Future Generation Computer Systems 29 (2013) 2041–2056

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds

Javier Povedano-Molina [a,*], Jose M. Lopez-Vega [a], Juan M. Lopez-Soler [a], Antonio Corradi [b], Luca Foschini [b]

[a] Dpto. Teoría de la Senal, Telemática y Comunicaciones, Universidad de Granada, Granada, Spain
[b] Dipartimento di Elettronica, Informatica e Sistemistica, University of Bologna, Bologna, Italy

HIGHLIGHTS

- We propose a decentralized architecture for monitoring Clouds.
- The proposed architecture uses a hybrid pull/push strategy to disseminate data.
- Our architecture is flexible, scalable and support multiple tenants.
- Our architecture uses standard technologies, such as REST, JSON and DDS.

ARTICLE INFO

Article history:
Received 30 April 2012
Received in revised form
15 April 2013
Accepted 17 April 2013
Available online 6 May 2013

Keywords:
Cloud monitoring
Cloud management
Data-centric communications
Publish–subscribe
DDS
Distributed Cloud management

ABSTRACT

One of the most important features in Cloud environments is to know the status and the availability of the physical resources and services present in the current infrastructure. A full knowledge and control of the current status of those resources enables Cloud administrators to design better Cloud provisioning strategies and to avoid SLA violations. However, it is not easy to manage such information in a reliable and scalable way, especially when we consider Cloud environments used and shared by several tenants and when we need to harmonize their different monitoring needs at different Cloud software stack layers. To cope with these issues, we propose Distributed Architecture for Resource manaGement and mOnitoring in cloudS (DARGOS), a completely distributed and highly efficient Cloud monitoring architecture to disseminate resource monitoring information. DARGOS ensures an accurate measurement of physical and virtual resources in the Cloud keeping at the same time a low overhead. In addition, DARGOS is flexible and adaptable and allows defining and monitoring new metrics easily. The proposed monitoring architecture and related tools have been integrated into a real Cloud deployment based on the OpenStack platform: they are openly available for the research community and include a Web-based customizable Cloud monitoring console. We report experimental results to assess our architecture and quantitatively compare it with a selection of other Cloud monitoring systems similar to ours showing that DARGOS introduces a very limited and scalable monitoring overhead.

1. Introduction

Cloud computing architectures have attracted more and more attention in the last years and many companies are looking at them for feasible solutions to optimally exploit their IT infrastructures [1]: part of the success of Cloud computing is due to its ability to provide virtualized resources on-demand according to current computation requirements. Such approach, inspired by the utility computing paradigm, assumes that computing resources are goods, such as gas, electricity, and water, and that they should be obtained and bought on a pay-per-use basis. This rationale contrasts with the classical IT approach where companies had to forecast the expected resource usage to properly dimension their IT infrastructure.

Apart from its intrinsic advantages, the new model raises also new challenging issues. First, incorrect system estimations might lead to either over or under resource usage. Moreover, applications moved to the Cloud may exhibit very heterogeneous resource needs that depend not only on service types – which may range from Web Services to high performance computation

* Correspondence to: Escuela Técnica Superior de Ingenierías de Informática y Telecomunicación, C\ Periodista Daniel Saucedo Aranda S/N, 180071-Granada, Spain. Tel.: +34 958241717.
E-mail addresses: jpovedano@ugr.es (J. Povedano-Molina), jmlvega@ugr.es (J.M. Lopez-Vega), juanma@ugr.es (J.M. Lopez-Soler), antonio.corradi@unibo.it (A. Corradi), luca.foschini@unibo.it (L. Foschini).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.04.022

offloading from mobile nodes, among many other ones [2–4] – but also on dynamic conditions and usage patterns imposed by final users, typically difficult to predict. In addition, emerging Cloud deployment models, such as private and community Clouds, might require that different tenants (or users) have different views of the same Cloud infrastructure resources, namely, physical hosts and Virtual Machines (VMs). Hence, it is essential to provide Cloud tenants a choice of monitoring components to present, extract, and meld together monitoring data with different levels of details. Although that flexibility would be highly beneficial, it requires enabling fine-grained monitoring mechanisms with scalable many-to-many communications. Finally, it is crucial to provide Cloud administrators with appropriate easy-to-use and easy-to-customize data center resource monitoring tools that enable timely reactions to potential problems.

To address all above issues, this paper proposes a monitoring architecture based on scalable data-centric publish/subscribe paradigm specifically designed for highly customizable Cloud monitoring, called Distributed Architecture for Resource manaGement and mOnitoring in cloudS (DARGOS).

Our approach exhibits the following core original aspects. First, to enable effective Cloud monitoring in multi-tenant Cloud scenarios, it allows customizing both granularity and frequency of received monitored data according to specific service and tenant requirements. Second, it is based on standardized communication technologies, and more precisely on the Data Distribution Service (DDS). Therefore, it eases interoperability with DDS legacy applications and tools and avoids vendor "lock-in" [5]. In addition, the use of DDS eases defining flexible QoS policies within the monitoring task itself that hide the communication issues to the application logic, and provide efficient, reliable and timely Cloud monitoring data delivery. Third, DARGOS has been implemented as an open-source tool based on the open-source OpenStack Cloud platform and is made available to the Cloud community.[1]

In order to better underline the benefits and original aspects of DARGOS, the paper presents a thorough survey of similar systems and provides qualitative and quantitative comparisons with a selection of relevant Cloud monitoring systems very close to ours. After the experimental evaluation, we claim that DARGOS outperforms other benchmarked systems in terms of performance and scalability.

The paper is structured as follows. Section 2 defines the main requirements and design guidelines for a Cloud monitoring system and overviews a thorough selection of related research efforts in the field. Section 3 describes the proposed architecture and its core functionalities. Section 4 studies some interesting implementation issues and details main components, such as our novel Web-based monitoring console. Section 5 shows the obtained experimental results that assess our architecture. Finally, Section 6 concludes the paper presenting directions of ongoing research.

2. Cloud monitoring: background and related work

Cloud computing architectures require complex monitoring infrastructures to transparently deal with the extreme complexity introduced by specific application and system monitoring needs of multi-tenant Cloud environments. This section introduces some background knowledge for a better understanding of the area. Section 2.1 defines requirements and guidelines for Cloud monitoring infrastructures. Section 2.2 presents an in-depth discussion about the related work in this area: it details different monitoring problems and optimization goals so as to offer a complete overview of the current ongoing research efforts. Finally, Section 2.3 gives some needed background material about Object Management Group (OMG) DDS publish/subscribe standardized middleware.

2.1. Requirements and emerging design guidelines

Monitoring is a core function of any integrated network and service management platform. Cloud computing makes monitoring even a more complex infrastructure support function, since it includes multiple physical and virtualized resources and because it spans several layers of the Cloud software stack, from Infrastructure as a Service (IaaS) to Platform as a Service (PaaS), and Software as a Service (SaaS).

First of all, Cloud pushes to the extreme the concept of resource sharing in an environment that is inherently highly dynamic and with loads that are difficult to predict; in addition, the setting also may change very often due to the typical pay-per-use requirement. Clouds are sometimes deployed in many data centers and clusters of servers, each of them possibly equipped with different characteristics and capabilities: for instance, in the same Cloud there could be clusters optimized for CPU intensive computation, such as media transcoding and data indexing, while others will be optimized for I/O throughput, such as Web applications provisioning and media storage. Consequently, a Cloud monitoring system should be aware of logical and physical groups of resources and should organize monitored resources according to certain criteria so as to separate and to localize the monitoring functions.

Another crucial issue is system scalability in terms of processing and bandwidth overhead: for instance, medium-size Clouds data centers typically include hundreds of physical hosts and thousands of VMs; even worst, big ones can easily go beyond thousands of physical hosts with multiple VMs equipped with several physical and logical sensors collecting monitoring data. These frameworks can eventually generate a great amount of network traffic that consumes precious network bandwidth; hence, the monitoring support should be as least intrusive as possible by adopting lightweight processing and communication solutions that limit the additional overhead. It also must assure timely monitoring data delivery.

Finally, for multi-tenant deployments, the monitoring system must also verify different requirements in terms of information granularity, accuracy, and update frequency. For instance, choosing a unique update rate for all interested nodes may cause intolerable overload. Therefore, the monitoring infrastructure should be flexible enough to accommodate heterogeneous application-related requirements and to harmonize various tenant needs.

Several solutions have been proposed in the last decade aimed to address all above issues. In the following, we identify recently emerged design guidelines for the development of proper Cloud monitoring infrastructures.

A first important direction is given by the adopted *distributed architecture* model. The *client/server* is surely the simplest one, with a direct interaction between monitoring clients and servers [6]. The *client/agent/server* model evolves from *client/server* and typically, for the sake of scalability, splits the unique server in an overlay of multiple agents acting as mediators between clients and servers [7–9]. The *publish/subscribe* paradigm enables many-to-many interactions where each entity can declare the monitoring data it is interested in. The publish/subscribe can either interpose an *event channel* between publishers and subscribers or adopt highly optimized *peer-to-peer* implementations [10].

A second design guideline is indicated by the adopted *interaction* model to transfer monitoring data between monitored (i.e., server/publisher) and monitoring entities (i.e., client/subscriber).

[1] Additional information, experimental results, and the DARGOS prototype code are available at: http://tl.ugr.es/dargos/.

Existing Cloud monitoring systems in the literature employ both *push* and *pull* models. They respectively require, either the monitored entities to asynchronously send the monitoring updates, or, alternatively, the monitoring ones to synchronously request it. More recently, mixed *push-and-pull* models also keep appearing [11].

A third guideline direction is defined by the selected distributed *measurement update strategy*. Almost all systems support *periodic* updates. Nonetheless, this strategy tends to waste precious bandwidth resources, especially when monitored resource usage is stable for long time intervals. Therefore, to further improve monitoring scalability, some systems also support *event-based* updates. In this case, monitoring information is only distributed if there is significant resource usage change or for specific alert conditions [10,11].

Last but not least, other important related aspects are the following ones. The *exchange format* used to express monitoring data that may range from simple plain text to more structured formats, such as the standard eXternal Data Representation and other proprietary representations [9,12]. The possibility of defining *filters* for incoming data content as well as defining *time delivery* constraints. Finally, while most monitoring systems are *statically configured*, some systems are starting to recognize the importance of including a *discovery function* to dynamically and flexibly bind monitoring entities to monitored ones without any need of previous static knowledge; that feature simplifies the introduction of new monitoring metrics and (logical) aggregated sensors [7].

2.2. Related work

In the last few years, we have witnessed a growing interest in management solutions for Cloud, not limited only to academia, but also driven by companies that adopt more and more Cloud-based solutions to flexibly provide their services and to increase their revenues.

In the following, without any pretense of exhaustiveness, we overview and present a selection of resource monitoring infrastructures for effective and scalable monitoring of Cloud environments. Our analysis considers the design directions introduced in the previous section.

We start presenting solutions designed for monitoring traditional large-scale IT infrastructures, then we overview monitoring facilities for grid environments – often employed also for Cloud –, and we terminate with a larger selection of very recent seminal monitoring infrastructures specifically proposed for Cloud. The section ends with a brief qualitative comparison of surveyed systems shown in Table 1.

