Monitoring Elastically Adaptive Multi-Cloud Services

Demetris Trihinas, George Pallis, Marios D. Dikaiakos

Abstract—Automatic resource provisioning is a challenging and complex task. It requires for applications, services and underlying platforms to be continuously monitored at multiple levels and time intervals. The complex nature of this task lays in the ability of the monitoring system to automatically detect runtime configurations in a cloud service due to elasticity action enforcement. Moreover, with the adoption of open cloud standards and library stacks, cloud consumers are now able to migrate their applications or even distribute them across multiple cloud domains. However, current cloud monitoring tools are either bounded to specific cloud platforms or limit their portability to provide elasticity support. In this article, we describe the challenges when monitoring elastically adaptive multi-cloud services. We then introduce a novel automated, modular, multi-layer and portable cloud monitoring framework. Experiments on multiple clouds and real-life applications show that our framework is capable of automatically adapting when elasticity actions are enforced to either the cloud service or to the monitoring topology. Furthermore, it is recoverable from faults introduced in the monitoring configuration with proven scalability and low runtime footprint. Most importantly, our framework is able to reduce network traffic by 41%, and consequently the monitoring cost, which is both billable and noticeable in large-scale multi-cloud services.

Index Terms—Cloud Computing, Elasticity, Resource Provisioning, Cloud Monitoring, Application Monitoring

1 Introduction

Cloud computing is dominating the interests of organizations across multiple business domains by providing on-demand virtualized infrastructure in a *pay-as-you-use* model [11]. Cloud adoption is driven by elasticity [27], that is the ability of the cloud to adapt to workload changes by automatically (de-)provisioning resources so that the allocated resources always match the current demand [20]. A common approach followed by elasticity controllers is to employ a control loop, often referred to as a MAPE-K loop [21], to manage resource allocation for deployed cloud services. The first step in the loop is to gather monitoring information regarding cloud-service performance and then, analyse it and decide if an elasticity action should be enforced. However, automatic resource provisioning [6] is challenging due to the fact that monitoring elastic cloud services is not trivial [38] and is still considered an open research problem [1].

Monitoring is essential for capturing the performance and understanding the behavior of deployed cloud services and the underlying infrastructure. Organizations acquire monitoring facilities to: (i) make decisions regarding resource allocation and tune their applications accordingly [13] [39]; (ii) detect and prevent security breaches or network problems [31]; and (iii) verify service level agreements (SLAs) [15]. To support these, a monitoring system must be able to monitor heterogeneous types of information at different time granularities, ranging from low-level system metrics (i.e., CPU usage, network traffic) to high-level application-specific metrics (i.e., throughput, latency). Furthermore, the recipients of monitoring metrics may vary or overlap in a multi-tenant environment [31]. For instance, a particular metric (e.g., network utilization) can be accessed -frequently and simultaneously- by many entities (e.g., cloud service stakeholder, resource provisioner, cloud provider) but interpreted differently.

• *Demetris Trihinas, George Pallis and Marios D. Dikaiakos are with the Department of Computer Science, University of Cyprus. e-mail: { trihinas, gpallis, mdd }@cs.ucy.ac.cy*

Manuscript received Apr 07, 2015; revised Oct 05, 2015.

With the wide adoption of open cloud standards [12] [38] and library stacks [24] the interest of cloud consumers to migrate their applications from one cloud to another (i.e., due to better pricing or availability) has significantly increased. Additionally, cloud consumers are now combining resource offerings from various vendors and distributing their cloud services across multiple cloud domains. For instance, hybrid clouds are gaining consumer interests with security concerns [11] [27], as data are kept on premises in private clouds while compute needs are outsourced to public cloud offerings. Cloud providers typically provide advanced proprietary monitoring facilities, which are made available to customers [3] [33] as Monitoring-as-a-Service offerings. Alternatively, general-purpose monitoring tools [9] [14] [19] can serve cloud monitoring needs through highly specialized configurations or extensions. However, current cloud monitoring systems are bounded to operating on specific cloud platforms. Also, cloud consumers are obliged to use and configure multiple monitoring systems if interested in distributing their applications across multiple cloud providers; this, makes application migration even more challenging. Therefore, it is desirable for cloud service stakeholders to seek for a monitoring system that is both platform-independent and interoperable allowing it to operate seamlessly across multiple cloud domains.

Despite the inherent ability of cloud platforms to provide elasticity and resources on-demand [39], monitoring systems were neither designed to facilitate the needs of elastic cloud services nor do they provide adequate elasticity support. Instead, re-contextualization [7] is required when the service topology adapts due to the enforcement of elasticity actions (e.g. new VM allocated to cloud service) or when obtained resources are re-configured (e.g. new disk attached to a VM). To avoid re-contextualization, current monitoring tools resort to information acquired either from special components deployed on the physical infrastructure [22], the underlying hypervisor [9] or via the cloud provider through a directory service [14]. These approaches may perform acceptably well for small and slowly changing topologies but they cannot be

considered in scenarios with large-scale and highly adaptive multi-cloud services where rapid elasticity is the case and not the exception.

In this article, we address the above challenges by focusing on the issues that arise when monitoring elastically adaptive multi-cloud services. We introduce **JCatascopia**, an open-source[1], fully-automated, multi-layer and platform-independent cloud monitoring framework. JCatascopia runs in a non-intrusive and transparent manner to any underlying cloud as neither the metric collection process nor metric distribution and storage are dependent on the underlying platform. JCatascopia uses a novel variation of the publish and subscribe communication protocol to dynamically detect, without any human intervention or dependence to the hypervisor, when monitoring instances have been (de-)provisioned due to elasticity actions. This diminishes the need for re-contextualization when providing elasticity support, by reflecting at all time the current topology and resource configuration. In addition, JCatascopia provides metric filtering to reduce the communication overhead for metric distribution and storage, and generates high-level application metrics dynamically by aggregating and grouping low-level metrics.

This article substantially extends our previous work [37], as follows: (i) We extend the JCatascopia communication protocol, to accommodate runtime VM re-configuration for horizontal and vertical elastic scaling (e.g. attach a new IP to VM) and we add the functionality to overcome network connectivity problems; (ii) JCatascopia has been re-designed as a modular system and is now capable of being deployed in different elastic monitoring topologies to better suit user and service provider needs; (iii) JCatascopia, as an elastic and self-managed system itself, is now able to automatically scale at runtime by an elasticity controller; (iv) We extend the JCatascopia communication protocol to consider failures, describing how the monitoring topology can automatically recover from faults; also, we showcase how we *monitor* the monitoring system; (v) Finally, we present an extensive comparison of our framework to other monitoring tools based on complex real-life testbeds, deployed on four public and private cloud platforms. Results show that the JCatascopia framework is capable of supporting automated cloud resource provisioning systems with proven interoperability, scalability, fault-tolerance and with a small runtime footprint. Most importantly, our framework is able to reduce network traffic by 41%, and consequently the monitoring cost, which is noticeable, billable and increases fast in large-scale elastic and distributed multi-cloud deployments.

The rest of this article is structured as follows: Section 2 presents a study of the related work. Section 3 presents the design, architecture and novelties incorporated to JCatascopia. Section 4 presents an evaluation of our system, while Section 5 concludes this article and outlines the future work.

2 State-of-the-Art & Related Work

Cloud specific monitoring tools such as Amazon CloudWatch [3], Paraleap AzureWatch [30] and RackSpace CloudKick [33] provide Monitoring-as-a-Service to cloud consumers. Despite the fact that these tools are easy to use and well-integrated with the underlying platform, their biggest disadvantage is that they are commercial and proprietary which limits their operation to specific cloud providers. Thus, these tools lack in terms of *portability* and *interoperability*. To address portability, Rak et al. [34] introduce the mOSAIC cloud monitoring system which collects metrics in a cloud-independent manner via the mOSAIC API. However, this system is limited to specific cloud platforms supported by the EU-funded mOSAIC project and is a centralized monitoring approach intended only for small-scale deployments.

General purpose monitoring tools such as Ganglia [19], Nagios [26], Zabbix [43] and GridICE [5] are traditionally used by system administrators to monitor fixed or slowly changing distributed infrastructures, such as computing grids and clusters. Cloud providers tend to adopt such solutions to monitor their platforms as well. However, cloud platforms have different requirements than computing grids [15] [18], as they consist of multiple layers and service paradigms (IaaS, PaaS, SaaS) providing users with *on-demand* resources through an *infinite* pool of virtual resources. This makes the aforementioned monitoring tools unsuitable for *rapidly elastic* and *dynamic* cloud deployments, where VMs are deployed for several minutes on a number of physical nodes and after a short interval migrate to other nodes or are terminated.

To address the limitations mentioned above, several approaches have been proposed. For example, sFlow integrates with Ganglia to monitor VM clusters. Xiang et al. [42] introduce VMDriver, which provides an interface to access metrics in an OS-independent manner but requires from the hypervisor to install a monitoring driver on each guest VM. Montes et al. [25] propose GMonE, a general-purpose monitoring tool applicable to all cloud layers. GMonE allows monitoring instances to be deployed at any level of the cloud and provides a pluggable interface where users can inject their own custom metrics to monitoring agents. However, GMonE cannot detect at runtime service topology changes due to elasticity action enforcement or resource configurations. Quoc et al. [32] propose DoLen, a multi-cloud monitoring tool for distributed cloud services but as a centralized approach it is suitable only for small-scale deployments. Calero et al. [2] introduce MonPaaS, a distributed and agent-less cloud monitoring solution, where a dedicated monitoring server is allocated per application. MonPaaS is not platform independent, as it is tightly coupled to Openstack. In addition, while scalability is claimed, with the use of only one monitoring server per application, intra-service monitoring is bounded by the monitoring intensity and number of running instances.

In regards to elasticity, a handful of academic approaches attempt to propose solutions for elasticity support but require for special entities at the physical level or depend on the underlying hypervisor to detect topology configurations. Thus, these approaches limit their *portability* at different levels of the cloud in favor of *elasticity* support. Specifically, Carvalho et al. [14] propose the use of passive checks by each physical host to notify the central monitoring server about the virtual instances that are currently instantiated. Katsaros et al. [22] extend Nagios through the implementation of *NEB2REST*, a REST event broker utilized to provide elasticity capabilities through an abstraction layer between monitoring agents and the management layer. Clayman et al. [9] introduce *Lattice*, an interesting cloud monitoring framework, which monitors not only physical hosts but also virtual instances. *Lattice* can

1. https://github.com/CELAR/cloud-ms

be utilized to monitor elastically adaptive environments. In particular, the process of determining the existence of new VMs is performed at the hypervisor level. A *controller* is the responsible entity for retrieving a list of running VMs from the hypervisor, detecting if new VMs have been added or removed. Thus, in contrast to our solution, while Lattice offers elasticity support it moves the dependency to the hypervisor layer as it is tightly coupled to Xen hypervisor. In turn, Lattice cannot monitor applications distributed across multiple cloud providers due to its limited multicast network communication model. Moreover, Lattice features an excessive runtime footprint in contrast to our solution.

