JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud

Demetris Trihinas, George Pallis, Marios D. Dikaiakos

Department of Computer Science
University of Cyprus
Email: { trihinas, gpallis, mdd }@cs.ucy.ac.cy

***Abstract*—Over the past decade, Cloud Computing has rapidly become a widely accepted paradigm with core concepts such as elasticity, scalability and on-demand automatic resource provisioning emerging as next generation Cloud service -must have- properties. Automatic resource provisioning for Cloud applications is not a trivial task, requiring for both the applications and platform, to be constantly monitored, capturing information at various levels and time granularity. In this paper we describe the challenges that occur when monitoring elastically adaptive Cloud applications and to address these issues we present JCatascopia; a fully automated, multi-layer, interoperable Cloud Monitoring System. Experiments on different production Cloud platforms show that JCatascopia is a Monitoring System capable of supporting a fully automated Cloud resource provisioning system with proven interoperability, scalability and low runtime footprint. Most importantly, JCatascopia is able to adapt in a fully automatic manner when elasticity actions are enforced to an application deployment.**

Keywords*—*Cloud Computing; Cloud Monitoring; Application Monitoring; Elasticity

I. Introduction

The concept of Cloud Computing is dominating the interests of organizations, as they seek to empower their business units to promptly address market opportunities, while at the same time aiming to minimize their running IT costs. Elasticity in Cloud Computing allows the Cloud environment to -ideally automatically- assign a dynamic number of resources to a task, aiming to ensure that the amount of resources needed for its execution is indeed provided to the respective task [29]. Elasticity in Cloud Computing is still considered a primary open research issue [7][30]. Many Cloud providers and systems claim that they offer elasticity but usually only provide scalability in terms of increasing availability through horizontal replication. For instance, Amazon's AutoScaling [2] provides elasticity in a semi-automatic manner. This allows Amazon EC2 [4] instances to be seamlessly and automatically added when demand increases, based upon manually defined boolean formulas and monitoring metrics collected by CloudWatch [3].

Monitoring Systems deployed on large-scale distributed infrastructures, such as Clouds, are essential for capturing the performance and understanding the elastic behavior of an application under various circumstances and possibly unexpected workloads. On one hand, monitoring is a key tool for controlling and managing software and hardware infrastructures; on the other hand, it provides information and key performance indicators for both platforms and applications.

One of the most challenging tasks in Cloud Computing is resource and capacity management [1]. Once the problem of enabling developers to manage Cloud resources is solved, in both a clear and flexible way, a new problem emerges: *How to automatically provision resources for Cloud applications?* Although Cloud infrastructures are inherently elastic, allowing applications to throttle resources on-demand [33], to the best of our knowledge, there exists no open-source Monitoring System fully capable of supporting a multi-grained, elastic *Resource Provisioning System* for deployed Cloud applications in an automated manner [10]. However, this is not a trivial task, since it requires for both the applications and platform, to be constantly monitored, capturing information at various levels and time granularity. Aceto et al. [1] advocate that accurate and fine-grained monitoring activities are required to efficiently operate Cloud platforms and to manage their increasing complexity. Specifically, a Monitoring System should monitor heterogeneous types of information of different granularity, from low-level system metrics (e.g. CPU usage, network traffic, memory allocation, etc.) to high-level application specific metrics (e.g. throughput, latency, availability, etc.), which are collected across multiple levels (physical, virtualization, application level) in a Cloud environment at different time intervals. Furthermore, the recipients of monitoring metrics are also heterogeneous. For instance, a particular metric (e.g. CPU utilization of a VM) can be accessed -frequently and simultaneously- by many entities (e.g. Cloud service consumers, resource provisioner, Cloud provider) but is interpreted differently.