Nagios is a very popular server monitoring service widely spread in data centers to monitor the status of services and resources [12]. Nagios adopts a centralized *client/server* architecture in which the central server is responsible of gathering monitoring information from remote monitored nodes for the sake of storage and visualization. Nagios supports two main deployment alternatives for local resource monitoring. Namely, the Nagios Remote Plugin Executor (Nagios NRPE) – installed within every monitored resource – enables periodic *pull* updates through request–reply interactions triggered by a central server; and the Nagios Service Check Adaptor (Nagios NSCA) which supports asynchronous *push* measurements and events from monitored nodes to the central server. 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, in highly dynamic scenarios. In addition, it does not adopt any standard format for monitoring data representation.

Ganglia [9] is another relevant system widely employed to monitor grids and clusters of computers. To achieve scalable monitoring of clusters, Ganglia adopts a hierarchical *client/agent/server* distributed architecture: each monitored node runs an agent, called *gmond*, that sends information about resource usage upon request to another agent, called *gmetad*, acting as an aggregator and executing at a higher level of the hierarchy. Moreover, Ganglia allows both *push* and *pull* interaction models between agents, and supports both XDR and XML data *exchange formats*. At the same time, Ganglia lacks some relevant features such as *discovery function* at inter-cluster level.

More recently, infrastructures for monitoring distributed grids have started recognizing the importance of *dynamic discovery*. MonALISA enriches the traditional *client/server* architecture via the introduction of a centralized registry used to dynamically discover the entities to be monitored [13]; in addition, it supports both *pull* and *push* interaction models, data *filtering*, and standard XDR-based data *exchange format*. Hyperic HQ is a monitoring solution developed by VmWare Inc specifically designed to support the discovery of local resources and services, but it still relies on a centralized *client/agent/server* architecture and monitored hosts have to be explicitly added and registered [7].

Another example of *push*-based monitoring architecture is [17], a monitoring system for grids with a high accuracy coupled with low bandwidth consumption. This feature is achieved by dynamically adjusting the period between measurements according to the running average of time intervals between information changes. This system also *filters* the updates delivered to consumers according to their value: if changes are insignificant they will not be delivered, thus saving resources.

Based on the GRID superscalar infrastructure (GRIDSs), [18] presents a *pull*-based monitoring and steering system for grid applications. This proposal – with some scalability limitations – relies on a unique server for processing and storing the monitoring information generated by the agents. Furthermore, [19] uses *Complex Event Processing* (CEP) for aggregating multiple real-time streams of monitoring information.

Another possible metric to be considered in grid computing is the level of QoS fulfillment. In this respect, GRIDIFF proposes a software architecture that covers all necessary steps to integrate QoS into the process of allocating resources for the execution of jobs in the grid [20].

Monitoring resource usage in Clouds is very important to achieve resource optimization goals [21]. In this sense, based on OpenNebula, the Claudia project [22] proposes the creation of an abstraction layer for distributing services across multiple Clouds. Claudia allows the automatic deployment and escalation of services depending on the service status (not only on the infrastructure). More recently, the OPTIMIS toolkit [23] also addresses the problem of accessing multiple Clouds simultaneously. OPTIMIS proposes optimizing multiple-Cloud systems by using two different entities called Cloud Optimizers (CO) and Service Optimizers (SO). Cloud Optimizers monitor and optimize the performance of any Cloud, while Service Optimizers gather information about multiple Clouds and decide which Cloud should perform a specified service. Finally, [24] proposes a Cloud management platform for VM consolidation based on OpenStack, by using CPU, power, and network usage metrics to optimize VM consolidation. OPTIMIS relies on a centralized monitoring component which processes the monitoring information and makes VM placement decisions.

About resource monitoring solutions in Clouds, [11] proposes a hybrid *pull-and-push* approach to monitor Cloud resources. A function allows switching the communication strategy according to user needs in terms of consistency and efficiency, to tailor monitoring information to user needs. However, it focuses only the interaction model (*pull/push*) and does not tackle the problem of scalable and timely data distribution itself.

Another interesting work proposes a distributed monitoring for load balancing in virtual networks based on network metrics

Table 1
Monitoring infrastructures: a comparison of main characteristics.

	Distributed architecture	Interaction model	Discovery	Filtering	Exchange format	Measurement update strategy
Nagios NSCA [6]	Client/server	Push	No	–	Plain text	–
Nagios NRPE [8]	Client/server	Pull	No	–	Plain text	Periodic
Ganglia [9]	Server/agent	Push/pull	Cluster level	–	XDR/XML	Periodic/event
Monalisa [13]	Service/client	Pull/push	Registration	Yes	XDR	Periodic
Hyperic HQ [7]	Server/agent	Pull	Only sensors	–	Proprietary	Periodic
PCMONS [14]	Server/agent (Nagios Plugin)	Push	No	–	Plain text	Periodic
Lattice [10]	Publish/subscribe	Push	Yes (multicast)	Time	XDR	Periodic/event
[15]	Publish/subscribe	Push	–	Yes	Proprietary	Periodic/event
[16]	Publish/subscribe	Push	No	Yes	Proprietary	Periodic

gathered by agents deployed at each virtual network interface [25]. Nonetheless, this work only focuses on the network resources of Cloud without considering other local resources, such as CPU and memory.

A *push*-based approach is proposed in [26]. It 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, namely, active sessions, without taking into account hardware resources and, additionally, it does not support any monitoring at the infrastructure level.

PCMONS is an open-source *pull*-based Cloud monitoring solution [14]. It relies on a monitoring server that receives information from multiple monitoring data integrators that *pull* Cloud monitoring information in their turn. As a disadvantage, this proposal relies on a unique server for processing the obtained monitoring information so impairing its scalability. [15] proposes an *agent*-based system to monitor resources in multi-tenant Cloud environment where any Cloud is administered by multiple cooperating entities. It includes a Data Stream Management System (DSMS) along with a message bus to distribute monitoring data tailored to each user needs. Similarly to DARGOS, this proposal supports one-to-many communications but does not support any delivery guarantee.

Resources and Services Virtualization without Barriers (RESERVOIR) is a project funded by the European Framework Programme 7 (FP7) that aims to enable massive scale deployment and management of complex IT services across different administrative domains, IT platforms, and geographies [27]. Within the context of this project, [10,28] propose Lattice, an infrastructure for Cloud monitoring that adopts a *publish/subscribe*paradigm to disseminate monitoring information originated from Cloud nodes. Thus Lattice supports also monitoring scenarios with multiple producers and consumers. Lattice is the proposal in the literature most similar to DARGOS: it has been specifically optimized to be highly scalable so to perform complex monitoring tasks on large scale virtual networks [29]; however, it does not tackle fault-tolerance issues, such as reliability of data dissemination, that are left as part of future work.

More recently, [16] proposes a distributed *peer-to-peer* monitoring framework that retrieves monitoring information from multiple sources, such as Nagios and Ganglia log files, to provide self-management capabilities. This proposal adopts sophisticated query mechanisms to support *filtering*. However, this work does not focus on data distribution aspects and uses non-standardized communication technologies.

Let us conclude this section with some remarks about all surveyed solutions. Table 1 reports the main features of every studied system. First of all, most of the overviewed monitoring systems rely on centralized architectures, where a central server gathers and stores monitoring data relative to all node resources and services making them available to interested parties through a well-defined interface. The centralized approach suits many scenarios but is not scalable and clashes with fault tolerance, especially when the number of services interested on monitoring increases. Some systems face this problem by using the *publish/subscribe* paradigm [10,15]; in this case, monitored nodes push resource information directly into each node interested in remote monitoring data, thus avoiding any requirement of centralized servers. However, in some scenarios, the *publish/subscribe* approach needs source filtering capabilities based on either data rate or content to adapt data updates to subscriber requirements. In brief, to the best of our knowledge, current systems tend to use a global filtering approach, instead of supporting a more flexible per-subscriber approach. Finally, the discovery of resources available for monitoring is also an important issue in modern Cloud environments. Typically, the process of adding new resources into the Cloud is done by specifying resource locations and centralized servers into configuration files. Some systems tried to reduce human intervention by adding *automatic discovery* features. For instance, MonALISA [13] uses an independent registration service, while Lattice [10] automatically discovers data sources by using multicast. Another relevant aspect of distributed monitoring is the ability of handling asynchronous events, such as generating alarms or triggering events. In this sense, only [6–8, 10,13,15] consider asynchronous event notification, while the rest of the pack leaves that anomalous situations and asynchronous events to be handled at application level, hence delegating more complexity to the application logic.

To cope with and to overcome all above issues, DARGOS design combines a *publish/subscribe* decentralized architecture, *automatic discovery*, *filtering* capabilities, and *asynchronous event* handling to achieve real-time monitoring of resources in Cloud environments. We believe that these features along with the use of standard technologies and well-defined and flexible data *exchange formats* may represent a big step forward in the Cloud monitoring research area.

2.3. Data Distribution Service (DDS)

We conclude the section providing some necessary background material about the DDS standard used in DARGOS. Proposed by the Object Management Group, DDS adopts a topic-based publish/subscribe communication model. According to the DDS standard, a topic is a data atom that is shared between data producers and consumers. Topics are fully defined by their name and type.

Data producers and consumers, respectively called publishers and subscribers, are completely decoupled (in space and in time) peer entities that exchange topic samples within a Global Data Space (GDS), according to a contract established in a discovery phase. Topic samples are exchanged between publisher and subscriber always satisfying the QoS requirements imposed by the peers [5].

Publishers and subscribers discover each other automatically and match whenever they have a compatible topic and QoS (see Fig. 1). The underlying DDS data distribution overlay is completely decentralized and adopts a peer-to-peer model that does not

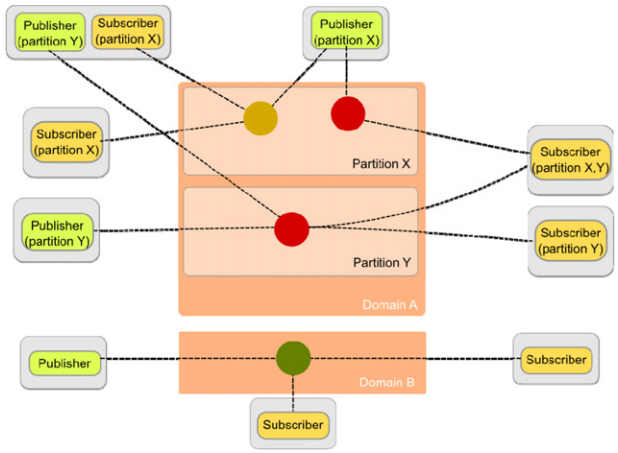

Fig. 1. DDS architecture and main entities. Colored balls represent different topics. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

require any centralized broker; thus, it grants high reliability and robustness because any failure of participating nodes does not compromise the whole system.

In addition to the distributed nature of DDS-based applications, other aspects of this middleware are crucial for the monitoring of Cloud environments, such as the *caching mechanism* specified by the DDS wire-protocol, the Real-Time Publish–Subscribe protocol (RTPS). In particular, for each topic data sample pushed by the publishers in the GDS, the middleware delivers it to all subscribers that maintain a copy of it in their local caches. In other words, DDS supports *push* interactions only, mainly to save precious network resources.

DDS can deal with multiple samples, called instances (differentiated by an associated key), that are maintained at the same time on the cache; cached entries continue to be available even when the originating publisher is no longer alive. In other words, each subscriber maintains a local database that contains the latest values of each topic instance and can locally perform query operations without generating any network traffic.