Another approach is Panoptes [40], which utilizes a pub/sub communication model between agents and servers to enhance private cloud monitoring performance. In contrast to our solution, Panoptes requires a broker (similar to a directory service), which acts as a central contact point for newly instantiated monitoring agents to: (i) contact and request a list of available monitoring servers; (ii) notify all monitoring servers of their existence, and (iii) wait for monitoring servers to start the subscription process, which is a significant overhead for rapidly elastic environments (see Section 3.3.1). Finally, Varanus [41] is an interesting monitoring tool which leverages a multi-tier P2P architecture to achieve *in situ* monitoring of the monitoring infrastructure based solely on resource utilization. However, as we show later, resource utilization is not always the monitoring bottleneck.

Based on the above, we believe that none of the existing solutions can be considered as a complete approach to provide elastic multi-cloud monitoring alongside an elasticity controller. In addition, although monitoring autonomicity is initially studied, it is still far from being considered as achieved. Therefore, to address the above limitations we have designed the JCatascopia multi-cloud monitoring framework.

3 JCatascopia: Design and Features

In this section, we introduce the design, architecture and key features of JCatascopia.

3.1 Platform Independence and Interoperability

JCatascopia is an open-source and elastic monitoring framework designed to provide cloud-independence and interoperability of the monitoring process. Specifically, JCatascopia can be utilized to monitor: (i) federated cloud environments where cloud services are distributed across multiple clouds [32]; and (ii) cloud bursting environments where cloud services deployed on a private cloud, burst to a public cloud when resource demand increases [35].

JCatascopia achieves platform-independence by satisfying the following requirements. First, its components (i.e. monitoring agents, servers, etc.) are portable, meaning they are capable of running on any physical machine or VM instances. To this end, JCatascopia components are developed in Java with none of their functionality being dependent on OS or machine libraries, with the only requirement being a Java installation (v1.6+). In Sections 3.2.x all JCatascopia components and their features are described in detail. Each component presents clear Java interfaces and abstractions for their features and endpoints. This allows features to be customizable and extensible, while the clear endpoints (e.g. Metric Interface) allow integration between JCatascopia and

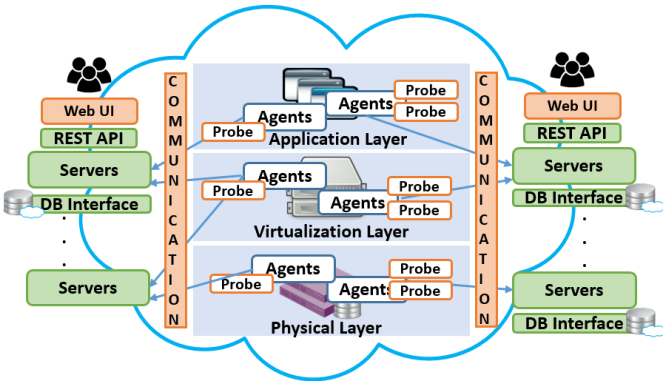


Fig. 1: JCatascopia Abstract Architecture

other systems (e.g. elasticity controller). For example, the Monitoring Database Interface (see Section 3.2.4) is extensible to allow developers or systems to utilize the metric storage backend of their choice via the Database Interface endpoint.

Second, its metric collection process is not dependent on cloud provider APIs or OS libraries. JCatascopia is designed to provide multi-level cloud monitoring and is capable of collecting heterogeneous metrics of different granularity across multiple levels of the cloud. Specifically, system-level metric collection, such for CPU, memory, disk and network utilization, is handled via the Java runtime management interface[2], so that metric collection does not pass directly through OS-dependent endpoints (e.g. `/proc/*` for UNIX). Nonetheless, users are free to create OS-dependent metric collectors via its Java API. Moreover, JCatascopia is enhanced with a metric rule mechanism, which allows developers or cloud entities (e.g. elasticity controller) to request for aggregated metrics or to compose high-level metrics from low-level metrics via a directive-based language introduced in Section 3.5.

Third, the communication between monitoring components, especially in the case of elasticity support, is capable of being handled over a mixture of networks and is not dependent on any cloud provider services or brokers. In contrast to other self-adaptive general-purpose monitoring tools [9] [14] [22], JCatascopia provides fully-automated elasticity support. Neither special entities deployed on physical nodes nor information from the hypervisor or any other central repository regarding the current running virtual instances are required. In Section 3.3, we present a mechanism based on a variation of the *pub/sub protocol* to dynamically detect at runtime elasticity action enfocement. In Section 3.4, we describe how the monitoring topology is automatically reconfigured in the presence of network failures or when monitoring servers are (de-)provisioned. We embrace the ZMQ message framework[3] to implement the JCatascopia communication mechanism which is built on top of ZMQ sockets. These sockets provide JCatascopia with abstract asynchronous message queues over public and private network interfaces for multiple messaging patterns. Unlike other message oriented middleware, ZMQ sockets run without a dedicated message broker. This is important since JCatascopia is designed as a brokerless and completely de-centralized system. Moreover, ZMQ allows JCatascopia to maintain its portability as the ZMQ Java bindings are OS-independent.

2. Java runtime management interface: goo.gl/WLIhdT
3. ZMQ Java library: https://github.com/zeromq/jeromq

3.2 Architecture and Components

Figure 1 depicts an abstract overview of JCatascopia monitoring framework architecture. JCatascopia follows an *agent-based*, *producer-consumer* architectural approach. Specifically, metric collectors, named *Monitoring Probes* (Section 3.2.1), gather metrics from the cloud element they reside on (e.g. VM or physical node) and performance metrics from deployed cloud services. *Monitoring Agents* (Section 3.2.2) are responsible for coordinating the metric collection process by managing Monitoring Probes and disseminating collected metrics to Monitoring Servers over the communication pane. *Monitoring Servers* (Section 3.2.3) are in charge of receiving, processing and storing metrics to the database backend of choice via the respective *Database Interface* (Section 3.2.4). If enabled by the user, Monitoring Servers can be accessed through a Web Interface. Communication over the monitoring topology follows the *JCatascopia Communication Protocol* (Section 3.3). This, enables automatic Monitoring Agent discovery and removal, and automatic resource configuration discovery (e.g. elastic IP attachment to VM) at runtime. Monitoring Servers, distributed even across multiple clouds, comprise the *Monitoring Topology* (Section 3.4). The Monitoring Topology is configurable with users able to select the topology fit for their needs. Moreover, the Topology is elastically scalable with "monitoring the monitoring system" capabilities (Section 4.5 & 4.6). Finally, as we show in our evaluation (Section 4.5), JCatascopia is scalable as it can cope with an increasing number of monitoring metrics and instances.

In the following, a more elaborative description of the components comprising JCatascopia is provided.

3.2.1 Monitoring Probes

Monitoring Probes are metric collectors responsible for collecting low-level metrics from VMs or physical machines and performance metrics from deployed cloud services. Probes feature both a push and pull metric delivery mechanism. Monitoring Agents benefit from the push mechanism by avoiding the overhead of constantly checking for metric updates. Thus, metrics can be collected at different time granularities as the collection process of each Probe is decoupled from other Probes. On the other hand, users or other interested parties (e.g. elasticity controller), may use the Monitoring Agent API to immediately pull Probe metric updates. Probes logically group multiple metrics together, in order to reduce the monitoring overhead when accessing common resources. For example, consider a Probe monitoring a load balancer which exposes values for its *active sessions*, *response time* and *error rate*, via a single REST call in JSON format. A JCatascopia Probe is able to share resources (i.e., HTTP connection, JSON parser) and reduce the computation overhead when collecting these metrics, whereas other monitoring tools (e.g. nagios [26]), which isolate each metric, require three REST calls and JSON parsers to accomplish the same task.

A metric filtering mechanism is introduced at Probe level. This allows users to attach filters to metrics (e.g. F=1%) . At runtime, the filter mechanism checks collected metrics and if metric values are in the range $[prevValue - F, prevValue + F]$ they will be discarded *in place* rather than being distributed through the network. Developers can take advantage of the JCatascopia Probe API[4], which provides an interface to the

4. Probe API and Repository available at https://goo.gl/kpFhMj

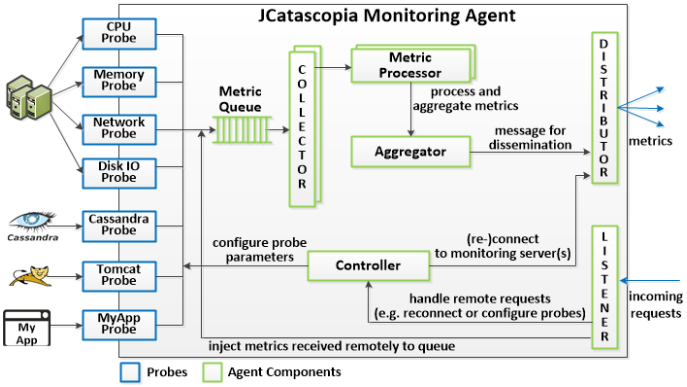


Fig. 2: JCatascopia Monitoring Agent

necessary abstractions that hide the complexity of Probe functionality when implementing Monitoring Probes. Monitoring Probes run independently from each other and can be deployed dynamically without the need to restart the monitoring process. If a Probe encounters a problem (e.g. unexpected termination) the metric collection of other Probes is not affected. Finally, Monitoring Agents encapsulate in their core a *dynamic java class loader*, which allows Probes to be dynamically *plugged-in* as lightweight monitoring threads, in two ways: (i) by compiling the Monitoring Agent with the source code of the Probe in the probe directory; or (ii) externally, by feeding the Monitoring Agent with the location (as a file path or URL) of the Probe bundled as a *jar* either via its config file or at runtime via its API.

3.2.2 Monitoring Agents

Monitoring Agents are light-weight monitoring instances deployable on cloud elements to be monitored, such as VMs or physical nodes. Monitoring Agents are responsible for managing the metric collection process on the respective cloud element, which includes processing and distributing metrics originating from Probes to Monitoring Server(s). An Agent is considered as the *probe manager* for the element it is deployed on. A Monitoring Agent is responsible for Monitoring Probe (de-)activation and configuration, according to user-defined parameter requests. Figure 2 depicts the internal architecture of a Monitoring Agent and its (sub-)components.