Cloud providers typically offer advanced monitoring facilities [3][9]; also general purpose monitoring tools such as Ganglia [20] or Nagios [24] serve this purpose in highly specialized configurations or extensions. However, existing monitoring tools cannot run on any underlying virtualized infrastructure thus binding them to operate on a limited number of Cloud platforms. In addition, they require re-contextualization [8] when an application and/or resource related parameter changes. Finally, they fail to detect configurations in the application topology (e.g. new VM is added) that occur due to elasticity actions unless this

information is either obtained from the underlying hypervisor [11] or from the Cloud provider through a directory service [14]. In this paper we address the above challenges focusing on the issues that occur when monitoring elastically adaptive Cloud applications. As a result, we introduce **JCatascopia**; a multi-layer, interoperable Cloud Monitoring System which offers the following features:

- is an open-source Cloud Monitoring System that can run in a non-intrusive and transparent manner to any underlying virtualized infrastructure;
- dynamically detects at runtime when monitoring instances have been added/removed from the overall system due to enforcing elasticity actions without any human intervention or the need to restart the Monitoring System;
- is application-adaptive, diminishing the need for recontextualization each time an application and/or resource related parameter changes;
- generates high-level application metrics dynamically at runtime by aggregating and grouping low-level metrics;
- provides filtering capabilities to reduce the communication overhead for metric distribution and storage.

In this paper, we also present a thorough evaluation of our system by comparing it to other monitoring tools [11][20]. The experiments are conducted with a testbed that utilizes: (i) different domains of Cloud applications [6][13][27], (ii) various VM flavors, and (iii) both public and private Cloud infrastructures [4][26][17]. Results show that JCatascopia is a suitable Monitoring System to support a fully automated Cloud resource provisioning system with proven interoperability, scalability and low runtime footprint. Most importantly, JCatascopia is able to adapt in a fully automatic manner when elasticity actions are enforced to an application deployment.

The rest of the paper is structured as follows: Section 2 presents a study of the related work in the field of Cloud Monitoring. Section 3 presents the design concepts and the architecture of JCatascopia. Section 4 provides a detailed description of the implementation of each component that comprise JCatascopia. Section 5 presents an evaluation of our system while Section 6 concludes this paper and outlines the future work.

II. Related Work

Cloud specific monitoring tools such as Amazon CloudWatch [3], Paraleap AzureWatch [9] and RackSpace CloudKick [28] provide Monitoring as a Service to Cloud service consumers. Despite the fact that these tools are easy to use, fully documented and well-integrated with the underlying platform, their biggest disadvantage is that many of them are commercial and proprietary which limits them to operating on specific Cloud IaaS providers. Thus, these tools lack in terms of portability and interoperability.

General purpose monitoring tools such as Ganglia [20], Nagios [24], Zabbix [35], MonALISA [25] and GridICE [5] are used traditionally by system administrators to monitor slowly changing and fixed large-scale distributed infrastructures, such as Computing Grids and Clusters. Cloud providers tend to adopt such solutions to monitor their infrastructures. However, Cloud platforms are inherently more complex than Grid infrastructures, consisting of multiple layers and service paradigms (IaaS, PaaS, SaaS) providing users and applications with *on-demand* resources through an *infinite* pool of virtual resources. This makes the aforementioned monitoring tools unsuitable for addressing a rapidly adapting and dynamic Cloud infrastructure where, for example, a virtual instance is deployed for several minutes on one physical node and after a short interval it can migrate to another node.

To address the above limitations, several approaches have been proposed. For instance, sFlow [31], can be integrated with Ganglia to monitor VM clusters. Carvalho et al. [14] propose the use of passive checks by each physical host to notify a *Nagios Server* via a push notification mechanism about the virtual instances currently running on the system. Katsaros et al. [19] extend Nagios through the implementation of *NEB2REST*, which is a RESTful *Event Brokering* module utilized to provide elastic capabilities through an abstraction layer between monitoring agents and the management layer. Clayman et al. [11] propose *Lattice*, a scalable Cloud monitoring framework which monitors not only physical hosts but also virtual instances. *Lattice* can be utilized to monitor elastically adapting environments. In particular, the process of determining the existence of new VMs in an application deployment is required to be performed at the hypervisor level. A *Controller* is deployed as the responsible entity for retrieving a list of running virtual instances from the hypervisor, detecting if new VMs have been added or removed.