DDS also provides *built-in* data isolation mechanisms – called *domains* and *partitions* – to improve system scalability. Middleware nodes can be organized into physical (domains) and logical (partitions) groups and hierarchies to promote multiple views of the same physical DDS deployment, namely, to logically separate subscriptions within a domain depending on partition membership, as shown in Fig. 1.

Moreover, DDS includes *time and content filtering* capabilities to limit data sample exchanges, for instance to suppress notifications that exceed a certain frequency and whose content does not match a given condition.

The DDS specification defines a *data-centric* model that decouples data sources and consumers: publishers and subscribers share the same data model for data (topics) and can build distributed architectures where the local application logic is decoupled from the data model. The data-centric approach of DDS, along with the set of QoS policies defined by the OMG standard specification, makes it a good candidate middleware for Cloud monitoring and management.

Toward that goal some DDS features are relevant. First, it provides built-in topics for automatic entities discovery, so adding new publishers/subscribers is completely user-transparent and new entities discover other remote entities without any need of a broker. Second, it has filtering capabilities to optimize the usage of resources such as network bandwidth by sending only relevant data to subscribers. For instance, DDS publishers do not send topic samples that do not match a filtering criteria imposed by subscribers. Finally, through domains and partitions, DDS allows to physically/logically divide the GDS. In this way, it can isolate both unrelated entities and the network load, thus facilitating the design of highly scalable support for data distribution in Cloud monitoring.

3. DARGOS for flexible and scalable multi-tenant Cloud monitoring

DARGOS is a flexible and robust monitoring support for Cloud environments, built atop of the data-centric DDS push-based publish/subscribe paradigm. DARGOS organizes monitoring among Cloud nodes and interested peers in a fully-distributed manner, enabling timely, flexible, and reliable monitoring in highly dynamic and multi-tenant Cloud provisioning scenarios.

DARGOS adopts a publish/subscribe distributed architecture based on two entities called Node Monitor Agent (NMA) and Node Supervisor Agent (NSA). NMAs are responsible of gathering monitoring data from the local node and deliver these data to interested nodes. NSAs are responsible of collecting monitoring data from remote hosts and make them available to final system administrators through DARGOS local API and visualization tools, as it is shown in Fig. 2.

NMAs and NSAs communicate directly by using a decentralized approach based on the DDS data-centric middleware. NMAs publish monitoring data according to the criteria established by the Cloud administrator and NSAs are responsible of receiving, storing, and maintaining the monitoring data received in a local cache that represents the status of all sensors coming from remote NMAs.

One of core advantage of the data-centric architecture is its reliability when either one NMA or one NSA crashes. These potential faults do not affect the whole system, but only the directly interconnected entities, thus making this approach more robust than those based on one centralized server.

To improve the scalability of Cloud data centers, hosts are usually deployed hierarchically, typically adopting tree-based networking topologies where hosts in the same group are considered leaves of the same router [30–32]. In the following, we call *zones* these host groups, and we use zones to support multiple tenants. Each tenant is associated with a certain zone, so any tenant can have her own view of the Cloud, although many tenants may share the same Cloud. We associate one NMA with one zone, but at the same time, one NSA could be interested in receiving monitoring information coming from multiple zones. To support this behavior, as shown in Fig. 3, NSA can associate with multiple zones. Even though, the Cloud administrator must keep in mind that the more the zones NSA is associated with, the higher the bandwidth requirements for the reception of related monitoring information.

Focusing on data representation and communication aspects, DARGOS uses DDS. In our proposal, each monitored host keeps track of its most relevant physical and virtual resources and publishes their statistics by using dedicated topics. For instance, if one NSA is interested in receiving the CPU usage of a certain node, it has to declare interest in the corresponding CPU topic. Similarly, the Hypervisor topic represents the status of the hypervisor controlling VM in a certain physical compute node. To identify topics coming from the same host, DARGOS associates each topic sample to a Universal Unique Identifier (UUID) [33]. This one-to-one mapping between logical monitoring sensors and topics enables a fine-grained specification of monitoring requirements, including also update rates, reliability aspects, etc. It also permits monitoring contracts to be tailored for specific tenant monitoring needs: for instance, while CPU usage is very dynamic, the usage of swap memory is usually highly static, which makes their publication rates very different, namely high-rate for CPU and

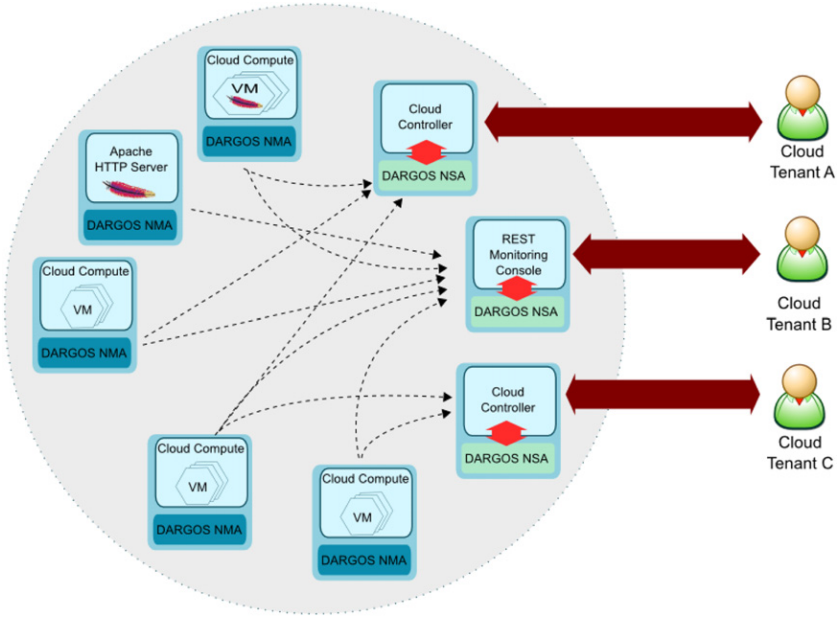

Fig. 2. DARGOS global architecture. Each DARGOS NSA monitors a group of nodes and provides access to monitoring information to multiple tenants through different APIs.

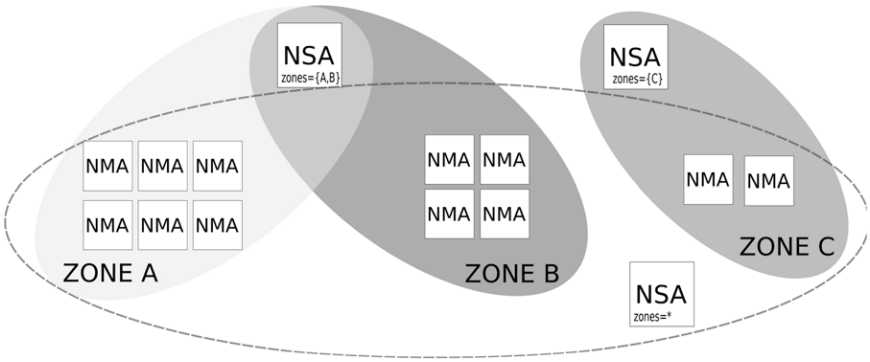

Fig. 3. Cloud zone management. Each NMA is associated with one zone only. Each tenant has an NSA that can receive monitoring information from multiple zones.

low-rate for memory. In addition, such fine-grained approach allows subscribers to specify the sensors they are interested in, by subscribing appropriate topics only, and without any need of being statically bound to them, as shown in Fig. 4.

Hence, to further improve system scalability and to minimize the traffic overhead (e.g., a logging application interested in every resource being used in the Cloud) DARGOS also allows declaring interest in all logical sensors within the same host. This is achieved through a unique topic sample called *HostSummary* (see Fig. 4). This approach, less flexible, contributes to reduce the number of topics needed to subscribe to, and consequently reduces communication overhead and network usage, both in the initial discovery procedure and later in data transmission itself.

With a finer degree of details, processing raw monitoring data might produce unnecessary overhead in many scenarios. For instance, a Cloud console application does not need to receive every update of monitor information: hence, sending each resource usage update to the console application NSAs is unnecessary and may congest the network and the application. In other cases, NSAs do not need an exact estimation of resource usage, as it is enough to know if the resource usage lies within certain thresholds: for instance, if CPU usage is low (e.g. [0–25]%) it might not be necessary for certain NSAs to receive periodic updates while the usage remains within that range.

DARGOS supports data filtering mechanisms aimed to reduce monitoring overhead by eliminating unnecessary monitoring data distributions. The first mechanism is a *time-based filter* used by NSAs to limit the number of updates accepted within a certain period of time from a given resource. For instance, a resource application that requires only 1 measurement per minute can limit the reception rate to such values. The second mechanism is a *value-based filter* that permits NSAs to be notified of updates only if the resource usage has significantly changed (with respect to a given set of user-defined thresholds) since the last published update. This strategy avoids unnecessary traffic in scenarios where an approximate estimation of resource usage is more than satisfactory.

4. Platform implementation and Web monitoring console

This section delves into the implementation details of DARGOS. Section 4.1 introduces DARGOS Application User Interface (API) for monitored nodes and monitoring points, respectively NMAs and NSAs. Section 4.2 explains the current DARGOS implementation for the OpenStack Cloud IaaS. Finally, Section 4.3 overviews a DARGOS-based Web-console tool to ease multi-tenant Cloud monitoring.

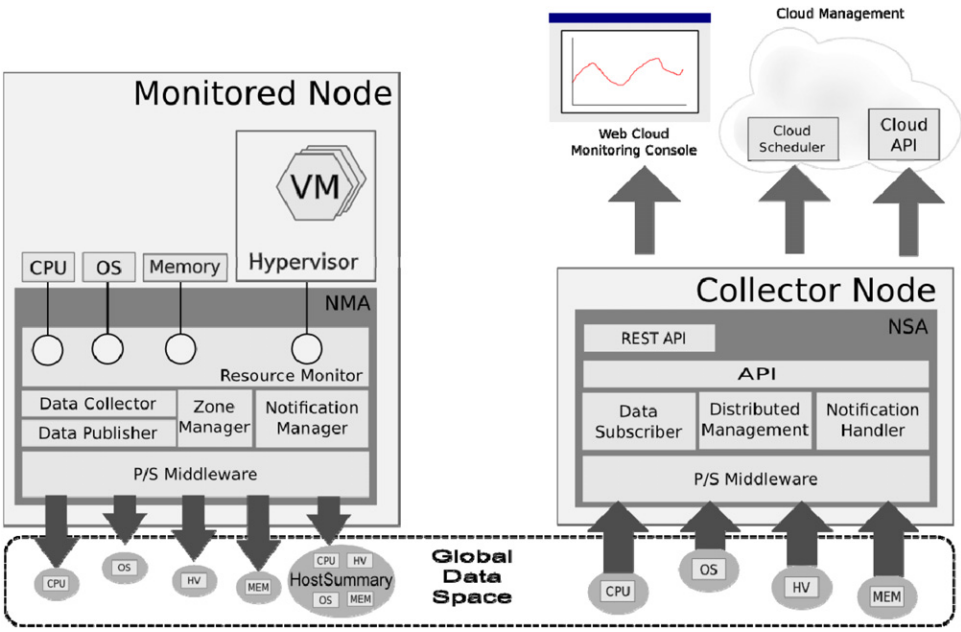

Fig. 4. DARGOS components. The NMA is installed in every monitored host. The NSA is installed in every node that accesses to monitoring data.

