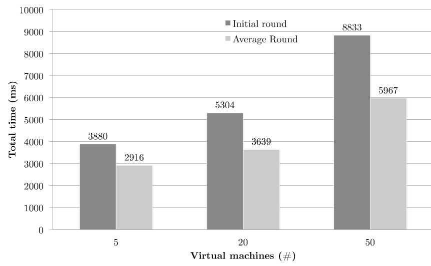

Fig. 5: Initial and average data gathering and RDF data conversion

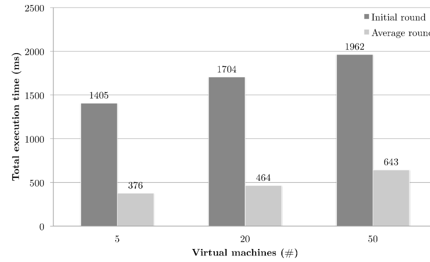

Fig. 6: Initial and average RDF data storing

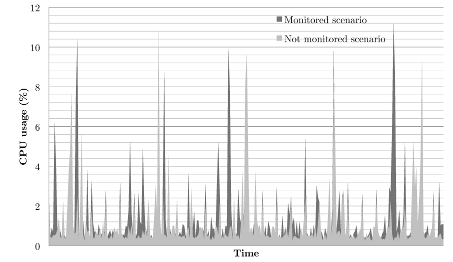

Fig. 7: CPU overhead

joint use of a SPARQL endpoint, TDB storage and RDF data representation is critical to archive the required role-separation between the two layer (*gathering* and *semantic*) in SMACS.

V. Evaluation

In order to prove the validity of our solution, we report a series of different experiments regarding the architectures described in the previous section. Benchmarks on scalability and performance measurements of the SMACS has been performed in the IBM Research Lab of Dublin (Ireland), where a real distributed scenario for OpenStack and SMACS was installed. Subsequently, further experiments on SMACS functionalities are conducted in the University of Bologna.

A. Performance Analysis

We consider a SMACS testbed consisting of 4 high-end physical nodes placed in the computing cluster of IBM Research in Dublin. Each node consist of 2 Intel Xeon X5670 @2.93 GHz (12 cores each) processors, 256 GB memory, 1 TB hard drive and running RHEL 6.3 as operating system. Each physical node is provided with two network interfaces, one with 1 Gbit/s throughput reserved as private interface (which mainly involves VMs communication) and one with 10 Gbit/s throughput used as public interface for inter-node communication. Three of the aforementioned nodes are used to manage an OpenStack Grizzly installation which consists of one master node running all OpenStack Nova services (e.g., compute, scheduler, network) and two slaves node running nova-compute service. The last node of the four is used to run the SMACS semantic layer, the TDB database, the SPARQL endpoint and the Data Provider.

The deployment of SMACS agents concerns both the physical nodes and VMs of the OpenStack environment. VMs are sized with a standard m1.tiny flavor (1 VCPUs, 512 MB memory). Our evaluation of SMACS is focused on analyzing the performance and scalability as well as proving the feasibility of the proposed solution. We tested our system in a low workload and traffic scenarios to highlight the overhead introduced by the monitoring layer. We further investigate the scalability of SMACS by increasing the number of running VMs from 5 to 50 while logging the execution time of RDF data storing, data gathering and RDF data conversion.

1) Survey on Data Gathering and RDF Conversion: We evaluated the performance of our agent-based polling system, including the RDF conversion process. Figure 5 details about polling and RDF data conversion time. The measure starts when Data Gatherer begins to request data, continues during all the communications between and stops when the “Map and Merge” phase ends. The chart shows an initial high value due to the instantiation and configuration of the Data Provider, Data Agents and Storage Consistency Maintainer. That can considered a warm-up “issue”, which will not affect the normal behavior of the system. After few rounds, the time decreases reasonably and the monitoring system replies faster. Using SMACS, increasing the number of VMs by a factor of 10 changes the total data gathering and RDF conversion time only by a factor of 2 showing that the monitoring system scales reasonably. We note that a considerable portion of the measured execution time is spent to load and use SIGARs libraries, which shows a good choice for its abstraction capability but with a trade-off in terms of performance.

2) Consideration on the RDF Data Storage: The novelty in using TDB as a container for the metrics of the monitored resources requires verification if its use fits the purpose. We logged the time of storing RDF data and checked the behavior of data storage by increasing the number of VMs monitored (and therefore the number of information that have to be stored). The result of this experiment is shown in Figure 6. The initial high value is due to the creation and configuration of the TDB which occurs only during the monitoring system startup. The storing attempts are not affected by this overhead. The increase in the number of VMs by a factor of 10 changes the storing time by a factor of 2. This shows that in a real cloud setup with hundreds of running VMs, the use of TDB as a data backend is a feasible choice in providing a reasonable storing time and scalability.

3) Monitoring Overhead: Considering the purpose of SMACS, a significant aspect is the overhead introduced by the monitoring layer on top of the running cluster. A high overhead might compromise the ability of nodes to achieve the required performance and may make SMACS unsuitable for cloud systems. In our evaluation, we consider both the CPU and network overhead. In both cases, we present the data collected on the master node of the OpenStack cluster. The master node represents a hot-spot in our cluster since it is running the majority of OpenStack services. We have decided to focus on this node since it represents the worst-case for our monitoring solution.