Wang et al. [34] propose *VScope*, a flexible monitoring and analysis middleware for troubleshooting multi-tier data intensive applications residing on Cloud infrastructures. *VScope* can operate on any set of nodes, software components or across different software levels. In contrast to *VScope* which targets data-intensive Cloud applications, Montes et al. [22] propose *GMonE*, a general-purpose Cloud monitoring tool applicable to all Cloud layers. *GMonE*, allows for monitoring instances to be deployed on any level of the Cloud and provides a pluggable model where users can inject their own custom metric collecting scripts to monitoring instances. Finally, Emeakaroha et al. [15] propose *LoM2HiS*, which attempts to bridge the gap between mapping low-level metrics collected from existing monitoring tools (Ganglia) to SLA parameters.

III. JCatascopia Design

In this section, we focus on introducing the key features that distinguish JCatascopia from the monitoring tools and academia approaches discussed in the previous section and provide a description of the proposed architecture.

A. JCatascopia Features

JCatascopia encapsulates in one Monitoring System the following key aspects:

- JCatascopia is **fully automated**, requiring for no human intervention after deployment. In contrast to

Lattice and the proposed Nagios extensions, JCatascopia supports elasticity without requiring any special entities deployed on physical nodes, nor does it request information from the hypervisor regarding the current running VMs in an application deployment. In Section 3.C, we present a mechanism based on a variation of the *publish and subscribe* message pattern [16] and *heartbeat monitoring* [18] to dynamically detect, at runtime, when elasticity actions occur and notify the Monitoring System accordingly.

- JCatascopia is both **platform independent** and **interoperable**. Thus, it is not limited to operate on specific Cloud Providers and can be utilized to monitor (i) federated Cloud environments where applications are deployed on VMs residing on multiple Cloud platforms and (ii) Cloud bursting environments [32] where applications deployed on a private Cloud *burst* into a public Cloud when resource demand increases.
- JCatascopia has been designed to provide **multi-level Cloud monitoring** and is capable of collecting heterogeneous types of information of different granularity across multiple levels (physical, VM, application level) of the Cloud. JCatascopia is both **customizable** and **extensible** and offers a complete API (discussed in Section 4.A) which assists application developers to implement their own custom metric collectors to report application specific performance metrics. Furthermore, JCatascopia is enhanced with a *metric subscription rule mechanism* where application developers and Cloud entities (e.g Resource Provisioner), can subscribe to aggregated metrics collected from any level of the Cloud and also compose high-level metrics from low-level metrics via a directive-based metric subscription rule language introduced in Section 3.E.
- Finally, JCatascopia is equipped with filtering capabilities to minimize both storage and communication overhead. In Section 3.D we present a technique towards ensuring that values will be filtered by adjusting the filter window range based on the percentage of previously filtered values.

B. Architecture

Figure 1 depicts an abstract view of JCatascopia's architecture. The architecture follows an agent-based *producer-consumer* approach. This approach provides a scalable, real-time, Cloud Monitoring System that can be utilized to collect monitoring metrics from multiple layers of the underlying infrastructure, as well as performance metrics from deployed applications. During the metrics collection process, JCatascopia takes into consideration the rapid changes that occur due to the enforcement of elasticity actions on the application execution environment and the Cloud infrastructure.

Monitoring Agents are light-weight monitoring instances deployable on any Cloud elements to be monitored, such as physical nodes or virtual instances. Monitoring Agents are the entities responsible for managing the metric collection process on the respective Cloud element.