Initially, when a Monitoring Agent is deployed (see Section 3.3.1), the *Controller* establishes a connection to its respected Monitoring Server(s), as part of the *agent announcement process*. Once a connection and metric stream is established, monitoring metrics are published and made available for consumption. JCatascopia is able to control the message flow between Agents and Servers, adapting, if needed, to network transmission failures by re-connecting, re-scheduling and re-sending messages. New metric values are added to the Monitoring Agent *Metric Queue* either directly by Monitoring Probes or by the *Controller*, via the *Listener*, which listens for metric requests from other processes. The *Controller* also listens for probe parameter configuration requests (e.g. configure Probe collecting period) originating from either a Monitoring Server or by users via the JCatascopia REST API. Metrics are dequeued by *Metric Collectors* and processed by *Metric Processors*. Metric processing refers to preparing a message for distribution with the latest collected metrics. Initially, a metric is converted to a human readable format in a semi-structured manner and then Monitoring Agent metadata

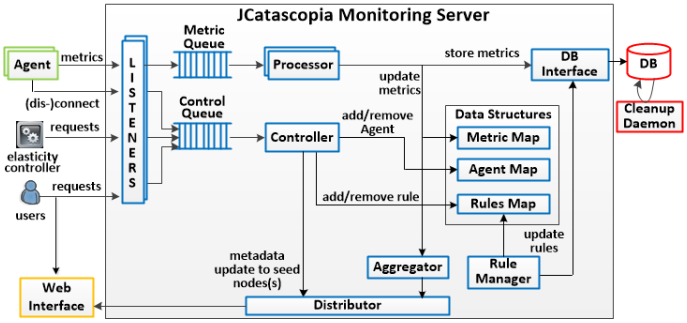


Fig. 3: JCatascopia Monitoring Server

are added to the message. The number of *Collectors* and *Processors* is customizable by simply changing the default values defined in the Agent configuration file (located in the installation directory).

After processing, metrics are passed to the *Aggregator*. The *Aggregator* is responsible for grouping metrics into messages and performing aggregation functions based on user-defined policies (i.e. AVG, MEAN). Aggregation is an important feature aiming at reducing network traffic from constantly transmitting metrics over the network. Built into JCatascopia is a *time-based* (e.g. distribute collected metrics every X seconds) and a *volume-based* policy (e.g. distribute metrics if message size exceeds X KB). Multiple aggregation policies can be utilized together (e.g. both of the above policies), with policies being configured through the Agent config file. Developers are free to create and attach their own custom aggregation policies by adhering to the aggregation interface. When an aggregation policy is satisfied, the message is distributed to all interested parties. Of course, aggregation is a trade-off between efficiency and accuracy. For this reason, in contrast to other monitoring tools, JCatascopia's Aggregator is fully configurable with users able to select which policies to enable.

3.2.3 Monitoring Servers

Monitoring Servers are the entities responsible for receiving, processing and storing monitoring metrics to the Monitoring Database. Monitoring Servers handle metric and configuration requests, delegating them to the appropriated Monitoring Agents. The communication between Monitoring Agents and Monitoring Servers is accomplished by utilizing a variation of the traditional *publish and subscribe* (pub/sub) messaging paradigm that reduces the related network communication overhead (see Section 3.3.1). A Monitoring Server processes received monitoring metrics and forms high-level metrics based on Metric Rules upon user request (see Section 3.5). Monitoring Servers are deployable on either physical nodes or virtual instances without having to reside in the same cloud platform with their Monitoring Agents. In particular, since JCatascopia is interoperable, both Monitoring Agents and Servers can be distributed across different cloud platforms. Using multiple Monitoring Servers is optional. However, sustainability, fault-tolerance and scalability can be improved if metric traffic is directed through multiple Monitoring Servers. Figure 3 depicts the internal architecture of a Monitoring Server and its (sub-)components. *Listeners* are the entities listening for (i) incoming Monitoring Agent connections, (ii) newly collected metrics, and (iii) API requests from other interested parties (e.g. elasticity controller or users). Monitoring Agent connection and termination requests are handled by the *Controller*, which parses requests and stores in suitable data structures (Agent and Metric Map) metadata describing the Agent and the collected metrics. After establishing a connection, an Agent *publishes* metric messages to the respected metric stream. *Listeners* receive incoming metric messages and enqueue them to the *Metric Queue*. Messages are dequeued from the *Metric Queue* and processed by *Metric Processors*. The number of Processors is customizable, by changing the default value defined in the Server configuration file. Processing messages refers to the task of parsing the message, decomposing it to grab the metrics in a message and updating the metric data structure. The metric data structure stores metric metadata and their latest reported values.

If a *Monitoring Database* is attached to the Monitoring Server, received metrics are stored via the *Database Interface* to the database. Also, if a Monitoring Server acts as an intermediate in a hierarchical topology (see Section 3.4), metrics are aggregated (if needed) and distributed to Monitoring Servers higher in the hierarchy. The *Rule Manager*, depicted in Fig. 3, is part of the rule mechanism, which is based on the *JCatascopia Rule Language* introduced in Section 3.5. The *Rule Manager* allows users and cloud entities (i.e. Billing entity) to apply expressions (e.g. $cpuTotal = 1 - cpuIdle$) on low-level metrics to create new high-level metrics. The *Rule Manager*, retrieves Rule requests from the *Control Queue* and updates existing Metric Rule values based on the updating period specified by the user via the JCatascopia REST API.

3.2.4 Monitoring Database Interface

JCatascopia offers users the ability to use the database solution of their choice to handle metric insertion and extraction. To this end, it provides a *Database Interface* to a database backend. Currently, JCatascopia supports two database backends: MySQL and CassandraDB. The MySQL implementation provides users and entities (i.e. elasticity controller) the ability to perform various types of complex queries on monitoring data, such as table joins. A *Cleanup Daemon* is available to extract and process old monitoring data to reduce the size of database and query response time. The Cleanup Deamon is activated either when the size of the database exceeds a specified by the user threshold or when a time interval expires. The NoSQL CassandraDB interface allows for fast writes and reads on recent metrics with also the ability to add a configurable Time-To-Live expiration parameter to inserted metrics, eliminating the need of a Cleanup Daemon.

3.3 JCatascopia Communication Protocol

3.3.1 Monitoring Agent Discovery and Removal

To support an automated *elasticity controller*, which adapts elastically cloud service deployments, Monitoring Agents must be both re-configurable and dynamically deployable at runtime. Specifically, when a newly provisioned virtual instance is added to a deployment, a new Monitoring Agent must be configured and added to this virtual instance. In turn, the monitoring system must be notified for this addition. Similarly, the monitoring system must also be aware when a Monitoring Agent has been removed due to the removal of a previously allocated virtual instance.

In the classic *publish and subscribe* message pattern, entities, referred to as *subscribers*, initially express interest and subscribe to an event stream of another entity, referred to

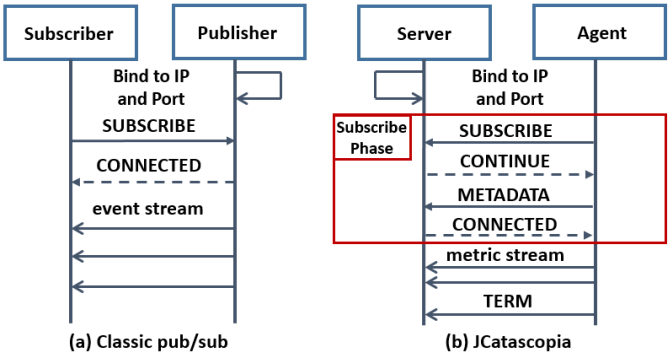


Fig. 4: Dynamic Agent Discovery and Removal Process

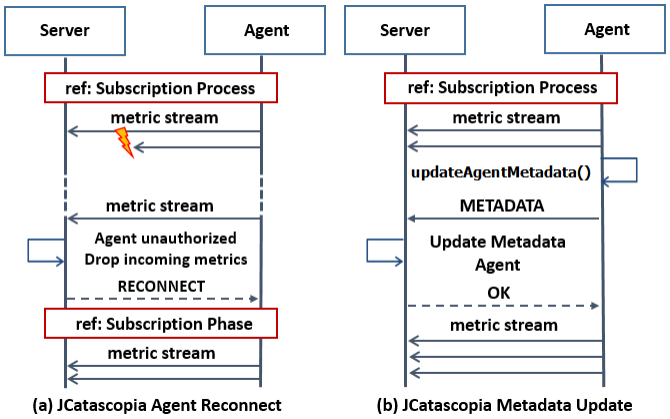


Fig. 5: Agent Automatic Reconnect and Metadata Update

as the *publisher*. A *subscriber* can be interested in receiving event notifications from multiple *publishers*. When events are generated, the publisher distributes them to its subscribers, eliminating the need of the subscriber to constantly poll the publisher to check if new events are available. Due to the reduced overhead from eliminating polling, this approach has been adopted by a number of monitoring systems [14] [32] [40]. However, this approach introduces the following limitations when monitoring rapidly elastic cloud service environments: (i) The addition of a new metric publisher (Monitoring Agent) requires for every potential subscriber (Monitoring Servers) to be notified of the new publisher's existence by a broker (or directory service) storing the network locations of Monitoring Agents and Servers. Then, interested subscribers will initiate the subscription process. This features a significant overhead in a highly adaptive large-scale environment where rapidly provisioned Monitoring Agents must contact all Monitoring Servers in the deployment; (ii) In a similar manner, when de-provisioning a Monitoring Agent, each and every subscriber must be notified and the metric stream(s) must be closed.

In our approach (Fig. 4) we differ from the classic pub/sub protocol by allowing Monitoring Agents to (dis-)appear dynamically and rapidly due to elasticity actions. Specifically, we vary the message pattern as follows: (i) Monitoring Servers bind to a network interface, awaiting for incoming requests; and (ii) Monitoring Agents, which are the metric publishers, initiate the subscription process by immediately contacting interested Monitoring Servers of their existence with a SUBSCRIBE message[5]. Afterwards, Monitoring Agents send a METADATA message to the interested Monitoring Servers including information such as the metrics they are responsible to collect and their agent id. Finally, after the subscription process is complete, Monitoring Agents can start publishing metrics to the established metric stream. With the proposed variation, the Monitoring Server is agnostic to the network location of its Monitoring Agents, allowing them to appear and disappear dynamically in a flexible manner by eliminating the need: (i) to restart or reconfigure the Monitoring System; (ii) to depend on the underlying hypervisor; and (iii) to require a directory service that contains these locations.