4.1. NMA and NSA DARGOS API

Although at the lowest layer DARGOS integrates with DDS via standardized C communication primitives, to increase DARGOS portability and flexibility, and to integrate with existing Clouds such as the OpenStack Cloud platform, we also provide a high-level DARGOS Python API to NMA and NSA. It aims at simplify the access to monitored information from scripts and command line interfaces.

First of all, let us remind that NMAs are the monitoring agents responsible of collecting monitoring data and delivering it to interested parties. NSAs are the access points for the user to Cloud monitoring data. Accordingly, the NSA component provides a flexible API that allows to read local monitoring data available in each node participating to the Cloud infrastructure. Following our main design guidelines, DARGOS API has been designed to be flexible and expressive enough to: (i) define the zones that a NSA wants to monitor, (ii) obtain the latest measured value for a certain resource, (iii) access to most recent historical measurements for a resource, (iv) verify the capabilities of resources discovered automatically by inquiring their status and by enabling the registration of handlers for remote event notifications.

Fig. 5 depicts the most relevant entities and API methods for NMA and NSA. On the one hand, NMA includes only a few methods to configure the monitoring agent as well as to manage logical monitoring sensors. On the other hand, NSA offers additional methods to ease resources usage reading and notification handling. For the sake of space limit, we refer readers interested in additional details about DARGOS API to the DARGOS project Web site.[2]

4.2. OpenStack-based DARGOS implementation

OpenStack is an open source project supported by a large community of developers and companies that allows creating and managing large-scale Cloud IaaS deployments. The overall OpenStack initiative consists of several parts and subprojects [34].

In this paper, we use the OpenStack Nova project for managing the virtualized IaaS environment. In a more detailed view, OpenStack Nova consists of multiple services that collaborate to maintain the necessary infrastructure and control the lifecycle of VMs based on user requests.

The *Cloud API service* is the frontend component that provides a REST interface to satisfy user requests. The requests are converted into Cloud actions such as instantiating and terminating a VM. To foster wide interoperability, the OpenStack API provides methods compatible with other major Cloud solutions, such as Amazon EC2 [35], thus also making possible to easily manage hybrid public–private Clouds.

The *Compute Service* controls VM lifecycle and configuration in any specific node. In particular, it commands the local hypervisor to create and terminate VMs.

The *Network Service* maintains and manages the virtual networks that interconnect VMs by enabling dynamic private and public IP addresses assignments. It routes the network traffic generated or consumed by the VMs. Finally, the *Scheduler service* decides where to allocate the resources necessary to instantiate a new VM.

Fig. 6 shows how the OpenStack services (the white boxes) interact. It also includes two other ancillary components. The Advanced Message Queuing Protocol (AMQP) server is an implementation of event channel for publish/subscribe AMQP-based communication model. It enables many-to-many interactions between Nova service components. Whereas, the SQL-based server acts as a centralized storage for all information needed for IaaS configuration. Since both AMQP and SQL servers are centralized, they may suffer scalability problems and are potentially single points of failure. As a result, OpenStack adopts a so-called "share-nothing" approach that minimizes communications between components and lets each component run without imposing specific dependencies to other Nova components. Similarly, to further reduce monitoring overhead, Nova maintains only a few resource indicators, typically updated using very long periods and shared among services using MySQL storage.

Albeit DARGOS may be also used in different Cloud environments, it is particularly suitable for OpenStack Nova, because it fully exploits its benefits. The changes we made to OpenStack were very simple and confirmed the ease-of-use and wide applicability

[2] Additional information, experimental results, and the DARGOS prototype code are available at: http://tl.ugr.es/dargos/.

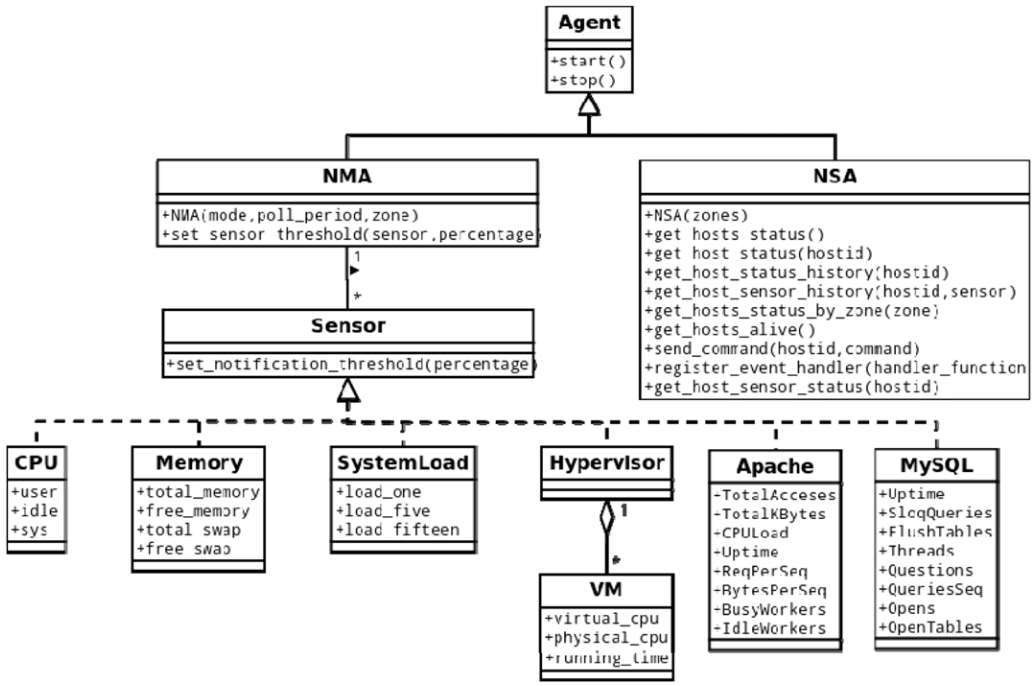

Fig. 5. DARGOS API class diagram.

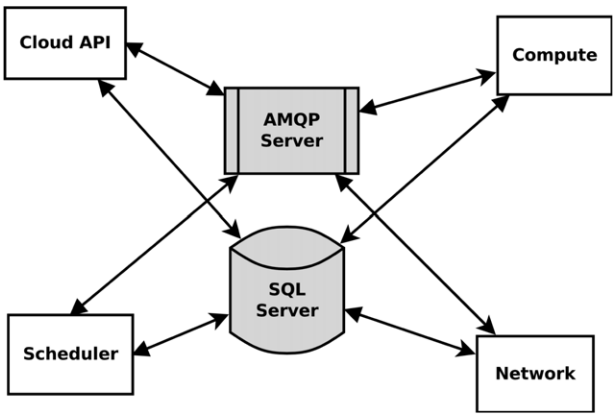

Fig. 6. OpenStack distributed architecture.

of our architecture. First, we attached one NMA to every Compute Service such that it collects physical node and hypervisor statistics. We also attached one NMA for monitoring other Nova services. In addition, we created a DARGOS-based scheduler that uses the resource statistics data collected by DARGOS NSA to take more dynamic resource allocation decisions. From our experience, we believe that applying DARGOS to other Cloud environments, different from OpenStack, is quite straightforward.

For a closer look to implementation details, DARGOS includes a set of built-in sensors that represent the most common system resources relevant in Cloud scenarios: CPU usage, memory usage, system load, and monitoring information about running VMs obtainable from the Hypervisor. The information provided by these sensors gives an idea of the system state of every node in the Cloud. It can be used to determine the current load of a specified physical node and to identify critical situations. However, in some scenarios monitoring only system resources may be not enough without doing the same for the applicative status of services, crucial to determine the health of Cloud deployments. For this reason, DARGOS implements also a set of built-in service sensors to monitor the status of most popular servers, such as Apache HTTP and MySQL database servers. These sensors collect application-related statistics, such as the number of requests per second and their uptime – for services that run either directly over the physical host or within a VM –, thus making possible to detect server overload situations and possible faults. Apart from available sensors, our architecture has been designed with extensibility in mind: more specifically, to add new sensors, either within the system or the VM, to fit new requirements is easy; to do it, one defines a new topic by following few simple steps.

Finally, due to the high dynamicity of resources usage, such as the CPU, sometimes monitored data can be very noisy and can present abrupt and short usage peaks that can lead to misleading monitoring conclusions when not properly smoothed. To avoid such situations, we have also implemented a family of smoothing functions based on recent historical data. More precisely, we have included three different monitoring data smoothing mechanisms: Average, Weighted Moving Average (WMA), and Exponential Weighted Moving Average (EWMA) [36–38]. These smoothing functions can be evaluated locally, without any additional overhead, by using recent historical data cached by the NSA.

4.3. Web-based DARGOS monitoring console

Our architecture gives access to the monitored data within the Cloud by using DARGOS API by default. However, providing open access to publish/subscribe infrastructure from outside the Cloud can lead to potential security and scalability problems, and requires some technical background about DARGOS and Python. Hence, to further facilitate its use, DARGOS includes also a Representational State Transfer (REST) API aimed to give access to the DARGOS monitoring data and, consequently, to increase its interoperability with Web technologies.

REST is a widely used standard that provides access to data resources. It uses an URL-based scheme, the Hypertext Transfer Protocol (HTTP), and other widely diffused technologies such as JavaScript Object Notation (JSON) to transfer data [39]. JSON is a lightweight exchange format used in several fields – ranging from Service Oriented Architectures (SOA) to Internet of Things (IoT) – because it enables exchanging data with minimum overhead

```
[{
  "last_update": {
    "tv_sec": 1335723764, "tv_nsec": 969164997
  },
  "zone": "zone-a",
  "mem": {
    "mem_free": 644190208, "swap_total": 4291817472, "mem_total": 4094332928, "swap_free": 4249845760
  },
  "hostname": "sunfire",
  "id": "415d43590fd841f8b8291722a59d371b",
  "vm_status": [],
  "cpu": {  "cpu_sys": 0.10,  "cpu_user": 0.0,   "cpu_idle": 99.89  },
  "system_load": {
    "load_one": 0.14, "load_five": 0.16, "load_fifteen": 0.21,
  }
}]
```


Fig. 7. DARGOS JSON example.

and fostering the widest interoperability. For instance, by simply accessing the http://nsahost/hosts URL, it is possible to obtain a JSON object containing the latest resource monitoring information of all the hosts currently monitored by the NSA installed at the machine nsahost. Similarly, the URLs http://nsahost/databases/ and http://nsahost/apache/ return the monitoring information relative to the current usage of database and Apache HTTP servers in JSON format. Fig. 7 shows an example of JSON-encoded DARGOS monitoring data.

Finally, our REST-based API eases the integration of DARGOS with other legacy services that support HTTP and JSON formats, such as Yahoo! Pipes [40].