Probes are metric collectors managed by Monitoring Agents. Probes are responsible for gathering low-level

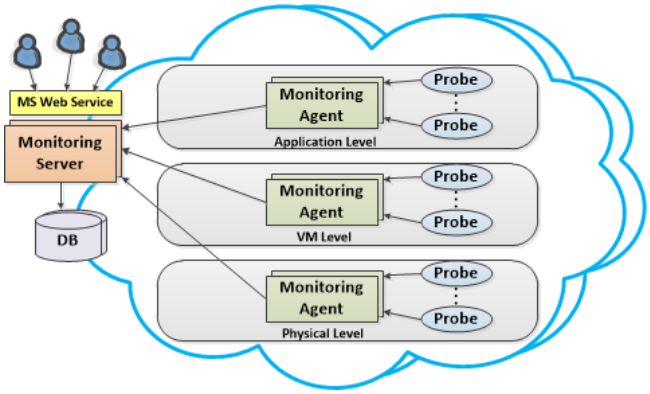

Fig. 1: Abstract Architecture View

monitoring metrics from the Cloud element they reside on, and performance metrics from deployed user applications. Instead of enforcing an Agent polling mechanism to collect metrics from Probes in fixed time intervals, Probes utilize either a push or pull mechanism to forward metrics to their corresponding Agent. This provides Probes with complete control on how to handle metric dissemination, by either reporting metrics periodically or upon the occurrence of a specific event.

An Agent aggregates and distributes processed metrics to **Monitoring Servers** that have expressed interest in receiving metrics from the specific Agent using as a delivery mechanism a slight variation of the traditional *publish and subscribe* (pub/sub) messaging paradigm. Employing a pub/sub mechanism minimizes related network communication overhead, since it avoids the need for the Monitoring Server to constantly poll Agents for new metrics. A Monitoring Server processes the received metrics forming *composite metrics*, upon consumer request (subscriptions), performing aggregation and grouping metrics from various instances together. Monitoring Servers are deployable on either physical nodes or virtual instances without requiring to reside in the same Cloud platform with their Monitoring Agents. In particular, since JCatascopia is interoperable, both Monitoring Agents and Servers may be distributed and operate across different Cloud platforms.

Using multiple Monitoring Servers is optional, however when utilized the topology gains in terms of scalability. Redirecting metric traffic originating from Monitoring Agents through intermediate Monitoring Servers offloads a central Monitoring Server from continuous information processing. This schema also provides fault-tolerance by eliminating single points of failure. This results in information resilience when Monitoring Servers are reported unavailable, by allowing metrics to be retrieved from healthy unaffected intermediate Monitoring Servers.

Finally, JCatascopia provides a RESTful **Web Service** that allows for external entities such as application users or decision-making *Resource Provisioners* to access monitoring information stored in the **Monitoring Database**.

C. Dynamic Monitoring Agent Discovery and Removal

Monitoring Agents must be both re-configurable and dynamically deployable at runtime, in order for a Monitoring System to support an automatic *Resource Provisioning*

System which elastically adapts application environments. Specifically, when a new VM is added to an application topology, a new Monitoring Agent must be configured and added to this VM. In turn, the Monitoring System must be notified for this addition. Similarly, the Monitoring System must also be aware when an Agent has been removed due to the removal of a previously allocated VM.

In the classic pub/sub message pattern, entities (subscribers) initially express interest and subscribe to an event stream of another entity (publisher). When events are produced, 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. In our Cloud Monitoring paradigm we differ from the classic pub/sub approach, since the static part (not affected by elasticity actions) of the model is the Monitoring Server, opposed to the Monitoring Agents which appear and disappear dynamically due to elasticity actions. In contrast to the classic pub/sub pattern (Fig. 2), we vary the message pattern as follows: (i) Monitoring Servers bind to a network address and port, awaiting for incoming requests and (ii) Monitoring Agents, which are considered metric publishers, initiate the subscription process by ping-ing the Monitoring Server of their existence, with a *request to connect* message. Afterwards, Monitoring Agents inform the Monitoring Server for the metrics they are responsible to collect and their metadata. Finally, after the connection phase, Monitoring Agents can start publishing metrics to the 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 re-configure the Monitoring System, (ii) to depend on the underlying hypervisor, and (iii) to require a directory service that contains these locations (required in the classic pub/sub pattern).