A Monitoring Agent is removed from the topology when either: (i) a scaling down action is issued to the VM it resides on; or (ii) due to a user shutdown request via the Monitoring Agent API or the OS (process is killed). In any case, upon Monitoring Agent termination a *java shutdown hook* is triggered. This shutdown hook initiates the *agent termination process* which gracefully stops Monitoring Agent functionality. Specifically, it immediately notifies the Monitoring Server(s) associated with it, that it is shutting down via a TERM message (Fig. 4). In contrast to heartbeat monitoring, this allows for Monitoring Agent removals to be rapidly discovered. Most importantly, in contrast to other solutions [40] which require 1 TERM message to the pub/sub broker and N messages to notify each Monitoring Server that the Agent is leaving the topology, our solution only requires M TERM messages, where M is the number of Monitoring Servers the Agent is assigned to (usually $M << N$).

3.3.2 Agent Automatic Reconnect and Metadata Update

JCatascopia takes into consideration realistic special-case scenarios which may be considered as exceptions to the smooth functioning of the monitoring process.

The first scenario considered is the presence of network connectivity issues between a Monitoring Agent and its respective Monitoring Server(s). For monitoring systems, if an Agent is unavailable for a specified period of time, then the connection is dropped. Afterwards, any incoming message or metric streams from the Monitoring Agent are marked as malicious and ignored with the Agent not having any knowledge that the metric values sent are discarded. The only way to re-establish the connection is to terminate and re-instantiate the Monitoring Agent. However, if a JCatascopia Agent (Fig. 5a) sends metric values or issues a request to a Monitoring Server after the connection is dropped, the Monitoring Server will reply with a RECONNECT message. This allows the Monitoring Agent to re-establish the connection (and re-authenticate) without the need to restart the monitoring process or the interference of a system admin.

The second scenario that we take into consideration relates to the uncertainties imposed due to re-contextualization such as in the case where the IP address of a VM is changed at runtime (elastic IPs are a form of vertical scaling). This scenario is not extreme, since a number of cloud providers offer elastic ip services [4] [28] by (de-)allocating network interfaces to/from virtual instances on the fly without restarting the instance. To address this, JCatascopia Agents periodically update their metadata (i.e. IP addresses or available metrics) and then send a METADATA message to the Monitoring Server as depicted in Figure 5b. The limitation of this approach is that a request (e.g. a metric pull request) issued from

5. Determining how a Monitoring Agents knows to which Monitoring Servers to connect to is described in Section 3.4

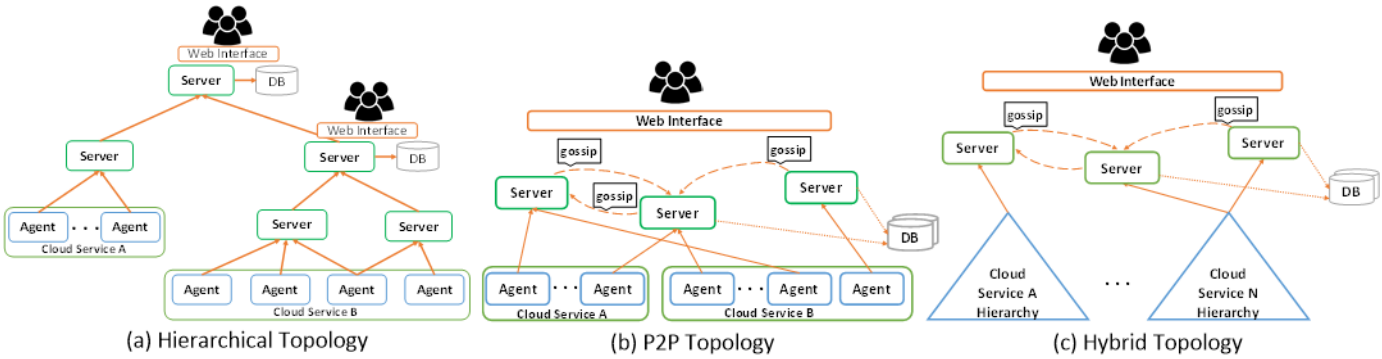


Fig. 6: Monitoring Topology Configurations

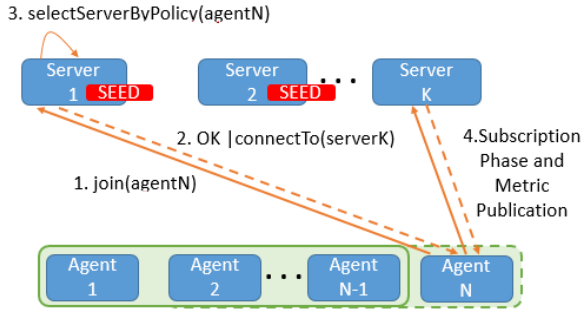


Fig. 7: Monitoring Agent Placement

a Monitoring Server to an Agent of a VM that just changed IP, will fail if the Agent metadata has not yet been updated. The error space can be shortened if the service issuing the configuration (e.g. elasticity controller) informs the Monitoring Agent of this by triggering either the `updateAgentIP()` or `updateALLAgentMetadata()` API calls. The same approach is followed when the IP of a Monitoring Server is changed with the addition of one more step: after an IP address update, the Monitoring Server notifies its respected Agents with a METADATA message containing the new IP address.

3.4 Monitoring Topology

JCatascopia as a modular system is comprised of the following components: *Probes*, *Agents*, *Servers*, *Database Interface*, *Web Interface*. JCatascopia grants users, service developers and monitoring providers (all noted as monitoring stakeholders) flexibility and freedom in the monitoring process. Specifically, monitoring stakeholders are free to configure the overlay network interconnecting Monitoring Agents and Servers to better suit their needs, focusing on *scalability*, *locality* or *logic of separation*. It is important to note that the utilized monitoring topology is transparent to the underlying Monitoring Agents, and consequently to user VMs. This means that no reconfiguration is required when the topology adapts or when substituted, even at runtime, with a different configuration. Figure 6 depicts three different topology configurations that can be considered by stakeholders for different purposes.

Figure 6a depicts a **hierarchical topology**, where the Monitoring Servers, and consequently their respective Monitoring Agents, are logically grouped together, forming a *tree*. Metrics processed from Monitoring Servers lower in the hierarchy can either be forwarded to Monitoring Servers higher in the hierarchy *as is* or aggregated presenting an overview of their branch. Additionally, a JCatascopia Web Interface and a Monitoring Database can be attached to any intermediate Monitoring Server collecting insights for the health and performance of the topology up to that point. Specifically, in Figure 6a, Cloud Service B features a dedicated Web Interface and Monitoring Database. As a side note, Monitoring Servers do not need to utilize the same Database Interface, e.g. Monitoring Servers lower in the hierarchy may use a relational database while others utilize a NoSQL database.

In contrast to the previous topology, Figure 6b depicts a **Peer-2-Peer topology**, where Monitoring Servers are distributed across the network forming a P2P *gossip* network. For JCatascopia, gossip is a compressed message exchanged between peers to periodically discover the state of other Monitoring Servers participating in the network, as well as the number of Monitoring Agents assigned to each peer and their network location. In this arrangement, all Monitoring Servers (peers) have the same monitoring responsibilities (e.g. receive, process, store metrics) with some peers also acting as *seeds*. A Seed is responsible for bootstrapping new peers joining the network and monitoring the health of the Monitoring Servers assigned to it, in par with P2P distributed database communication protocols [23]. Seeds are not a single point of failure nor do they serve any other special purposes. This topology benefits from automatic horizontal scaling based on the load imposed to the monitoring tier, if combined with an elasticity controller. In Section 4.5 and 4.6 we show how to monitor and scale an elastic monitoring system.

A **hybrid topology** featuring a combination of two or more topologies is another configuration supported by JCatascopia. Figure 6c depicts a hybrid topology where clusters of Monitoring Servers follow a hierarchical topology internally, although externally a P2P topology is used. This provides developers of multiple applications a high-level monitoring overview of their applications via the same web interface.

In addition to allowing developers configure the monitoring topology, JCatascopia provides developers with the ability to implement and integrate their own **Monitoring Agent placement policies**. Specifically, when a new Monitoring Agent attempts to establish a connection to the monitoring network, the placement policy is used to determine candidate Monitoring Server(s) for connection. Such placement policies may include: (i) assigning Monitoring Agents to Servers based on *fairness*, thus following a round-robin distribution; (ii) based on *locality*, where Monitoring Agents are assigned to the closest, in terms of latency, Monitoring Server(s) or in a multi-cloud scenario, to Monitoring Servers in the same availability zone; and (iii) based on Monitoring Server utilization, where agents are load-balanced amongst Monitoring Servers. To

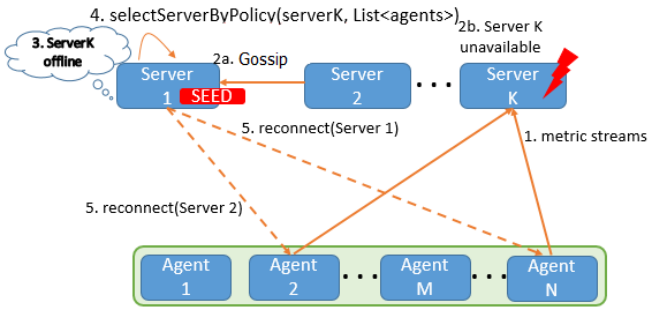


Fig. 8: Topology Reconfiguration After Server Fault

accommodate placement before the subscription process, a Monitoring Agent is notified by a seed node which Monitoring Servers are valid candidates for connection, as depicted in Figure 7. It must be noted, that an Agent upon deployment, may attempt to join the network via any Monitoring Server, and the request will be redirected to a seed node. Hence, no previous knowledge of seed location is required beforehand from Monitoring Agents.

3.4.1 *Monitoring Topology Automatic Recoverability*

Monitoring Servers, as Monitoring Agents, are subject to faults and network uncertainties. In a dynamic and elastic cloud environment, where users deploy their services and leave resource provisioning to be handled automatically [8], the monitoring system should not become a bottleneck or require manual configuration when the monitoring topology exibits problems due to sudden Monitoring Server unavailability. To accommodate such issues, JCatascopia undertakes the task of monitoring itself. Monitoring Servers, as depicted in Fig. 8, exchange periodically gossip containing the network location and status of Agents assigned to each peer. When a Monitoring Server is unavailable, its respective seed nodes rebalance the topology by requesting from the Monitoring Agents assigned to the faulty Server to reconnect to other Monitoring Server(s) based on the Monitoring Agent placement policy. This process is fully-automated and requires no human intervention. In addition, because seed nodes have no other special purpose and peers send gossip to multiple seeds (e.g. 3), if a seed node becomes unavailable, it is treated just as another peer failure. Moreover, rebalancing is not triggered only in faulty scenarios but also in the background after the (de-)allocation of a Monitoring Server to always reflect the current state by re-distributing the monitoring load.