The JSON exchange format is human readable, but it is still not easy to understand by users and typically requires further processing to build user-friendly views of monitored data. To cope with this issue, DARGOS provides a Web-based console that shows collected monitoring data in an intuitive and easy-to-read way, as shown in Fig. 8. It is available for the researcher community in a first open demo version.[3]

Focusing on technical details, the provided console consists of a local JavaScript application that automatically refreshes the information by performing periodical AJAX requests to NSAs via their REST APIs. The advantages of using a Web console application are manifold, but the most relevant are the following ones. (i) It enables accessing the monitoring data information from outside the data center without the need of having DARGOS installed. (ii) AJAX requests can be directed to multiple NSAs, thus it facilitates to cross different Cloud and hosts monitoring data. It also allows tailoring specific and multiple views of the same deployment for each tenant in multi-tenant Clouds. (iii) The integration with Yahoo! Pipes easily enables for the definition of new processing and data crossing functions, as well as for the integration of DARGOS into existing SaaS application mash-ups.

5. Experimental results

This section shows the results of a thorough technical and performance assessment of DARGOS. We compare it with two notable monitoring solutions, namely Nagios and Lattice. Section 5.1 provides relevant technical details about Nagios and Lattice and presents an in-depth qualitative technical comparison of both systems with DARGOS. Section 5.2 presents quantitative experimental results about DARGOS and compares it with Nagios and Lattice as well. It also shows the advantages of using DARGOS to enable prompt scheduling decisions in highly dynamic OpenStack-based IaaS deployments.

[3] A demo of the Web Cloud Monitoring Console is available at http://prague.ugr.es/dargosdemo/.

5.1. Technical comparison between benchmarked solutions

Among the systems considered in Section 2.2, we have selected Nagios [12] and Lattice [10] for comparison with DARGOS. Nagios is selected as a relevant example of a general-purpose data center monitoring system based on plug-ins, because of its widely-spread use due to its simplicity and extensibility. Lattice is a very recent state-of-the-art Cloud monitoring system for multi-tenant and interoperable Cloud environments.

As mentioned before, Nagios is used for checking the host status (e.g., via connectivity and ping delay) and services status (e.g., HTTP), but it also supports resource monitoring through the NRPE extension mechanisms (to enable pull interactions) and the NSCA mechanisms (for push ones).

Lattice is a publish/subscribe framework for monitoring Clouds used in the RESERVOIR project, where monitored entities (DataSources) push monitoring data into data consumers and supports both unicast and multicast communication modes. Its main advantages include its decentralized architecture, the support for multiple consumers via multicast, the data filtering support, and the use of standardized formats for data exchange (i.e., XDR). Being Lattice very similar to the data-centric publish/subscribe approach proposed by DARGOS, it is a good candidate for comparison with DARGOS.

Table 2 shows a side-by-side qualitative technical comparison between the most relevant features of the selected monitoring systems.

Nagios is the only one that uses a centralized architectural model, where one central server is responsible of storing and processing monitoring usage. Furthermore, the centralized nature makes Nagios the only feasible approach in many-to-one scenarios. Lattice supports many-to-many cardinality only in multicast scenarios, where it requires installing multiple agents in absence of multicast. On the contrary, DARGOS supports many-to-many communications in any case, thus making it a monitoring solution suitable for completely distributed and decentralized scenarios.

Regarding the interaction model, Nagios NRPE is the only one that uses a pull approach, but that yields higher latencies for getting the monitoring data because it needs both a request and a reply.

Another important feature is the discovery of sensors and hosts. Lattice and DARGOS support automatic discovery, whereas Nagios relies on configuration files. This limitation restricts the use of Nagios to static scenarios only.

The exchange of data between monitored nodes and consumers is another important characteristic: it should be efficient and – as far as possible – should rely in standardized formats. In that sense, while Nagios uses plain text for serializing data, DARGOS uses Common Data Representation (CDR) and Lattice uses eXternal Data Representation (XDR) [41,42]. Furthermore, DARGOS is the only one that uses a standardized protocol to

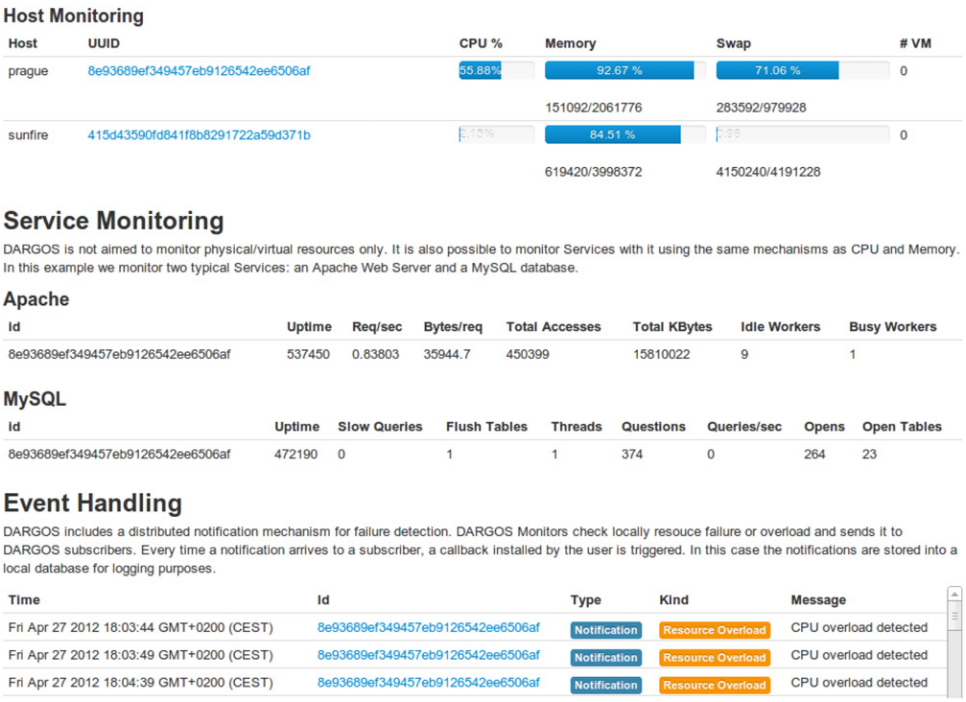

Fig. 8. DARGOS Cloud Web Monitoring Console screenshot.

Table 2
DARGOS, Lattice, and Nagios qualitative comparison.

	DARGOS	Lattice	Nagios
Coupling	Loose	Medium	Tight
QoS support	Yes	No	No
Metadata	Out band	In band	In band
Data exchange format	CDR	XDR	Plain text
Communication model	Push	Push	Push (NSCA) or pull (NRPE)
Discovery method	Automatic	Automatic	Configuration files
Historic data	Yes	No	Yes
Notification strategy	Periodic/event based	Periodic/event based/on change	Depends on plugin implementation
Filtering support	Time and content based. On source and destination	Content based. On origin	Depends on plugin implementation
Network transport	UDP	UDP	TCP
Architectural model	Distributed	Distributed	Centralized
Communication cardinality	N:M	1:N and N:M (only in multicast)	1:N
Transport protocol	RTPS2 standard	Proprietary	Proprietary (NSCA and NRPE)
Asynchronous event notification	Yes	No	Yes

exchange data (RTPS OMG standard) [43], while Nagios and Lattice use proprietary protocols. In this sense, the integration of DARGOS with third party DDS-based applications is easy due to its RTPS protocol compliance. Another difference is that Nagios uses the TCP protocol for exchanging data between monitored nodes and the central server; that increases the latency – due to TCP connection establishment, error, congestion and flow control – compared to the UDP connectionless protocol used by DARGOS and Lattice.

The handling of metadata that identify the attributes of each monitored sensor and their types are also different. Nagios sends metadata inline within each data sample; hence, the server has the responsibility of parsing the attributes and their values properly. Lattice has the ability to choose whether sending the metadata with the sample or not; however, when no metadata are sent an additional mechanism has to be established by the application to help Lattice consumers to fill-in the attributes for each sensor. DARGOS exchanges sensor metadata only during the initial discovery phase, thus it saves bandwidth, especially when the number of sensors and samples are large.

In order to save extra bandwidth and resources, monitoring systems can implement certain filtering and update mechanisms. The main method to save bandwidth is to send measurement updates only upon certain events instead of periodically. While both DARGOS and Lattice support such capability, Nagios does not support that feature that has to be implemented at the application level. Another difference between DARGOS and Lattice is that only the former lets each consumer define its own data filters. In other words, each DARGOS subscriber can define the rate for receiving updates, while in Lattice this rate is globally identified. Consequently, in DARGOS it is easier to deploy heterogeneous applications with different requirements such as multi-tenant Clouds.

The last aspect we want to stress is the ability to send and handle asynchronous event notifications (including also alarms, failures, etc.). This feature can be useful to trigger reconfiguration mechanism on failures, send notifications to the administrator, and detect anomalous situations. In this respect, only DARGOS and Nagios provide native support, while Lattice leaves it at the application level.

5.2. Performance results

To thoroughly assess the DARGOS functionalities and performance, we have carried out four sets of experiments. First, we evaluate DARGOS network usage and compare it with plain OpenStack.

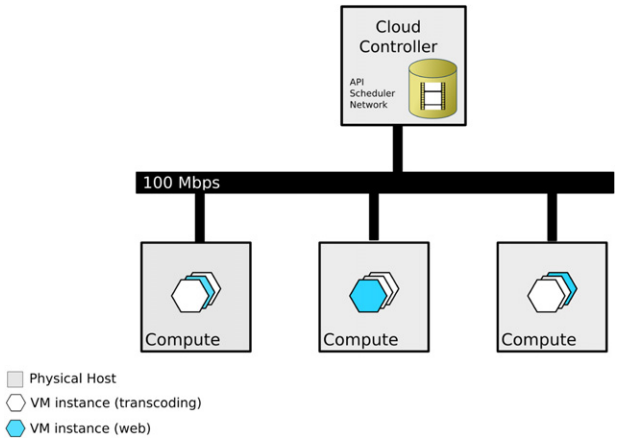

Fig. 9. *Testbed*-1 configuration.

Then, we provide DARGOS, Nagios, and Lattice performance comparisons. Next, we evaluate the scalability of DARGOS and Lattice in a highly challenging real medium-size datacenter. Specifically, we use 51 physical hosts with a very high number of publishers (VM monitoring agents) ranging from 200 to 2400. In all the reported results, to measure the generated traffic we used the Wireshark tool. Finally, in the last test set we provide some seminal experimental results about how DARGOS can be used to enable efficient scheduling in the management of a highly dynamic real Cloud scenario.

(1) *DARGOS network usage impact*

The first set of experiments quantifies the impact of DARGOS in the network usage. We have deployed DARGOS within a real Cloud testbed based on OpenStack. The so called *Testbed*-1 (Fig. 9) consists of four servers interconnected by one 100 Mbps switch. Table 3 details *Testbed*-1 hardware and software characteristics. In particular, it includes OpenStack Nova (diablo) as a Cloud IaaS, OpenStack Glance as VM Image delivery Service [44], KVM hypervisor for VM virtualization [45], and RTI DDS publish/subscribe middleware.

In this experiment, we have measured the network traffic generated for monitoring the Cloud nodes. In particular, we compare DARGOS traffic with the reference OpenStack monitoring function. The baseline OpenStack scheme uses both MySQL to store and query static information (typically users, number of running VMs, etc.) as well as AMQP to exchange messages and RPC requests/responses between all involved OpenStack services.