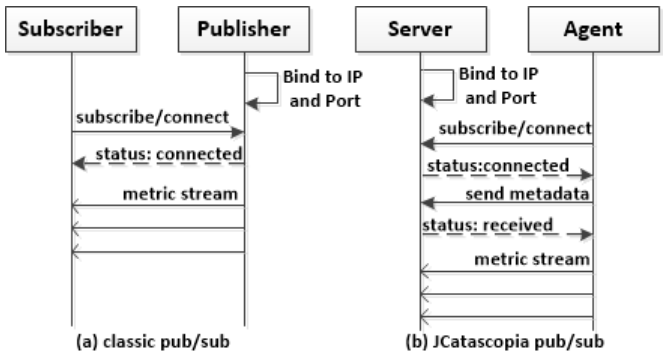

Fig. 2: Dynamic Agent Discovery Connection Phase

JCatascopia utilizes heartbeat monitoring [18] as an approach to automatically detect without the need of any external or human intervention when a Monitoring Agent: (i) ceases to exist due to scaling down elasticity actions, or (ii) is considered down due to temporary network connectivity issues which, inevitably, occur in large-scale distributed systems. A *Heartbeat Monitor* can be integrated in the Monitoring Server to check periodically, and report the status (Figure 3) of the connected Agents, setting their status to DOWN when finished. If an Agent fails to contact the Server until the next interval by either sending fresh metrics or a heartbeat (if no new metrics are collected) the status of the Agent will remain DOWN. If this occurs consecutively for a number of times (this value and the period is configurable) then the Agent is considered DEAD and is disconnected from the Server.

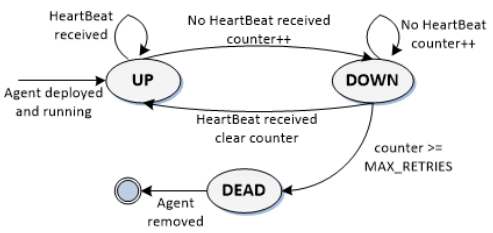

Fig. 3: Monitoring Agent State Diagram

JCatascopia also considers uncertainties imposed by application re-contextualization (e.g. dynamic IP configuration) due to VM migration which may affect the communication between Monitoring Agents and Servers. To address this issue, in each metric message, the Agent adds metadata (e.g. IP address) which are periodically updated. Even if the IP address of an Agent changes, the Server will be notified. The limitation of this approach is that a request (e.g. metric pull request) issued from a Monitoring Server to an Agent will fail if the IP address has not yet been updated. The error space can be shortened if the service issuing the migration (e.g. Resource Provisioner) informs the Monitoring Agent of this by triggering either the `updateAgentIP()` or `setAgentIP(ipAddress)` API calls. The same approach is followed in the situation where the IP address of a Monitoring Server is changed with the addition of one more step: After an IP address update occurs, the Monitoring Server notifies its respected Agents with a metadata message containing its new IP address.

D. Adaptive Filtering

Filtering is an essential feature of a Monitoring System, which assists in minimizing the communication and storage overhead, and consequently cost, by not transmitting or storing continuous values of a metric with very small variances between them. Utilizing a simple uniform fixed filter window (e.g. $previousValue \pm R$ where R is a fixed percentage range), for all metrics is only effective when the workload imposed to the monitored instance is considered stable. When consecutive metric values differ from one another, then there is no guarantee that any values will be filtered [12]. Taking this scenario into consideration we apply an *adaptive filter window range* approach where the window range depends on the percentage of values previously filtered.

For instance, let N be the number of values that will be used to determine if the window range R should change and let A, short for *aggressiveness*, be the target percentage of the values that ideally should be filtered. Initially, let R be set to $minR$ which is the minimum window range. For N consecutive metric values we use the filter window: $previousValue \pm R$. When N values have been collected, we compare the percentage of values,

filterValsPercent, that were filtered to the defined *aggressiveness*. If *filterValsPercent* is less than A then this is an indicator that we must consider a wider range since not enough values were filtered. We then set the new window range R to $R + step$, provided that $R \leq maxR$, where *step* is the length that we defined to make the window wider/narrower. If *filterValsPercent* is greater than A then we set R to $R - step$ provided that $R \geq minR$ since the filter window is too wide and we must consider a smaller window range.