3.5 Metric Rule Language

There are cases where entities are not interested in viewing monitoring metrics of a single instance but instead require an overview of the overall system or parts of it. Such a case is when an elasticity controller is required to autoscale a tier comprised of several VMs (e.g. database cluster) of a cloud service topology. In this case, the focus of the controller is shifted from single instances to an aggregated performance overview of the tier itself. To address this issue, we introduce a **Metric Rule Language**, which can be utilized to:

- Aggregate and group low-level metrics originating from single instances. For example, one may be interested in calculating the daily number of error responses issued from a web server by summing the collected *errorCount* metric: $SUM(errorCount)$;
- Generate high-level metrics dynamically at runtime from low-level metrics. For example, *Availability* can be defined from low-level metrics by applying the formula $Availability = 1 - downtime/uptime$.

A **Metric Rule Mechanism** is embedded in each Monitoring Server and can be reached by users either through the Monitoring Server REST API or graphically via the JCatascopia Web Interface. It must be noted that when utilizing the Web Interface, users do not need to have knowledge of the Metric Rule Language since everything is done graphically and later mapped in the background to respective JCatascopia directives. We describe Metric Rules as triplets with the following main elements:

`{Filter, Members, Action}`

The *Filter* is the part of the rule where a new metric is defined. The definition of a new metric consists of the operations (e.g. +, - ,*, /) to be applied to low-level metrics, collected from Monitoring Agents, and optionally a grouping function (e.g. AVG, SUM, MIN, MAX). The IDs of the selected Agents are specified in the *Members* part of the metric rule. Metric Rules may be updated at anytime (e.g. when a scale in/out action occurs) via a simple REST API call without the need to create a new rule each time. Figure 9 depicts the metric rule language in BNF. An exemplary Filter to create a new metric, named *DBthroughput*, which calculates the average throughput of a database cluster can be defined, by:

(i) the throughput of each node from the low-level metrics *readps* and *writeps* (reads/writes per second) as follows:

```
DBthroughput = readps+writeps
```

(ii) the aggregated throughput over each of the N nodes comprising the data cluster to find the average throughput:

```
DBthroughput = AVG(readps+writeps)
```

When a Metric Rule Filter is violated, the *Action* specified is enforced. Actions are either *time-based* (notify periodically) or *event-based* (notify if threshold is violated). An example of a time-based action, where the subscriber is notified periodically every 25 seconds is the following:

```
ACTION = PERIOD(25)
```

An example of an event-based action where the subscriber requests to be notified only if the newly created metric reports values lower that 25% or higher than 75% is the following:

```
ACTION = NOTIFY(<25,>=75)
```

With this approach, an elasticity controller does not need to poll the monitoring system to constantly check metric values as it is notified only when a violation occurs.

Finally, the complete example of a metric rule that calculates the average database throughput of a distributed database comprised of N nodes from the low-level metrics *readps* and *writeps*, notifying the subscriber when its values are lower that 25% or higher than 75%, is the following:

```
DBthroughput = AVG(readps+writeps)
MEMBERS = [agentID1, ... ,agentIDN]
ACTION = NOTIFY(<25,>=75)
```

4 Evaluation

In this section, we showcase JCatascopia by presenting: (i) its core functionality while monitoring real multi-cloud services deployed on both public and private clouds; (ii) a runtime comparison to other monitoring systems; (iii) a scalability evaluation; and (iv) a recoverability evaluation.

```
<Rule> ::= RULE = <Filter>, <Members>, <Action>

<Filter> ::= <Metric> = <Expr> | <GroupFunc>(<Expr>)
<Expr> ::= <Operand> | <Operand> <Op> <Expr>
<Operand> ::= <Number> | <Metric> | (<Expr>)
<Op> ::= +|-|*|/
<Metric> ::= <String>
<GroupFunc> ::= AVG|SUM|MIN|MAX|MEAN

<Members> ::= MEMBERS = ({<AgentID>,} <AgentID>)
<AgentID> ::= <String>

<Action> ::= ACTION = NOTIFY(<Act>) | PERIOD(<Number>)
<Act> ::= ALL | {<Relation> <Number>,} <Relation> <Number>
<Relation> ::= <|>|=|!=|>=|<=
```

Fig. 9: Directive-Based Rule Language in BNF

4.1 Testbed

In our experiments we use VMs of various flavors and operating systems, originating from 4 different cloud platforms. Specifically, our testbed consists of the following:

- 25 VMs from the GRNET Okeanos public cloud [28] with the following characteristics per VM: 1 VCPU, 1GB RAM, 10GB Disk, Ubuntu Server 12.04.4 LTS.
- 25 VMs from the Flexiant FlexiScale cloud [16] with the following characteristics per VM: 2 VCPU, 2GB RAM, 20GB Disk, CentOS 6.6.
- 25 Amazon EC2 [4] instances with 1VCPU, 2GB RAM, 160GB Disk and Debian 7.7 (Wheezy).
- 150 VMs from our own OpenStack private cloud [29] with the following characteristics per VM: 2 VCPU, 2GB RAM, 20GB Disk, Ubuntu Server 12.04.2 LTS.

To monitor the performance of the cloud services comprising our testbed, we have developed several Probes[6] using the *JCatascopia Java Probe API*. Table 1 presents these Probes and their default collecting period. We compare JCatascopia to two monitoring systems, which follow a similar agent-based architecture: (i) Ganglia [19], which is an open source, production-ready, general purpose monitoring tool; and (ii) Lattice [9], which is a monitoring framework that can be used to monitor elastically adaptive application environments and has a prototype available online. Therefore, on all the acquired virtual instances we have deployed JCastascopia Monitoring Agents, Ganglia gmonds and Lattice DataSources. In order for the comparison to be meaningful, we configure each monitoring system to report the same metrics at the same rate. Both, Ganglia and Lattice offer the CPU, memory and network metrics that JCatascopia offers as well. For the disk usage and application-specific metrics, we extend both Ganglia's metric library and Lattice, by implementing Python modules and Java Probes respectively. As an elasticity controller we use the open-source rSYBL elasticity controller [12] which already provides a JCatascopia monitoring interface.

4.2 Elastically Adapting a Three-Tier Web Service

In the first experiment, we elastically scale a **three-tier online video streaming service**, deployed on Amazon EC2. The video service is comprised of: (i) an *HAProxy Load Balancer* which distributes client requests (i.e., download/upload video) across multiple application servers with a CPU, Network and HAProxy Probe in use; (ii) An *Application Server Tier*, where each application server is an Apache Tomcat server exposing the video streaming web service. On each Application Server we deploy a CPU, Memory, Network, Tomcat and Video Probe; and (iii) A *CassandraDB Backend* hosting the video content (∼20GB). On each node we deploy a CPU, Network, DiskIO and Cassandra Probe.

6. All JCatascopia Monitoring Probes are open-source and available at: https://github.com/dtrihinas/JCatascopia-Probe-Repo

Probe	Metrics	Period (s)
CPU	cpuUserUsage, cpuNiceUsage, cpuSystemUsage, cpuIdle, cpuIOWait	5
Memory	memTotal, memUsed, memFree, memCache, memSwapTotal, memSwapFree	8
Network	netPacketsIN, netPacketsOUT, netBytesIN, netBytesOUT	10
Disk Usage	diskTotal, diskFree, diskUsed	30
Disk IO	readkbps, writekbps, iotime	15
HAProxy	activeSessions, requestRate, proxyBytesIN, proxyBytesOUT, avgResponseTime, serverCnt, errorRate, etc.	8
Cassandra	readLatency, writeLatency, nodeCnt, nodeSize, nodeload, cacheMissRate, etc.	20
Tomcat	maxThreads, currentThreadCount, currentThreadsBusy, bytesReceived, bytesSent, requestCount, errorCount, processingTime, reqThroughput, etc.	15
Video	downloadCnt, downloadSizeKB, uploadCnt, uploadSizeKB, CacheMissRate	25
CouchBase	itemCnt, ops, viewOps, load, writeLatency, readLatency, activeCons, etc.	20
MemCached	CasHits, CasMisses, CurrConnections, CurrItems, BytesReadIntoMemcached, BytesWrittenOutFromMemcached, etc.	8

TABLE 1: Available Probes

Initially, other than the static Load Balancer, the topology consists of 1 Application Server and 1 Database node. We stress this cloud service, as depicted in Figure 10, by generating random client requests of different type (i.e. download/upload video) and video length (between 2-10 minutes) under a variable request rate. To cope with the workload, our elasticity controller must scale in/out either the application and/or database tier, based on scaling actions, defined as JCatascopia metric rules for each tier respectively:

```
AvgActiveConnections = AVG(currentBusyThreads)
        MEMBERS = [id1, ... ,idN]          (R1)
      ACTION = NOTIFY(<70, >=140)

    AvgCPUUsage = AVG(1 - cpuIdle)
      MEMBERS = [id1, ... ,idN]            (R2)
    ACTION = NOTIFY(<30, >=85)
```

The metric rules notify the elasticity controller when a violation is detected and a scaling action is then enforced to the affected tier, as depicted in Figure 10. To better highlight when a scaling action occurs, Figure 11 depicts the metric values (and thresholds) for the first 30 minutes of the metric rule targeting the application server tier, as returned via the JCatascopia API.