DARGOS setup consists of 1 NSA at the Controller node and 3 NMAs – with one CPU usage sensor monitor –, each one at OpenStack Compute nodes (Fig. 9). We assessed both monitoring update strategies: event-based (with the following thresholds for CPU usage [0, 25, 50, 60, 70, 75, 80, 85, 90, 95, 100]%) and periodic (with the very challenging and short period of 1 s). Computational load depends on VM instantiation. Each VM imposes a pure computational load chosen randomly between 0% and 60% at VM starting time. Requests follow a Poisson distribution with an average inter-arrival time of 4 s. Each experiment lasted 300 s and was repeated 33 times.

Fig. 10, in a logarithmic scale for the sake of better reading, depicts the obtained results. It shows that (DDS-based) DARGOS performs better than the OpenStack reference implementation for both periodic and even-based notifications. In particular, DARGOS generates significantly less traffic than the aggregated OpenStack traffic (i.e., MySQL plus AMQP). Notably, all the reported results have a very small variance, always below 4%. Let us also stress that to make a fair comparison DARGOS results include also the traffic for automatic discovery of entities; however, discovery traffic is not necessary and accordingly not assessed in the baseline centralized OpenStack reported results.

The second set of experiments shows DARGOS performance when the number of sensors and NSAs increases. In this case, we used *Testbed*-1 with the same settings but with up to 4 sensors for each NMA. Specifically, besides the CPU sensor, each NMA has the OS load sensor (i.e., the average number of processes waiting for CPU), the memory, and hypervisor sensors (Fig. 11(a)). We also show the generated monitoring traffic when the number of supervisors (NSAs) increases (for the 3 NMAs) using both UDP unicast and multicast (Fig. 11(b)). Again, in these experiments the variance is always below 4%.

Fig. 11(a) shows that increasing the number of monitored sensors implies a linear growth in the network traffic. Hence in general terms, NSA should either subscribe only to sensors of interest or alternatively should define specific summaries, such as the default DARGOS *HostSummary* (Fig. 4).

However, if the number of NSAs is increased (Fig. 11(b)), we see that the network traffic can be drastically reduced by using DDS-based multicast communications. In fact, for unicast communications, the network traffic approximately grows linearly as the number of NSAs does, whereas for the multicast case it grants almost a constant traffic trend.

(2) *DARGOS, Nagios, and Lattice generated traffic comparisons*

This second experimental evaluation compares DARGOS, Nagios, and Lattice performance in *Testbed*-1 (Fig. 9) by adopting a configuration similar to the one in previous section. In this case, to ease the comparison we consider only one CPU sensor. The CPU usage was monitored every second for a 300 s period. In the case of DARGOS and Lattice we use the built-in CPU measurement mechanisms, whereas in Nagios we have implemented a Nagios compatible plug-in that measures CPU usage.

The first set of experiments measures the traffic generated by different monitoring systems when the number of publishers (NMAs for DARGOS) increases from 1 to 3 with just 1 subscriber

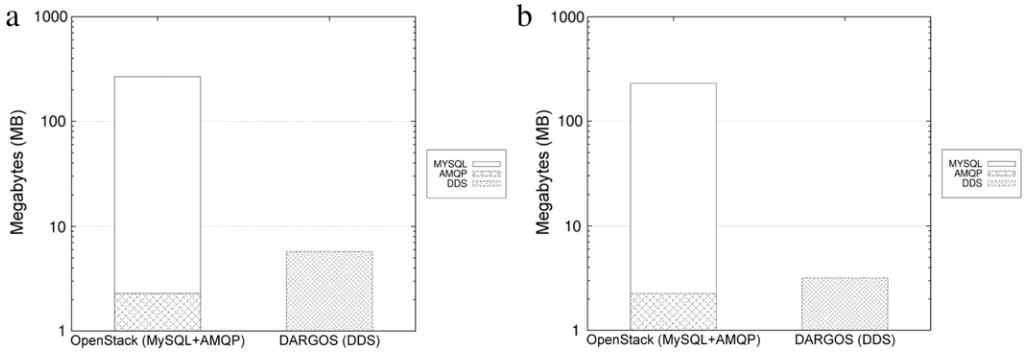

Fig. 10. Generated traffic per protocol for (a) periodic notifications and (b) event-based notifications in *Testbed*-1.

Table 3
Testbed-1 hardware equipment and software.

Host	Installed services
Controller: Intel(R) Xeon(R) CPU E5440 @ 2.83 GHz 8 Gb RAM	Nova (API, Nova scheduler, Nova network) Glance, NFS server, MySQL server and RabbitMQ, RTIDDS 4.5d
*Compute*01: Intel Core i5 CPU 750@2.67 GHz; 4 Gb RAM; 250 GB HD	Nova compute, KVM, RTI DDS 4.5d
*Compute*02: Intel Core i5 CPU 750@2.67 GHz; 4 Gb RAM; 250 GB HD	Nova compute, KVM, RTI DDS 4.5d
*Compute*03: Intel Core i5 CPU 750@2.67 GHz; 4 Gb RAM; 250 GB HD	Nova compute, KVM, RTI DDS 4.5d

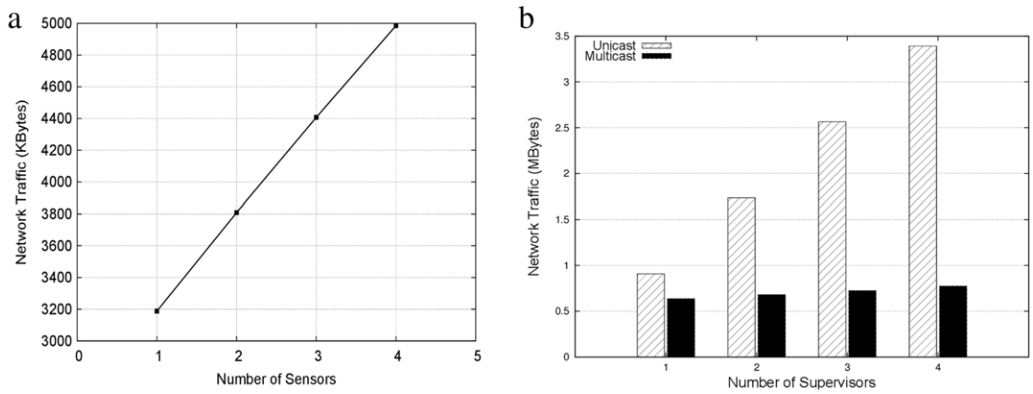

Fig. 11. DARGOS generated network traffic vs. (a) the number of sensors and (b) the number of NSAs in *Testbed*-1.

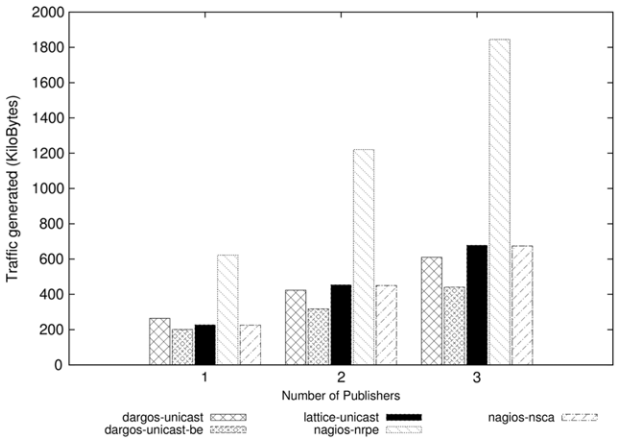

Fig. 12. Traffic generated by DARGOS, Lattice and Lattice in *Testbed*-1.

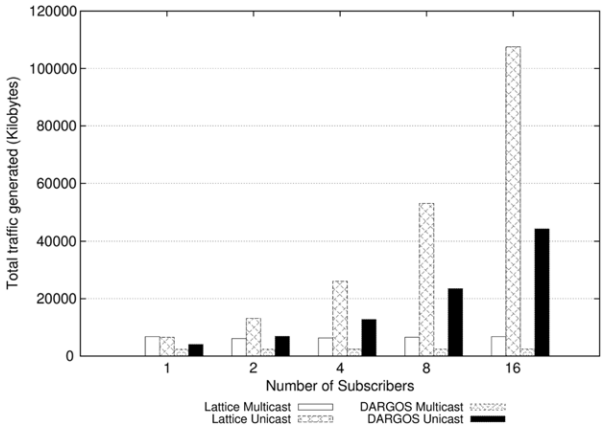

Fig. 13. DARGOS and Lattice total generated traffic (30 publishers) vs. the number of subscribers in *Testbed*-1.

(NSA for DARGOS). Fig. 12 shows the total traffic generated by each system. In particular, it includes the following schemes: *dargos-unicast* adopts the UDP-based DDS reliable unicast protocol; *dargos-unicast-be* is the scheme based on UDP with best effort and not reliable DDS unicast; *nagios-nrps* and *nagios-nsca* refer respectively to Nagios pull and push interaction models. In these two cases, both schemes use TCP; finally, *lattice-unicast* uses best effort unicast UDP transport protocol.

On the one hand, *nagios-nrps* and *nagios-nsca* by relying on TCP require establishing and maintaining TCP connections with each monitored node: that significantly affects the scalability and latency of the Nagios solution. On the other hand, DARGOS and Lattice exploit the lightweight UDP-based communications for monitoring data without establishing long-term connections with monitored nodes.

Results show that, for just one publisher, DARGOS generated traffic is higher than Lattice. This effect can be explained with the overhead caused by the metadata exchanged by DDS to enable dynamic peer discovery. In Lattice, instead, metadata are included in each data sample. Therefore, when the overall number of data samples increases, the impact on the overall traffic increases as well.

Remarkably, Lattice traffic is higher than DARGOS for 3 publishers (Fig. 12). Additionally, focusing on just the best effort evaluated schemes (labels *dargos-unicast-be* and *lattice-unicast*), we also see that DARGOS always outperforms Lattice.

The second set of experiments examines the impact of the number of subscribers on the overall traffic (Fig. 13). In this case, we have increased the number of subscribers from 1 to 16 while maintaining the number of publishers equal to 30. This scenario considers DARGOS and Lattice only because the comparison with Nagios would be unfair due to its centralized architecture does not fit the multiple-subscribers scenario. Lattice, instead, supports both unicast and multicast communication modes.

The results show that – as expected – for both Lattice and DARGOS unicast traffic increases linearly. However, multicast communication significantly reduces the overall generated traffic for both schemes. Notably, it remains almost constant as the number of subscribers increases. These results also confirm that DARGOS always (unicast and multicast) outperforms Lattice, partially due to the heavier Lattice serialization scheme, and due to the Lattice inline metadata transfer as well.

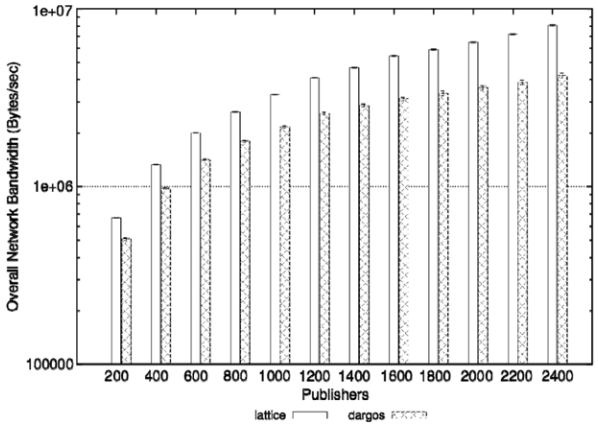

Fig. 14. Scalability results for DARGOS and Lattice (log scale) in *Testbed*-2.