Figure 7 illustrates the CPU usage time of the master node for the monitored and unmonitored case when the cluster is running 20 VMs. The estimated CPU overhead is between 3% to 4% on the master node of the OpenStack cluster and even lower on the slave nodes, which may be considered as a minimal overhead. In fact, the peaks in Figure 7 are mainly related to the OpenStack services and not directly related to our monitoring services.

The network usage of the master node is shown in Figure 8 and Figure 9. The peaks in these figures represent the traffic generated by the SMACS agents for the data collection task. Despite being remarkably higher than the unmonitored

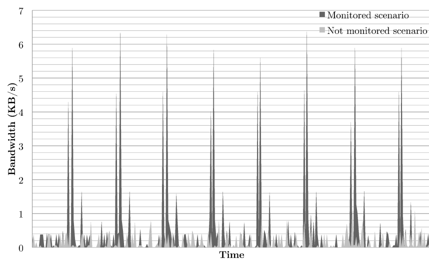

Fig. 8: Outbound network traffic overhead on master node

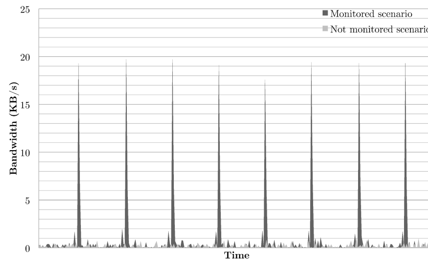

Fig. 9: Inbound network traffic overhead on master node

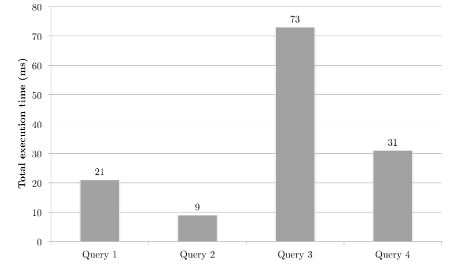

Fig. 10: Query execution time measures

scenario, it should be noted that the workload is a very low network-intensive one and that a peak of 18 KByte/s of receiving overhead is reasonable for most of the real-world scenarios. The low network-intensive scenario enables us to investigate and focus on this benchmark to study the overhead of our agent-based monitoring system. The difference of used bandwidth between the inbound and the outbound traffic is due to the diversity of the messages exchanged. A reply from an agent contains much more information and all of them have to be forwarded through the master node to the *Data Provider*. Overall, the results show that SMACS incurs minimal monitoring overhead for CPU and network resources.

B. Testing the Functionality

We also evaluate the execution of a significant set of queries using the *Resources Information Endpoint* against the *Data Storage* where metrics have been collected. For each of them we provide the starting query expressed with SPARQL syntax, the resulted table, and a measurement of the total execution time. Contrary to the results present in Section V-A, we use only one workstation in this experiment. The aim of this experiment is to demonstrate the unified access to the collected data from heterogeneous systems for verifying the correctness of the dataset, rather than analyze other performances metrics, such as, scalability, overhead, etc. Therefore, this experiment is configured with:

- All-in-one OpenStack installation with basic services (Nova, Keystone, Glance, Ceilometer) with three active VMs;
- CloudFoundry instance running on the first VM;
- CentOS 6.3 instances running on the second and third VMs. These instances host SMACS *Data Agent*;
- RHEL 6.5 instance running on Amazon Web Services and monitored by CloudWatch.

1) Sample Queries and Results: In order to provide quantitative results for the total query execution times for the 4 sample queries shown in Table I, SMACS is setup on a Dell Optiplex 780 configured with an Intel Core 2 Duo CPU E8400 @3.00GHz (2 cores) CPU, 4 GB memory and running CentOS 6.5 as operating system. All queries are executed three times after one warm-up query and the average of the three measures are shown in Figure 10. As expected, the information retrieval times are extremely low for SPARQL query engine and the TDB storage system. For Query 3, we note that the use of the FILTER expressions impact the total execution time. Therefore, for a more complex query, the usage of the FILTER functions must be seriously considered, since it may impact the execution time.

#	Description	SPARQL query
1	What kind of information is there available?	SELECT DISTINCT ?type WHERE { [] rdf:type ?type }
2	What kinds of information for an application are there available?	SELECT DISTINCT ?property WHERE {?s a smacs:Application; ?property []}
3	Retrieve measurements of resident memory for a CloudFoundry job named "nats". The results are filtered and sorted by timestamp.	SELECT ?usedMemory ?time WHERE {?jid a smacs:Process; smacs:reference ?reference. FILTER regex(?reference, "CloudFoundry") ?jid smacs:name ?jname. FILTER regex(?jname, "nats") ?jid smacs:memory ?usedMemory; smacs:timestamp ?time FILTER (?time > "2014-03-06T12:00:00"^^xsd:dateTime && ?time < "2014-03-06T12:59:59"^^xsd:dateTime) } ORDER BY ?time LIMIT 6