To illustrate how JCatascopia can provide users with insights of their cloud services, let us consider a use-case scenario where the developer of the video service would like to know if trading *performance* with *cost* is beneficiary, in an attempt to reduce his cloud provider invoice. Specifically, the developer would like his service to elastically scale when the request rate rises while limiting the maximum number of VMs per tier to N, where e.g. $N = 5$. With this in mind, the developer believes: *limiting the number of VMs per tier will*

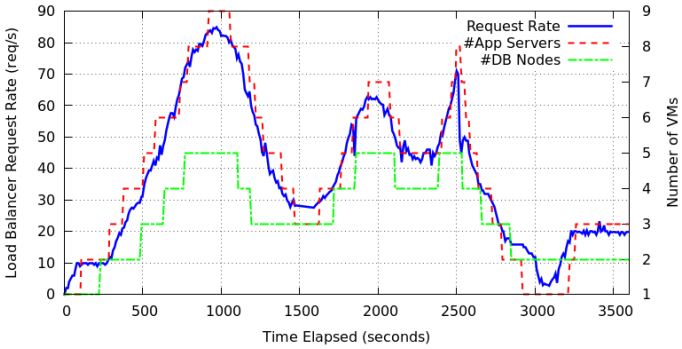


Fig. 10: Video Streaming Service Load and Number of VMs

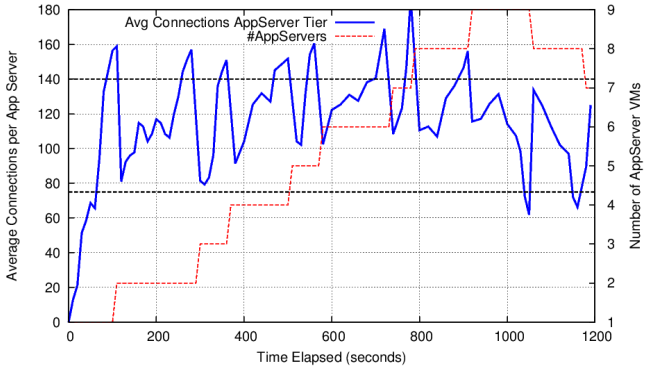


Fig. 11: Application Server Tier Average Connections

reduce costs with a propositional degradation in performance. We consider the following cost model, adopted by Amazon Web Services[7], to measure the cost of each case:

$$costVM = \lceil hoursRunning \rceil \cdot hourChargeRate + netOutTraffic \cdot transferChargeRate \quad (1)$$

$$total = costVM_{LB} + \sum_{m=1}^{|AS|} costVM_m + \sum_{n=1}^{|DB|} costVM_n \quad (2)$$

where based on the pricing scheme $transferChargeRate = \$0.12/GB$, $hourChargeRate = \$0.047/h$ and with $|AS|$ and $|DB|$ we denote the total number of VMs allocated per application and database tier, respectively.

Figure 12 depicts a cost comparison of letting the video service scale at will in contrast to *capping* the number of VMs per tier. We observe that, indeed, cost is reduced. Therefore, the video service developer now expects a similar performance degradation to be evident. However, in Figure 13 we observe that limiting the number of VMs results in a serious performance loss, in terms of response time, since user requests are queued and not served in time. Response time is dramatically affected for the specific application due to the high network traffic required to stream or upload a video (for this service it is between 50 to 95 MB per request). To make an even stronger case, let us consider this from a revenue perspective, where the developer, monitors the revenue of his application as a JCatascopia timeseries. In particular, revenue can be calculated by subtracting the costs from the profits. In turn, let us assume, that the profits are modelled based on youtube's reported 2014 Q4 revenue statistics [17], where it is estimated that per 1000 views Google makes a profit of $5.52. However, video requests requiring more than 20 seconds to load, are not considered when estimating profits as the connection is usually dropped by impatient viewers who do not wait for a video to load (such response time is only evident in the *capped* scenario). Therefore, as depicted in the revenue projection presented in Figure 14, limiting the number of VMs towards the high network traffic is not worth the tradeoff in terms of cost and performance. Thus, even though not a profiling tool, *JCatascopia, with its metric definition and aggregating mechanisms, can monitor cloud service resource allocation, cost and performance, providing, in turn, useful information to the decision-making mechanisms of an elasticity controller when planing to enforce intelligent elasticity control actions.*

7. http://aws.amazon.com/ec2/pricing/

4.3 Monitoring a Multi-Cloud Deployment

This experiment tests the feasibility of JCatascopia to monitor a deployment spread across multiple clouds. At the end of this experiment, a total of 75 VMs are utilized from the Amazon, Flexiant and Okeanos clouds. The evaluation is based on a multi-cloud deployment of an **Online Business Directory** hosting 7503 local businesses[8]. Specifically, the business directory is comprised of a *Manager* distributing user requests (i.e. get directions to shop X, top-K bars in Nicosia) as MapReduce-like queries to *Couchbase* database VMs.

The experiment starts with the deployment initially comprised of the *Manager* and a single *Couchbase* VM. A Couchbase VM is monitored by a JCatascopia Agent with 5 Monitoring Probes (CPU, Memory, DiskStats, MemCached and a CouchBase Probe). The multi-cloud deployment, as depicted in Figure 19, is stressed with a read-heavy load (report top-100 bars in the major cities of Cyprus) via an open-source workload generator[9]. Each Couchbase Probe exposes the current response time for the VM it resides on. We add the following JCatascopia Metric Rule to the Monitoring Server, in order to aggregate and report the overall response time averaged across all VMs in the multi-cloud deployment:

```
OverallResponseTime = AVG(CouchVMResponseTime)
        MEMBERS = [id1, ... ,idN]          (R3)
         ACTION = NOTIFY(>60sec)
```

When the threshold in the ACTION of the Metric Rule is violated, due to the increasing load (Fig. 19), the Monitoring Server immediately notifies the elasticity controller to take action. The elasticity controller then selects one of the three clouds and provisions a new Couchbase VM to cope with the increasing load. Selecting a cloud to provision the new VM is done in a round-robin fashion. The experiment stops when each cloud hosts 25 couchbase VMs, for a total of 75 VMs comprising the multi-cloud deployment monitored by JCatascopia. Figure 20 depicts the response time per cloud and the averaged response time across all clouds as measured by the aforementioned JCatascopia Metric Rule. Based on this, *JCatascopia can provide insights for the performance of multi-cloud deployments, allowing elasticity controllers to take more intelligent decisions for elasticity behavior analysis and VM placement to reduce costs and increase throughput.* For example, in this experiment we observe that Okeanos VMs initially present less response time and therefore, Okeanos should be preferred, but as the number of VMs per cloud

8. Real and synthetic data is provided by http://finditcyprus.com
9. https://github.com/UCY-LINC-LAB/WorkloadGenerator

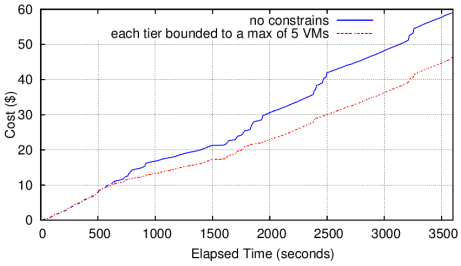


Fig. 12: Total Cost Comparison

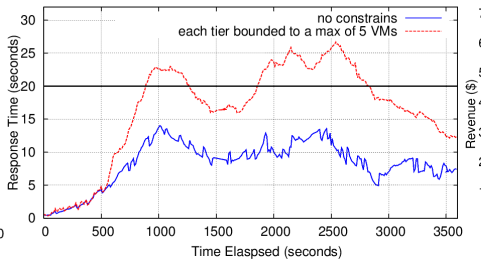


Fig. 13: Performance Evaluation

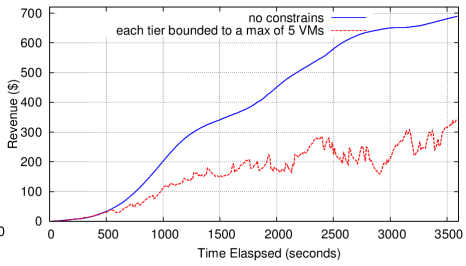


Fig. 14: Revenue Comparison

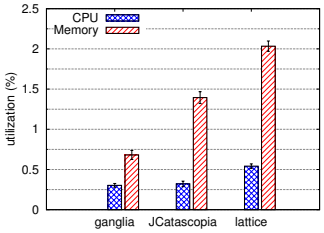


Fig. 15: Agent Utilization HAProxy

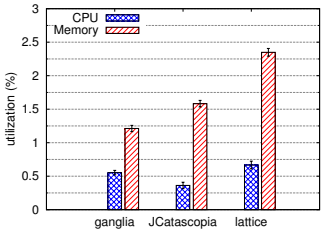


Fig. 16: Agent Utilization Cassandra

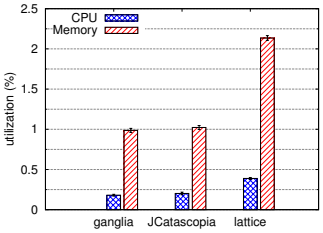


Fig. 17: Agent Utilization Online Directory

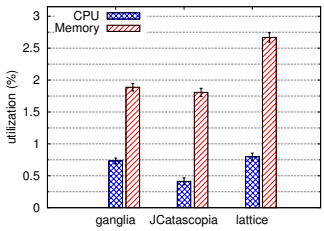


Fig. 18: Agent Utilization AppServer

increases, Amazon EC2 achieves a better performance while Flexiant, in all cases, features higher response times.

4.4 Runtime Impact Evaluation

In this section we study the impact Monitoring Agents have on *user paid* VMs. We compare JCatascopia to Ganglia and Lattice in terms of the overhead and cost imposed by each monitoring tool to the VMs of the services presented in the previous sections. To explore in depth the difference between the compared systems we used small VM flavors. As a side note, because Ganglia and Lattice do not support multi-cloud deployments, the online business directory (75VMs) was re-deployed on the Openstack cluster configured to use the Xen hypervisor (requirement for Lattice). Finally, while JCatascopia is capable of monitoring itself, the other tools are not. Therefore, to be fair, a system daemon on each VM was utilized to periodically collect (every 1s) the runtime overhead of each monitoring system from the underlying Linux OS[10].

Figures [15-18] depict the comparison in terms of CPU and memory utilization. From these figures, we notice that Lattice runtime footprint is larger than the other two monitoring systems. From Figure 15, we observe that Ganglia's footprint is smaller than JCatascopia. This is due to collecting primarily low-level system metrics and only a few application-specific metrics on HAProxy. Ganglia is lightweight when utilizing its built-in metrics but its runtime footprint rises significantly when increasing the metric count by deploying user-developed Python modules (external processes) targeting application level metrics. This is noticeable in Figure 16, where Ganglia's memory footprint doubled; in Figure 17 where the difference between Ganglia and JCatascopia is under 0.03%; and finally, in Figure 18, where JCatascopia has a smaller runtime impact. In regards to network utilization, as depicted in Figure 21, we observe that JCatascopia has an inherently smaller network footprint than the other monitoring tools. When enabling filtering (1% filter window), a feature absent from the other tools, we observe that JCatascopia network overhead drops even more. In addition, Figure 22 depicts the above translated in terms of cost/h, based again on AWS pricing scheme, for the online business directory (Section 4.3). From this, we observe that monitoring traffic over the network imposes a significant cost consideration factor, with the difference in cost/h between the other systems and JCatascopia increasing as time progresses. In conclusion, *when in need of application monitoring, JCatascopia reduces monitoring network traffic and consequently the monitoring cost, which is both noticeable and billable (i.e. network traffic on AWS is charged $0.12/GB,) in a large-scale distributed deployment.*

10. Each deamon collects CPU and Memory usage via the `top -p` command and outgoing network traffic via `nethogs` command

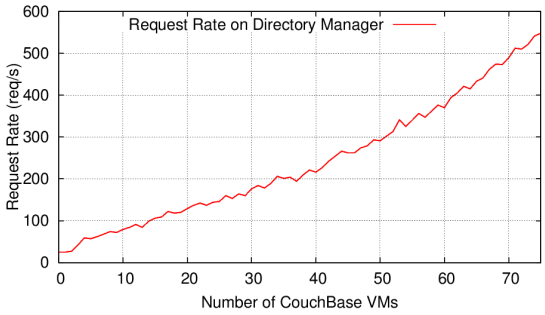


Fig. 19: Request Rate of Generated Load on the Manager

4.5 Scalability Evaluation

The goal of this experiment is to evaluate JCatascopia scalability under different elastic monitoring topology configurations, while the number of Monitoring Agents, and consequently the number of metrics, increases. When scaling, the performance of a Monitoring Agent, is not affected, as each Agent is an independent entity responsible of collecting metrics originated from only a single instance. However, this does not apply to a Monitoring Server, which depends highly on the number of Agents in the deployment, the monitoring intensity and the number of metrics.