(3) *DARGOS and Lattice scalability*

Having established that Lattice and DARGOS multicast are the best candidates for high scalability, the third part of our evaluation focuses on these two solutions and compares them in a real deployment when the number of publishers, and consequently the total data rate, significantly increases. In fact, real medium size data centers are usually composed from some tens to hundreds of computers, and thousands of VMs, so it is of paramount importance to test the behavior of our monitoring architecture when the number of entities is large.

With this goal we deployed Lattice and DARGOS in a different scenario (labeled to as *Testbed*-2) composed of 51 dual-core computers connected by a 100 Mbps switch with multicast support. *Testbed*-2 emulates a medium size data center with each host running one subscriber and multiple publishers.

In particular, to thoroughly stress the system scalability each publisher publishes 5 updates per second, more than the usual update rate for a data center. In addition, for this evaluation we increase the number of publishers from 200 to 2400. We run the same experiment with both DARGOS and Lattice in the same conditions (data rate and number of publishers): both systems were executed for 2 min and repeated it 33 times in order to measure the average global throughput generated in every host.

Fig. 14 reports collected results (average and the corresponding standard deviation) in a logarithmic scale for sake of better readability. Results are reported up to 2400 publishers because at this load Lattice traffic reaches network saturation (i.e. the maximum throughput obtainable in our 100 Mbps Ethernet).

Results show that DARGOS always outperforms Lattice. The differences are caused by the lower size of DARGOS messages; in fact, when the number of sensor samples increases, the throughput also increases. Obtained results also confirm that DARGOS architecture is more scalable than Lattice. Especially, for a high number of publishers (more than 1800) when the overall data rate ranges from $5 * 1800 = 9000$ to $5 * 2400 = 12\,000$ updates-per-second.

Let us close this part with some considerations about the employed deployment and the applicability of DARGOS in even larger data centers and deployment scenarios. Industrial scale Cloud data centers are usually deployed using hierarchical network topologies such as trees, fat trees, and VL2 [30–32,46]; as a main consequence, monitoring data can traverse multiple switch/routers. In our experiments, we have specifically addressed the scalability at the lower level of the data center (between leaf hosts within the same network topology branch) because, as introduced in Section 3 when we defined DARGOS zones, it is the main deployment scenario for DARGOS. At this level, DARGOS groups the hosts in the same monitoring zone and exploits direct and multicast-enabled communication between NMA and NSA agents. At the same time, DARGOS supports also more complex network topologies. On the one hand, it is possible to deploy NSA agents at the highest levels of the data center network topology acting as gateways to publish monitoring data toward the Internet through our DARGOS REST/JSON APIs; on the other hand, NSA could be deployed within other distant zones (different from the NMA one) so to collect, aggregate, and export monitoring data of interest out of zone borders. In either cases, system administrators should carefully configure and tailor NSA deployment so to satisfy different monitoring requirements and comply with monitoring quality levels, such as maximum delay constraints and bandwidth consumption, especially when monitoring data traverses multiple routing/switch devices. To ease that management and configuration task, system administrators could employ our advanced and lightweight DDS bridging solutions specifically designed to tackle such complex deployments, for instance to minimize bandwidth usage due to data duplication when multiple NSAs, all deployed in another network trunk, are interested in the same data [47].

(4) *DARGOS in a real Cloud use case*

The last set of experiments show the feasibility of DARGOS in the management of a real Cloud deployment with highly variable conditions. In this experiment we considered two tenants that require two different kinds of tasks that heavily stress the architecture. The first task is a multimedia Cloud transcoding application, deployed in one VM image. This application downscales and transcodes a High Definition (HD) video previously stored in the Cloud: those transcoding applications are likely to meet the requirements (i.e., frame dimension, CPU, and bandwidth) of resource constrained mobile devices. The second task is a VM with a classic Linux + Apache + MySql + PHP (LAMP) software stack with a Content Management Service (CMS) running on it. This arrangement corresponds to a typical Web deployment.

The two considered tasks can also affect the dynamicity of a Cloud: multimedia transcoding are typically CPU-intensive short lived tasks, whereas Web provisioning are network intensive long-running tasks.

For this experiment we consider again the *Testbed*-1 scenario shown in Fig. 9. We developed a custom Cloud scheduler for OpenStack in which DARGOS NSA gathers monitoring data that are used to take scheduling decisions. Each Compute node is responsible of VM lifecycle and publishes via DARGOS NMA its current resources usage. For each new VM activation request, the scheduler queries its local NSA cache to determine the current resource usage of servers on the pool. When the request arrives, the scheduler checks the usage status of all available nodes and it chooses the node where the VM will be launched depending on implemented scheduling strategy.

In this case, we generate 30 requests with a Poisson distribution and an average inter-arrival time of 4 s. The NMA is equipped with two sensors that measure the number of VMs running per hosts and the CPU usage of the physical node. VM requests are for the two above types of tasks, either Web server or transcoding service. The former is executed until the end of the experiment, while the latter has a finite lifetime that depends on the duration of the video (between 20 and 50 s) and the encoding parameters.

Fig. 15 shows the experimental results (number of VMs instantiated and CPU usage per host) obtained for two different scheduling strategies: server consolidation (Fig. 15(b)) and load balancing (Fig. 15(c)). The first strategy tries to use as fewer servers as possible in order to save power consumption, whereas the second one implements a simple load balancing function that instantiates new VMs in the server less loaded according to their resource usage measurements.

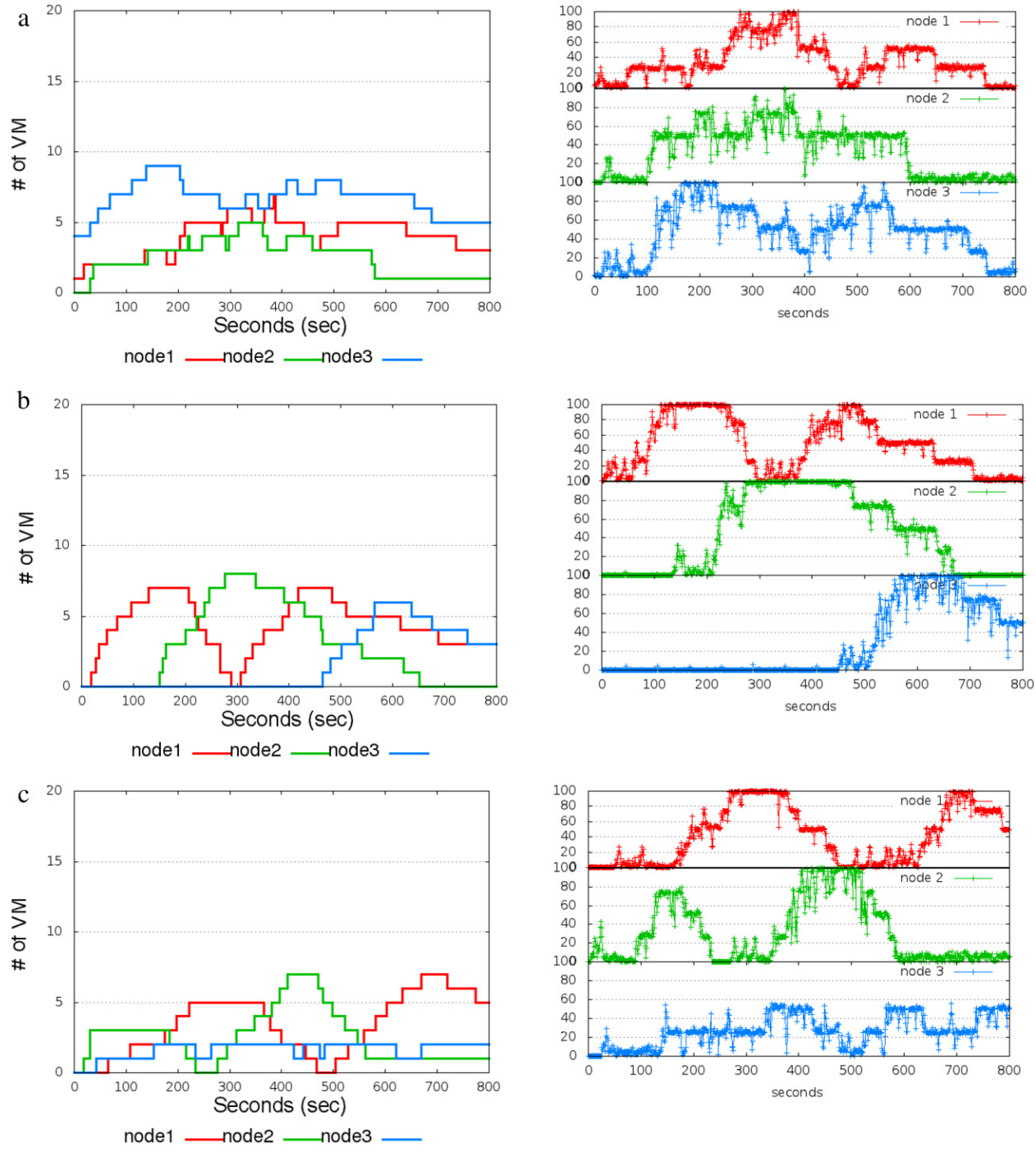

Fig. 15. Number of VMs instantiated and CPU usage per host with (a) no monitoring information, (b) server consolidation, and (c) load balancing strategies in *Testbed*-1.

Previously, Fig. 15(a) compares our strategies with the default scheduling policy provided by the default OpenStack Nova which places VMs by selecting one host randomly between the available ones [44]. The default strategy (Fig. 15(a)) loads all nodes almost equally, thus it uselessly wastes precious energy resources. By using our server consolidation strategy, instead, VMs machines are not instantiated in a second and a third node until, respectively, the first and the second nodes are close to congestion, based on the measurements given by the CPU sensors (CPU usage around 80%). Let us also note that the third node is not used until the first and second servers are loaded enough. Finally, our load balancing scheduling strategy aims to equally load all nodes since the beginning of the execution of the experiment, by delivering incoming VM requests to the least loaded one.

Obtained results show that DARGOS allows Cloud administrators to easily implement different and accurate resource provisioning strategies by using the monitoring data acquired by DARGOS. The collected data can be usefully employed not only to monitor the Cloud system but also to trigger prompt control actions, flexibly modifiable according to specific tenant requirements: for instance, to personalize monitoring actions and to use different scheduling strategies for different data center zones and according to specific SaaS/PaaS/IaaS service requirements.

6. Conclusion

This paper proposes a decentralized architecture for monitoring Cloud environments. It is designed to satisfy the requirements of a Cloud environment such as reliability, efficiency, multi-tenancy, and high scalability while introducing a low overhead. The proposed architecture is based on the publish/subscribe paradigm and allows choosing the zones to monitor and other communication features, such as the update rates and the set of sensors. In addition, we have implemented a prototype that has been integrated into a real Cloud architecture for a real use case evaluation. Due to deployment constrains, we performed scalability evaluations only at the cluster level within the same rack, but the flexibility of the DARGOS architecture and the use of specific DDS bridges and proxies allow extending our results for industrial scale scenarios [46].