E. Metric Subscription Rule Language

There are cases where one is 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 occurs when scaling an application component which is comprised of several VMs (e.g. database cluster). To address this issue we propose a **Metric Subscription Rule Language**, which can be utilized to: (i) aggregate and group low-level metrics originating from single instances, and (ii) 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 $Availability = 1 - downtime/uptime$.

We describe subscription rules as triplets with the following main elements:

`{Filter, Members, Action}`

The *Filter* is the part of the subscription 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 these Agents are specified in the *Members* part of the subscription rule. Figure 4 depicts the subscription rule language in BNF format.

```
<SubscriptionRule> ::= <Filter>, <Members>, <Action>

<Filter> ::= <MetricName> = <Expression> | <GroupFunction>(<Expression>)
<Expression> ::= <Operand> | <Operand> <Op> <Expression>
<Operand> ::= <Number> | <MetricName> | (<Expression>)
<Op> ::= +|-|*|/
<MetricName> ::= <String>
<GroupFunction> ::= AVG|SUM|MIN|MAX

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

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

Fig. 4: Subscription Rule in BNF

An exemplary Filter to create a new high-level metric, named *DBthroughput*, which calculates the average (*AVG*) throughput from the low-level metrics *read*, *write*, *insert* and *update operations per second* of a distributed database cluster is the following:

```
DBthroughput = AVG(rdps+wrtps+instps+updps)
```

When a filter is matched, the *Action* specified in the rule is enforced. Actions are either *time-based* (notified periodically) or *event-based* (notified when event 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)
```

Finally, a complete example of a subscription rule that calculates the average CPU usage from the low-level metric *cpuIdle* of a Web Server cluster comprised by N individual Web Servers is the following:

```
cpuTotalUsage = AVG(1 - cpuIdle)
   MEMBERS = [id1, ... ,idN]
    ACTION = NOTIFY(>=82%)
```

IV. Implementation

The following sections provide a detailed implementation description of the components that comprise the JCatascopia Monitoring System.

A. JCatascopia Monitoring Probe

JCatascopia provides the application developer with the ability to write custom Probes for collecting any metric that may be of interest and deploy them on Monitoring Agents. Probes are implemented using the *JCatascopia Probe API*[1] which provides an interface with the necessary abstractions for hiding the complexity of all the Probe functionality from the Developer. The *Probe Interface* is bundled in the *JCatascopia Probe jar* which allows users to import and use it when developing their applications. A Probe is not limited to collect only one metric; our implementation supports Probes that collect multiple metrics from the same resources in order to reduce the monitoring overhead. For instance, a Memory Probe can report *total*, *free* and *used* memory of a running VM, retrieving these values on a Linux OS distribution from different native fields available within the `/proc/meminfo` file.

In paricular, a Probe has the following capabilities: (i) enables the storage and querying for the last metrics reported and their timestamp; (ii) allows Probe Developers to add their own custom metric value checks (e.g. when collecting click statistics from a web server log file, do not report clicks from a specific IP); (iii) allows parameter configuration (e.g. configure collecting period); (iv) provides filtering; (v) can be dynamically deployed to a Monitoring Agent at runtime; (vi) distributes metrics at different time granularities, utilizing either a push or pull delivery mechanism.

Regarding filtering, JCatascopia introduces the idea of enhancing Probe functionality with the amount of logic in order to analyze, in place, the usefulness of collected raw metrics rather than distributing metrics over the network with little significance and finally performing filtering at the Server level. This enables JCatascopia to adapt the monitoring process based on the detected variance of the imposed workload. Users may enable adaptive filtering and

[1] A compete description of the JCatascpopia Probe API can be found at: http://www.celarcloud.eu/wp-content/uploads/2013/11/Cloud-Monitoring-Tool-V1.pdf

configure the filtering parameters (N, A, $minR$, $maxR$, $step$) for each Probe or metric separately by amending their default values in the Agent configuration file. Probe Developers can also assign through the Probe API default filtering parameters to a Probe to assist users which are not familiar with the filtering process.