This evaluation is performed on a total of 150 VMs originating from our Openstack private cloud. Each VM (Ubuntu Server 12.04.2, 2VCPU, 2GB RAM), runs an Apache Tomcat Servlet Container (v7.0.56) with a web service executing, per request, a compute intensive task for a random time period

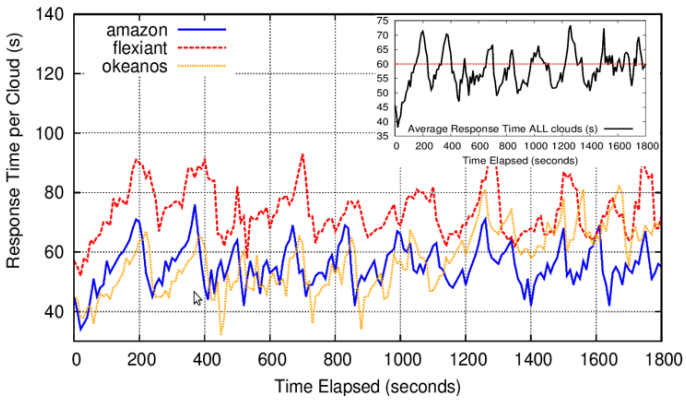


Fig. 20: Response Time per Cloud and Averaged Across All Clouds (Top Right Figure)

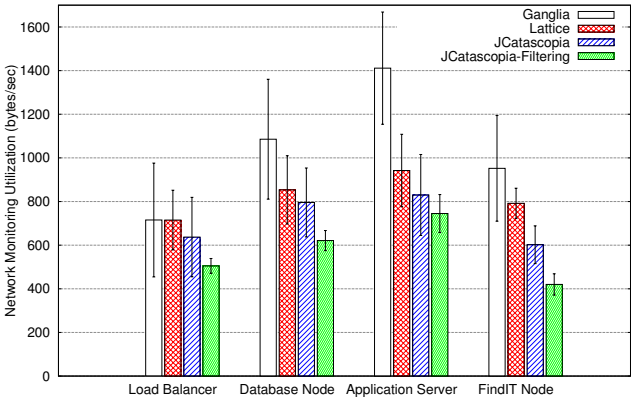


Fig. 21: Monitoring Agent Network Utilization

(15-90 seconds) and a Monitoring Agent collecting a total of 31 (system- and application-specific) metrics. The initial monitoring topology is comprised of 1 VM and the elasticity controller is configured to instantiate randomly, every 2 to 5 minutes, a new VM and add it to the deployment without the need to restart or reconfigure any part of the monitoring process. This is performed until the deployment numbers 150 VMs reporting metrics.

To evaluate JCatascopia's performance while scaling, we measure *archiving time* which is the average time required by a Monitoring Server to process and store a received metric to the Monitoring Database. A Metric Rule is defined to calculate the Monitoring Server maximum archiving time reported in the deployment where, essentially, we "Monitor the Monitoring System", as follows:

```
MaxArchiveTime = MAX(MonServerArchiveTime)
       MEMBERS = [id1, ... ,idN]          (R4)
        ACTION = NOTIFY(ALL)
```

The process described is repeated for each of the following configurations with Figure 23 depicting the results of the comparison. It must be noted, that although Monitoring Servers follow a different configuration, the user deployment is agnostic of the underlying topology requiring zero re-configuration effort from a user perspective.

At first, we utilize 1 Monitoring Server for the whole deployment with a RDBMS (MySQL) database backend. From Figure 23, we observe that this achieves an archiving time that grows linearly. However, if the metric publishing rate is higher than the archiving rate, metrics will queue at the Monitoring

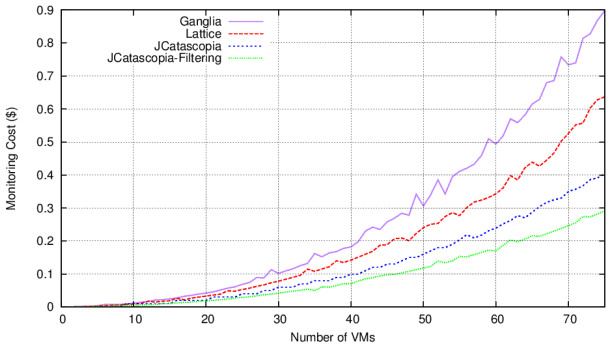


Fig. 22: Monitoring cost/h for a Multi-Cloud Application

Server and possibly be dropped. At this point one may argue that using a RDBMS is wrong. However, RBDMSs provide *joins* on tabular data, a functionality important to discover correlations among data from heterogeneous sources. For example, [13] analyses cloud service behavior by correlating monitoring metrics, provider-specific information (i.e. pricing schemes, user quotas), and service topology descriptions. To keep archiving time low, data after 2 weeks is compressed and archived in a HDFS. The next two configurations replace the RDBMS with a distributed NoSQL backend (CassandraDB) comprised of 1 and 2 nodes respectively. We observe that as the number of Monitoring Agents increases, a significant performance gain is observed. This, is primarily due to the fact that writing to the database (Cassandra writes are cheap) approximates 68% of the total processing time. Thus, this justifies why we evaluate scalability via archiving time instead of Monitoring Server CPU and Memory usage which in all configurations never passed 0.6% and 2.9% respectively.

In the fourth configuration, we establish a tree hierarchy by configuring 2 Monitoring Servers as intermediates which are utilized to process, aggregate and distribute monitoring metrics from the underlying Agents to a root Monitoring Server. The root Monitoring Server stores the metrics in the Monitoring Database. For this topology we observe that archiving time remains relatively stable for up to 85-90 VMs but as the number continues to grow, the root Monitoring Server becomes a bottleneck. We then, configure the topology to a P2P topology where Monitoring Servers are placed horizontally, each receiving, processing and then storing metrics to the database backend. From Figure 23 we observe that JCatascopia manages to keep archiving time relatively stable. Therefore, *when archiving time is considered high (e.g. archiving time > 100ms), monitoring traffic can be redirected through additional Monitoring Servers. This will result in a significant performance gain, allowing the monitoring topology to scale to any number of instances.* In addition, *the monitoring system can be elastically scaled in and out by an elasticity controller, without requiring additional effort and the presence of a system administrator.*

4.6 JCatascopia Recoverability in a MaaS Configuration

In this experiment we show that JCatascopia can recover from Monitoring Server "crashes" by automatically reconfiguring a monitoring topology established to offer Monitoring-as-a-Service. In addition, recovery must happen in a timely manner by employing the *recovery protocol* described in Section 3.4. For our evaluation, we utilize a MySQL monitoring database and define the same metric rule as in Section 4.5 (see R4)

but with a very low threshold (22ms), thus allowing a testbed comprised of 120 VMs to be scaled and distributed evenly among 5 Monitoring Servers in a P2P topology.

At first, network traffic is randomly blocked to 1 of the 5 Monitoring Servers, which becomes unavailable. This enables the recovery protocol which rebalances the topology. Specifically, affected Monitoring Agents are notified to reconnect to different Monitoring Server(s) based on the used Agent placement policy (we use a "fairness" policy, evenly load-balancing Agents to Servers) which suggests candidate Monitoring Servers to each Agent. When the topology returns to a stable state, we restore access to the faulty Monitoring Server and repeat the experiment but in the next iteration we simultaneously "crash" an additional Monitoring Server.

To evaluate recoverability, we measure: (i) *Rebalance Time* (*RBT*), which is the time required to determine (based on the applied policy) candidate Monitoring Servers for the affected Monitoring Agents and afterwards notify them. RBT highly depends on the number of faulty Monitoring Servers (and consequently Monitoring Agents affected) and the complexity of the placement policy; (ii) *Agent Reconnect Time* (*ART*), which is the time required for a Monitoring Agent to reconnect to a new Monitoring Server after notified about current Monitoring Server unavailability; and (iii) *Total Recovery Time* (TRT), which is the total time required for the system to return to a "stable" state. TRT is measured in our experiments, but can be modelled, as follows:

$$TRT = RBT + Max(ART) + \epsilon$$

where TRT is equal to the sum of RBT, the worst ART and an error margin (e.g. for network latency in a multi-cloud deployment). It is important to mention that while TRT depends on the worst ART, the recovery and, consequently, the monitoring process is not affected by a slow Agent reconnection as Monitoring Agents (re-)connect in parallel. Figure 24 depicts the results for each iteration of the experiment, where for 120 VMs distributed across 5 Monitoring Servers, *the monitoring process can automatically recover from any number of Monitoring Server crashes, observing linear recoverability.*