4	Get names of the monitored virtual machines, regardless the provider (Amazon or OpenStack). For each VM, the result might also contains the running applications names, sorted on the VM names.	SELECT DISTINCT
		?vmName ?provider ?applicationName
		WHERE {
		?vid a smacs:VirtualNode;
		smacs:name ?vmName;
		smacs:reference ?provider.
		OPTIONAL{
		?appid smacs:reference ?vname;
		smacs:name ?applicationName. } }
		ORDER BY ?vmName

TABLE I: Example queries

VI. Conclusions

This paper proposes a novel approach based on Semantic Web concepts for monitoring heterogeneous cloud systems. Compared to traditional monitoring systems, which are typically based on relational database systems, there are many aspects of Semantic Web technologies which make them suitable for integrating data from globally distributed, heterogeneous, and autonomous data sources. The proposed framework, SMACS, is based on a server-agent architecture and uses a set of brokers to communicate with several cloud components at different IaaS/PaaS/SaaS levels, translating data from heterogeneous data sources to RDF by leveraging the highly expressive RDF and SPARQL standards. The evaluation of SMACS using CloudWatch, Ceilometer, and Monit, shows that it is highly flexible and extendable and can be used with many information sources, monitoring tools, and cloud systems.

The future extensions of SMACS involve different aspects. The first one is to extend the set of supported monitoring tools increasing the number of available data sources. Furthermore, the wide spreading of Monitoring as a Service (MaaS) may help in gathering information from different systems. The second one is to investigate other possibilities on the semantic implementation. Finally, the third one is to manage the increasing complexity of SMACS due to new data sources and brokers, by employing RDF meta models and OWL ontologies.

References

[1] Rackspace Cloud Computing, "OpenStack Cloud Software," 2012, http://www.openstack.org/.

[2] Nagios Enterprises LLC, "Nagios - The Industry Standard in IT Infrastructure Monitoring," 2014, http://www.nagios.org.

[3] H. Huang and L. Wang, "P&p: A combined push-pull model for resource monitoring in cloud computing environment," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, July 2010, pp. 260–267.

[4] A. Ciuffoletti, "Monitoring a virtual network infrastructure: An iaas perspective," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, pp. 47–52, Oct. 2010.

[5] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *Proceedings of the 2009 IEEE International Conference on e-Business Engineering*, ser. ICEBE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 281–286.

[6] J. Povedano-Molina, J. M. Lopez-Vega, J. M. Lopez-Soler, A. Corradi, and L. Foschini, "Dargos: A highly adaptable and scalable monitoring architecture for multi-tenant clouds," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2041 – 2056, 2013.

[7] S. De Chaves, R. Uriarte, and C. Westphall, "Toward an architecture for monitoring private clouds," *Communications Magazine, IEEE*, vol. 49, no. 12, pp. 130–137, December 2011.

[8] P. Hasselmeyer and N. d'Heureuse, "Towards holistic multi-tenant monitoring for virtual data centers," in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, April 2010, pp. 350–356.

[9] S. Clayman, R. Clegg, L. Mamatas, G. Pavlou, and A. Galis, "Monitoring, aggregation and filtering for efficient management of virtual networks," in *Network and Service Management (CNSM), 2011 7th International Conference on*, Oct 2011, pp. 1–7.

[10] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data-the story so far," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 5, no. 3, pp. 1–22, 2009.

[11] G. Klyne and J. J. Carroll, "Resource description framework (RDF): Concepts and abstract syntax," World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210, February 2004.

[12] O. Lassila and R. R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification," W3C Recommendation, February 1999. [Online]. Available: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[13] Amazon, "Amazon Elastic Compute Cloud (Amazon EC2)," http://www.amazon.com/b?ie=UTF8&node=201590011.

[14] R. N. T. Metsch, A. Edmonds and A. Papaspyrou, "Open cloud computing interface - core," Open Grid Forum, Tech. Rep., 2011. [Online]. Available: http://www.ogf.org/documents/GFD.183.pdf

[15] "Cloud infrastructure management interface (cimi) model and rest interface over http," Distributed Management Task Force (DMTF), Tech. Rep., 2012. [Online]. Available: http://dmtf.org/sites/default/files/standards/documents/DSP0263_1.0.0.pdf

[16] VMware, Inc., "Hyperic SIGAR API," 2014, http://www.hyperic.com/products/sigar.

[17] H. Kreger, W. Harold, and L. Williamson, *Java and JMX: Building Manageable Systems*. Boston, MA: Addison-Wesley, 2003, ISBN: 978-0-672-32408-6.

[18] G. Antoniou, , G. Antoniou, G. Antoniou, F. V. Harmelen, and F. V. Harmelen, "Web ontology language: Owl," in *Handbook on Ontologies in Information Systems*. Springer, 2003, pp. 67–92.

[19] E. Prud'hommeaux and A. Seaborne, "Sparql query language for rdf," Latest version available as http://www.w3.org/TR/rdf-sparql-query/, January 2008.