Probes run independently from each other. If a Probe encounters a problem such as unexpected termination, the metric collection process of the other Probes is not affected. Probes are implemented separately from Monitoring Agents allowing them to be dynamically added/removed at runtime, without interfering with the monitoring process. This feature allows for a new Probe to be *plugged-in* to an Agent and configured at runtime. Furthermore, the pluggable functionality grants Agents with flexibility, which is useful when monitoring applications, by allowing the number and type of Probes corresponding to an Agent to vary.

As far as metric distribution is concerned, it is essential to gather metrics at different time granularities in order to enable the collection of heterogeneous types of metrics from various levels of the Cloud. For example, low-level metrics such as CPU, memory and disk I/O are usually required to be collected in shorter intervals (e.g. in the range of a few seconds) than high-level service metrics such as throughput, latency, and availability which are meaningful when collected in larger intervals.

B. JCatascopia Monitoring Agents

Monitoring Agents are responsible for processing and distributing monitoring information, originating from Probes, to Monitoring Server(s). An Agent is considered as the Probe Manager for the Probes deployed on the particular VM or physical node that the Agent resides on. A Monitoring Agent is responsible for (de-)activating Probes, configuring accordingly their parameters and upon user request pull different metrics. Figure 5 depicts the internal architecture of an Agent and its components.

Initially, when a new Agent is deployed, as part of the *Agent discovery process*, the *Server Connector* component pings the Monitoring Server requesting to connect. If the Monitoring Server responds, then the *Server Connector* transmits Agent metadata to the Monitoring Server, such as the Agent's ID and IP address and metadata of the metrics it will be collecting. Once the initial connection phase is over, a metric stream is created, allowing for monitoring metrics to be distributed to Monitoring Servers. We embrace the ZMQ framework [36] to implement the *JCatascopia pub/sub message distribution mechanism* which is built on top of the simplistic, yet powerful, ZMQ socket types. Thus, JCatascopia is able to control the message flow between Agents and Servers, adapting, if needed, to network transmission failures by rescheduling and resending messages. New metrics are added to the Agent *Metric Queue* either directly by Monitoring Probes or by the *Probe Controller* component which listens for metric requests from other processes. The *Probe Controller* also listens for probe parameter configuration requests (e.g. configure Probe collecting period) originating by either the Monitoring Server or from Application Users via the

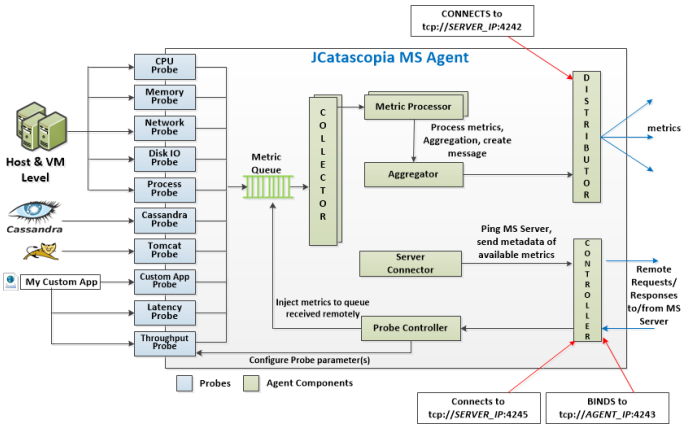

Fig. 5: JCatascopia Monitoring Agent

JCatascopia RESTful API. Metrics are dequeued by the *Metric Collectors* and processed by *Metric Processors*. Processing a metric refers to the task of 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 Agent metadata 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 the gathered metrics, these are given to the *Aggregator* which is responsible to withhold them from distribution until an aggregation policy is satisfied. The aggregation policy can be configured through the aforementioned Agent configuration file and can be *time-based* (e.g. aggregate and distribute metrics every 30sec); *volume-based* (e.g. aggregate and distribute metrics if message size exceeds 2KB) or can take into consideration both options. When aggregated metrics are ready to be sent, a message is created containing all the ready metrics and is subsequently sent to the Monitoring Server.