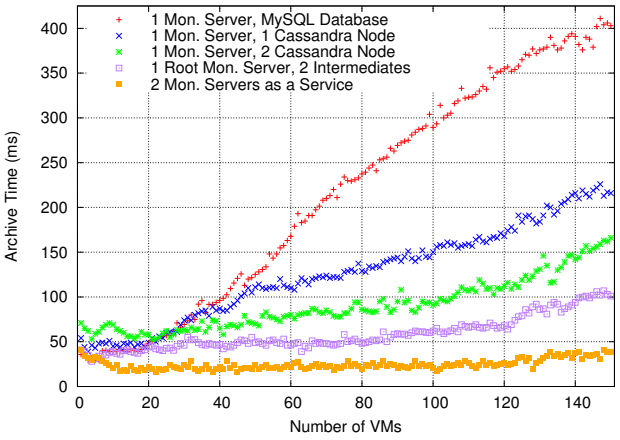


Fig. 23: Monitoring Server Average Archiving Time

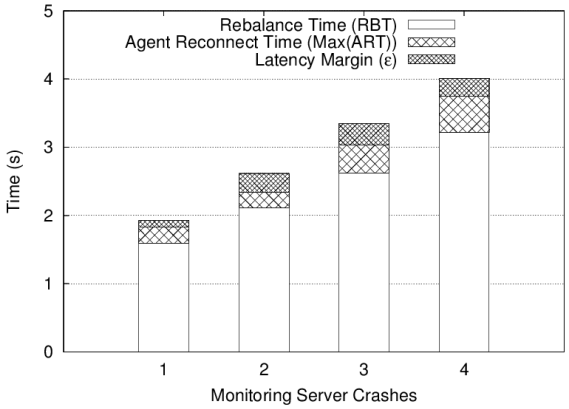


Fig. 24: MaaS Protocol Recoverability

5 Conclusion and Future Work

In this article, we introduce the design and key novelties of the JCatascopia Cloud Monitoring Framework. JCatascopia is a fully-automated, modular, multi-layer and interoperable monitoring framework. JCatascopia runs in a non-intrusive and transparent manner to any underlying virtualized infrastructure and is capable of detecting configuration changes due to elasticity actions in a cloud service deployment based on a novel variation of the pub/sub protocol. JCatascopia is equipped with a metric rule mechanism to generate, aggregate and group low-level monitoring metrics at runtime. Furthermore, JCatascopia is highly configurable, allowing its users to deploy it in different monitoring topologies and is capable of automatically recovering from Monitoring Server faults and network problems introduced, at runtime, in the monitoring configuration. Experiments on both public and private clouds show that JCatascopia is a suitable monitoring system to support a fully automated cloud resource provisioning system with proven interoperability, scalability, fault-tolerance and with a low runtime footprint. Most importantly, our framework is able to reduce network traffic by 41% over state-of-the-art solutions and consequently the monitoring cost, which is noticeable, billable and rises very fast in large elastic distributed multi-cloud deployments. Finally, JCatascopia is currently integrated with: (i) CELAR [8], a fully automated multi-grained platform which elastically provisions resources for cloud services; (ii) the ADVISE framework [13], which features a clustering-based learning mechanism for estimating and evaluating cloud service elasticity behavior; (iii) the rSYBL [12] elasticity controller; and (iii) CAMF [10], the newly established Eclipse Foundation Cloud Application Management Framework.

As future work, we are in the process of enhancing Monitoring Probes capabilities by pursuing *adaptive filtering* and *sampling* to dynamically adjust, at runtime, the metric filtering and collecting period based on the current workload [36]. This results in minimizing the communication, storage and computation overhead. Furthermore, we are in the process of developing a PaaS interface for our Monitoring Agents, allowing them to be seamlessly integrated and bundled in PaaS cloud services. Finally, JCatascopia will be equipped with the appropriate interface to monitor and identify performance and network interferences between VMs co-located on the same physical nodes.

Acknowledgement. This work is partially supported by the EU Commission in terms of the CELAR FP7 project (FP7-ICT-2011-8) and PaaSport FP7 project (FP7-SME-2013).

References

[1] G. Aceto, A. Botta, W. de Donato, and A. Pescape, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093 – 2115, 2013.

[2] J. Alcaraz Calero and J. Gutierrez Aguado, "Monpaas: An adaptive monitoring platform as a service for cloud computing infrastructures and services," *Services Computing, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[3] Amazon CloudWatch, http://aws.amazon.com/cloudwatch/.

[4] Amazon EC2, http://aws.amazon.com/ec2/.

[5] S. Andreozzi, N. De Bortoli, S. Fantinel, A. Ghiselli, G. L. Rubini, G. Tortone, and M. C. Vistoli, "Gridice: a monitoring service for grid systems," *Future Gener. Comput. Syst.*, vol. 21, no. 4, pp. 559–571, Apr. 2005.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[7] D. Armstrong, D. Espling, J. Tordsson, K. Djemame, and E. Elmroth, "Runtime virtual machine recontextualization for clouds," in *Proceedings of the 18th Inter. Conf. on Parallel Processing Workshops*, ser. Euro-Par'12, 2013, pp. 567–576.

[8] CELAR Project, http://celarcloud.eu/.

[9] S. Clayman, A. Galis, and L. Mamatas, "Monitoring virtual networks with lattice," in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, 2010.

[10] Cloud Application Management Framework (CAMF), Eclipse, https://projects.eclipse.org/projects/technology.camf.

[11] Cloud Computing Trends: 2014 State of the Cloud Survey, http://goo.gl/8i6ZPz.

[12] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "Multi-level elasticity control of cloud services," in *Service-Oriented Computing*. Springer Berlin Heidelberg, 2013, pp. 429–436.

[13] G. Copil, D. Trihinas, H.-L. Truong, D. Moldovan, G. Pallis, S. Dustdar, and M. Dikaiakos, "Advise: A framework for evaluating cloud service elasticity behavior," in *Service-Oriented Computing*. Springer Berlin Heidelberg, 2014, vol. 8831.

[14] M. B. de Carvalho and L. Z. Granville, "Incorporating virtualization awareness in service monitoring systems." in *Integrated Network Management*, N. Agoulmine, C. Bartolini, T. Pfeifer, and D. O'Sullivan, Eds. IEEE, 2011, pp. 297–304.

[15] V. C. Emeakaroha, M. A. Netto, R. N. Calheiros, I. Brandic, R. Buyya, and C. A. D. Rose, "Towards autonomic detection of {SLA} violations in cloud infrastructures," *Future Generation Computer Systems*, vol. 28, no. 7, 2012, special section: Quality of Service in Grid and Cloud Computing.

[16] Flexiant FlexiScale Platform, http://www.flexiscale.com/.

[17] Forbes - Youtube 2014 Q4 Earnings, http://goo.gl/2I6QAY.

[18] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE '08*, Nov 2008, pp. 1–10.

[19] Ganglia, http://ganglia.sourceforge.net/.

[20] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *ICAC*, 2013, pp. 23–27.

[21] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing - degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 7:1–7:28, Aug. 2008.

[22] G. Katsaros, R. Kubert, and G. Gallizo, "Building a service-oriented monitoring framework with rest and nagios," in *2011 IEEE International Conference on Services Computing (SCC)*, 2011, pp. 426–431.

[23] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[24] B. D. Martino, G. Cretella, and A. Esposito, *Cloud Portability and Interoperability - Issues and Current Trends*, ser. Springer Briefs in Computer Science. Springer, 2015.

[25] J. Montes, A. Sanchez, B. Memishi, M. S. Perez, and G. Antoniu, "Gmone: A complete approach to cloud monitoring," *Future Generation Computer Systems*, 2013.

[26] Nagios, http://www.nagios.org/.

[27] North Bridge and Gigaom Research 2014 4th Annual Future of Cloud Computing Survey, http://goo.gl/e6xkYL.

[28] Okeanos Public Cloud, https://okeanos.grnet.gr/.

[29] OpenStack, https://www.openstack.org/.

[30] Paraleap AzureWatch, https://www.paraleap.com/AzureWatch.

[31] 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, 2013.

[32] D. L. Quoc, L. Yazdanov, and C. Fetzer, "Dolen: User-side multi-cloud application monitoring," in *IEEE Future Internet of Things and Cloud*, 2014.

[33] Rackspace Cloud Monitoring, http://goo.gl/4BfqVf.

[34] M. Rak, S. Venticinque, T. Mahr, G. Echevarria, and G. Esnal, "Cloud Application Monitoring: The mOSAIC Approach," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, Nov 2011.

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

[36] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "AdaM: an Adaptive Monitoring Framework for Sampling and Filtering on IoT Devices," in *IEEE International Conference on Big Data*, 2015.

[37] D. Trihinas, G. Pallis, and M. Dikaiakos, "JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 226–235.

[38] D. Trihinas, C. Sofokleous, N. Loulloudes, A. Foudoulis, G. Pallis, and M. D. Dikaiakos, "Managing and Monitoring Elastic Cloud Applications," in *14th International Conference on Web Engineering*. Springer Publishing, 2014, pp. 523–527.

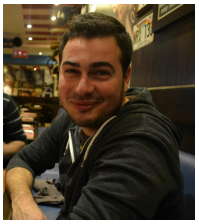
[39] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for nosql clusters using tiramola," *IEEE International Symposium on Cluster Computing and the Grid*, vol. 0, pp. 34–41, 2013.

[40] R. B. Uriarte and C. B. Westphall, "Panoptes: A monitoring architecture and framework for supporting autonomic clouds," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, May 2014, pp. 1–5.

[41] J. S. Ward and A. Barker, "Self managing monitoring for highly elastic large scale cloud deployments," in *Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing*, ser. DIDC '14. New York, NY, USA: ACM, 2014.

[42] G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen, and F. Zhao, "Vmdriver: A driver-based monitoring mechanism for virtualization," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, Oct 2010, pp. 72–81.

[43] Zabbix, http://www.zabbix.com/.



Demetris Trihinas is a PhD candidate at the Department of Computer Science, University of Cyprus. He holds a MSc in Computer Science from the University of Cyprus and a Dipl.-Ing. in Electrical and Computer Engineering from the National Technical University of Athens. His research interests include Application Monitoring, Cloud Computing, and Distributed Systems Behavior Analysis and Estimation. His work is published in IEEE/ACM conferences such as CCGrid, BigData, ICSOC, ICWE and EuroPar.

George Pallis is assistant professor at the Computer Science Department, University of Cyprus. His research interests include Cloud Computing with focus on Cloud Elasticity and Monitoring, Content Delivery Networks and Online Social Networks. His publication record is now at more than 60 research papers which have appeared in journals, book chapters, and conferences. He is Associate Editor in Chief in the IEEE Internet Computing magazine and he is also editing the "View from Cloud" department in this magazine.

Marios D. Dikaiakos is a Professor of Computer Science at the University of Cyprus, where he served as Department Head between 2010-2014. He received his Dipl.-Ing. in Electrical Engineering from the National Technical University of Athens in 1988, and M.A and Ph.D. degrees in Computer Science from Princeton University in 1991 and 1994, respectively. His research focuses on large-scale distributed systems with emphasis on designing, implementing and experimenting with systems and tools for Cloud and Grid Computing, and the World-Wide Web. He has directed many funded research projects, published over 150 original peer-reviewed scientific papers, and lead the development of several (open-source) software tools.