The reported experimental results show that our approach can monitor Cloud environments without incurring in a significant overhead, and compares favorably to other notable monitoring systems. Encouraged by these results, we are considering several future directions: on the one hand, we want to explore further optimizations techniques, such as message batching, to improve the scalability especially in huge scenarios where multicast is not available; on the other hand, we are developing new sensors for new services and networking-related metrics, such as the access control services proposed in [48,49]. Finally, we are convinced that DARGOS could be suitably employed for the monitoring of large-scale federated and multi-tenant Clouds for new Smart City scenarios.

Acknowledgments

This work was partially supported by the "Ministerio de Ciencia e Innovación"' of Spain under research project TIN2010-20323, and by the "Centro Inter-Dipartimentale per la Ricerca Industriale" (CIRI ICT) of the University of Bologna, credited by the Emilia-Romagna Region, Italy.

References

[1] Gartner, Gartner says energy-related costs account for approximately 12% of overall data center expenditures [Online]. http://www.gartner.com/it/page.jsp?id=1442113.
[2] Raúl Alonso-Calvo, Jose Crespo, Miguel García-Remesal, Alberto Anguita, Victor Maojo, On distributing load in cloud computing: a real application for very-large image datasets, Procedia Computer Science (2010) 2069–2677.
[3] Salekul Islam, Jean-Charles Grégoire, Giving users an edge: a flexible Cloud model and its application for multimedia, Future Generations Computer Systems (2012) http://dx.doi.org/10.1016/j.future.2012.01.002.
[4] K. Kumar, Y.H. Lu, Cloud computing for mobile users: can offloading computation save energy? Computer 43 (4) (2010) 51–56.
[5] Object Management Group, Data distribution service specification 1.2 [Online]. http://www.omg.org/spec/DDS/1.2/.
[6] E. Galstad, Nagios service check acceptor, NSCA, 2011 [Online]. http://nagios.sourceforge.net/download/contrib/documentation/misc/NSCA_Setup.pdf.
[7] Hyperic HQ., VmWare Inc., 2012 [Online]. http://www.hyperic.com/.
[8] Ethan Galstad, Nagios NRPE Documentation, May 2007 [Online]. http://nagios.sourceforge.net/docs/nrpe/NRPE.pdf.
[9] The Ganglia Project, Ganglia monitoring service, 2012 [Online]. http://www.ganglia.info.
[10] Stuart Clayman, et al., Monitoring service clouds in the future internet, in: Towards the Future Internet, 2010.
[11] He Huang, Liqiang Wang, P&P: a combined push–pull model for resource monitoring in cloud computing environment, in: 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD, 2010.
[12] Nagios Enterprises LLC, Nagios—the industry standard in IT infrastructure monitoring, 2012 [Online]. http://www.nagios.org.
[13] I. Legrand, et al., MonALISA: an agent based, dynamic service system to monitor, control and optimize distributed systems, Computer Physics Communications 180 (12) (2009) 2472–2498.
[14] Shirlei Aparecida de Chaves, Rafael Brundo Uriarte, Carlos Becker Westphall, Toward an architecture for monitoring private clouds, IEEE Communications Magazine 49 (12) (2011) 130–137.
[15] Peer Hasselmeyer, Nico d'Hereuse, Towards holistic multi-tenant monitoring for virtual data centers, in: Network Operations and Management Symposium Workshops, NOMS Wksps, 2010 IEEE/IFIP, Osaka, 2010, pp. 350–356.
[16] B. König, J.M. Alcaraz Calero, J. Kirschnick, Elastic monitoring framework for cloud infrastructures, IET Communications 6 (10) (2012) 1306–1315.
[17] Wu-Chun Chung, Ruay-Shiung Chang, A new mechanism for resource monitoring in grid computing, Future Generation Computer Systems 25 (1) (2009) 1–7.
[18] S. Reyes, C. Muñoz-Caro, A. Niño, R. Sirvent, R.M. Badia, Monitoring and steering grid applications with GRID superscalar, Future Generation Computer Systems 26 (4) (2010) 645–653.
[19] B. Balis, B. Kowalewski, M. Bubak, Real-time grid monitoring based on complex event processing, Future Generation Computer Systems 27 (8) (2011) 1103–1112.
[20] E. Torres, D. Segrelles, I. Blanquer, V. Hernandez, Service monitoring and differentiation techniques for resource allocation in the grid, on the basis of the level of service, Future Generation Computer Systems 27 (8) (2011) 1142–1152.
[21] O. Biran, et al. A stable network-aware VM placement for cloud systems, in: Proceedings of the IEEE International Symposium on Cluster, Cloud and Grid Computing, CCGrid'12, Ottawa, Ca, 2012, pp. 498–506.
[22] Luis Rodero-Merino, et al., From infrastructure delivery to service management in clouds, Future Generation Computer Systems 26 (8) (2010) 1226–1240.
[23] Ana J. Ferrer, et al., OPTIMIS: a holistic approach to cloud service provisioning, Future Generation Computer Systems 28 (1) (2012) 66–77.
[24] Antonio Corradi, Mario Fanelli, Luca Foschini, VM consolidation: a real case based on openstack cloud, Future Generation Computer Systems (2012).
[25] Augusto Ciuffoletti, Monitoring a virtual network infrastructure: an IaaS perspective, SIGCOMM Computer Communication Review 40 (5) (2010) 47–52.
[26] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, Alla Segal, Dynamic scaling of web applications in a virtualized cloud computing environment, in: Proceedings of the 2009 IEEE International Conference on e-Business Engineering, 2009, pp. 281–286.
[27] B. Rochwerger, et al., The Reservoir model and architecture for open federated cloud computing, IBM Journal of Research and Development 53 (4) (2009) 1–11.
[28] S. Clayman, A. Galis, L. Mamatas, Monitoring virtual networks with Lattice, in: Network Operations and Management Symposium Workshops, NOMS Wksps, 2010 IEEE/IFIP, 2010.
[29] S. Clayman, R. Clegg, L. Mamatas, G. Pavlou, A. Galis, Monitoring, aggregation and filtering for efficient management of virtual networks, in: 2011 7th International Conference on Network and Service Management, CNSM, 2011.
[30] CISCO, Cisco Data Center Infrastructure 2.5 Design Guide, 2011.
[31] A. Loukissas, M. Al-Fares, A. Vahdat, A scalable commodity data center network architecture, in: Proceedings of ACM SIGCOMM 2008 Conference on Data Communication, 2008, pp. 63–74.
[32] A. Greenberg, J.R. Hamilton, N. Jain, et al. VL2: a scalable and flexible data center network, in: Proceedings of ACM SIGCOMM 209 Conference on Data Communication, 2009, pp. 51–62.
[33] P. Leach, M. Mealling, R. Salz, RFC 4122: a universally unique identifier, UUID, URN Namespace, 2005.
[34] Rackspace, OpenStack [Online]. http://www.openstack.org.
[35] Amazon, Amazon Elastic Compute Cloud, Amazon EC2, 2012 [Online]. http://aws.amazon.com/ec2.
[36] Íñigo Goiri, et al., Resource-level QoS metric for CPU-based guarantees in cloud providers, in: Economics of Grids, Clouds, Systems, and Services, Vol. 6296, 2012, pp. 34–47.
[37] VMware Inc., VMware vSphere private cloud computing and virtualization, October 2012 [Online]. http://www.vmware.com/uk/products/datacenter-virtualization/vsphere/overview.html.
[38] Yongzhen Zhuangt, Lei Chen, X.S. Wang, Jie Lian, A weighted moving average-based approach for cleaning sensor data, in: 27th International Conference on Distributed Computing Systems, 2007, ICDCS'07, 2007, pp. 25–27.
[39] D. Crockford, RFC 4627: the application/json media type for javascript object notation, JSON, IETF, Request for Comments 2006.
[40] Yahoo! Inc., Yahoo! Pipes, April 2012 [Online]. http://pipes.yahoo.com/.
[41] M. Eisler, RFC 4506: XDR: external data representation standard, 2006.
[42] Object Management Group, CORBA 3.0—General Inter-ORB Protocol Chapter [Online]. http://www.omg.org/cgi-bin/doc?formal/02-06-51.
[43] Object Management Group, The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol specification, v2.1, 2010 [Online]. http://www.omg.org/spec/DDS-RTPS/2.1/.
[44] Rackspace, Openstack Nova, 2012 [Online]. http://nova.openstack.org.
[45] KVM Project, Kernel based virtual machine, KVM, April 2012 [Online]. http://www.linux-kvm.org/.
[46] C.E. Leiserson, Fat-trees: universal networks for hardware-efficient supercomputing, IEEE Transactions on Computers (1985) 892–991.

[47] J.M. Lopez-Vega, J. Povedano-Molina, G. Pardo-Castellote, J.M. Lopez-Soler, A content-aware bridging service for publish/subscribe environments, Journal of Systems and Software (2012).
[48] Jorge Bernal Bernabe, J.M. Marin Perez, J.M. Alcaraz Calero, F.J. Martinez Perez, A.F. Gomez Skarmeta, Semantic-aware multi-tenancy authorization system for cloud architectures, Future Generation Computer Systems (2012).
[49] F. Bracci, A. Corradi, L. Foschini, Database security management for healthcare SaaS in the Amazon AWS Cloud, in: Proceedings of the IEEE International Workshop on Management of Cloud Systems, MoCS'12, Cappadocia, Turkey, 2012, pp. 812–819.

Javier Povedano-Molina is a Ph.D. student at the Department of Signal Theory, Telematics and Communications of the University of Granada (Spain). He holds an M.Sc. degree in Computer Science from the University of Granada and a B.S. in Computer Science from the University of Córdoba (Spain). His research interests include multimedia networking, peer-to-peer architectures, distributed systems, and Cloud Computing.

Jose M. Lopez-Vega is a Ph.D. student at the Department of Signal Theory, Telematics and Communications of the University of Granada (Spain). He received his B.S. degree in Telecommunications from the University of Granada, and his M.S. in Multimedia Systems also from the University of Granada. He was granted a Ph.D. fellowship by the Education Spanish Ministry on July 2009 and started his Ph.D. studies. His research interests include QoS, peer-to-peer networks, mobile networks, and network middleware.

Juan M. Lopez-Soler is Associate Professor at the Department of Signals, Telematics and Communications of the University of Granada (Spain). He teaches Computer Networks, Data Transmission, and Multimedia Networking courses. He holds a B.Sc. and a Ph.D. in Physics from the University of Granada. In 1991–92, he joined the ISR at the University of Maryland (USA) as Visiting Faculty Research Assistant.

He has participated in 10 public and 13 private research projects, 9 as coordinator.

He has published 13 papers in indexed journals and more than 25 in international workshops/conferences.

His research interests include real-time middleware and multimedia networking.

Antonio Corradi is a full professor of Computer Engineering at the University of Bologna. He graduated from University of Bologna, Italy, and received M.S. in Electrical Engineering from Cornell University, USA. His research interests include distributed and parallel systems and solutions, middleware for pervasive and heterogeneous computing, infrastructure support for context-aware multimodal services, network management, and mobile agent platforms. He is a member of IEEE, ACM, and Italian Association for Computing (AICA).

Luca Foschini is a Research Fellow at the University of Bologna, Italy. In 2007, he received a Ph.D. degree in Computer Science Engineering from University of Bologna. His research interests include distributed systems and solutions for pervasive computing environments, system and service management, context-aware services and adaptive multimedia, and mobile agent-based middleware solutions. He is a member of IEEE and ACM.