C. JCatascopia Monitoring Servers

Monitoring Servers are the entities responsible for managing Agents and storing monitoring metrics to the target database. Figure 6 depicts the internal architecture of a Monitoring Server and its components. The *Control Listener* and *Agent Listener* components are the entities that listen for incoming Agent connection requests and newly collected metrics, respectively. Connection requests by newly deployed Agents are handled by the *Control Listener* of the Server, which receives, parses and stores in suitable data structures (Agent and Metric Map) metadata describing the Agent and the collected metrics.

After the connection phase, an Agent *publishes* metric messages to the respected metric stream. The *AgentListener* component listens for incoming messages and enqueues them in 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

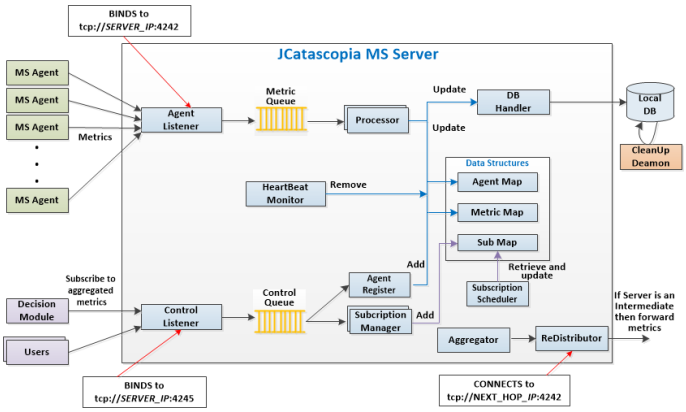

Fig. 6: JCatascopia Monitoring Server

structure stores metric metadata and their latest reported values. A *HeartBeat Monitor* is built into the Monitoring Server in order to detect when Agents have been removed due to elasticity actions, thus disconnecting them from the Monitoring System. If the Server acts as an Intermediate then after processing, aggregating and, if needed, filtering collected metrics it is ready for further distributing metrics to other Monitoring Server(s). If the Monitoring Server is not an intermediate, after a metric is processed it is stored by the *Database Handler* to the database.

The Subscription Mechanism, which uses the subscription rule language introduced in Section 3.D, is built into the Monitoring Server to allow users and internal platform entities (e.g. Billing entity, etc.) via the JCatascopia RESTful API to apply aggregation filters and grouping functions upon low-level monitoring metrics gathered by individual monitoring Agents to create new high-level metrics. The *Subscription Manager*, depicted in Figure 6, is the component that retrieves from the *Control Queue* and processes subscription requests, adding a new entry to the Subscription data structure and notifying the *Subscription Scheduler*. The *Subscription Scheduler* is the component that retrieves the Subscriptions from the respective data structure and updates their current value based on the minimum updating period specified by the user in the Subscription request.

D. JCatascopia MS Database

JCatascopia as a modular and extensible system offers its users with the functionality of implementing a database solution of their own, by providing a *Database Interface*, making JCatascopia flexible to which database is used. JCatascopia currently offers an implementation of a MySQL [23] interface. The selection of a relation database for the first version of JCatascopia was based upon providing the respected *Decision Entity* of an *Automatic Resource Provisioning System* with the ability to perform various types of complicated queries on monitoring data by allowing multiple joins on tables which cannot be facilitated by a NoSQL database. In order for query response time to not increase as the metric table grow, a *Cleanup Deamon* is used to extract old monitoring data from the database, process the data by filtering values that are not needed and performing aggregation functions on larger time intervals.

V. Evaluation

To validate the functionality and performance of JCatascopia, a testbed was created to establish that the proposed monitoring system is (i) platform independent, (ii) interoperable, (iii) scalable, (iv) able to provide dynamic agent discovery while scaling an application topology and (v) collect monitoring metrics from multiple levels of the Cloud infrastructure.

A. Testbed

For the purpose of our experiments we deployed and monitored on both public and private Cloud infrastructures the following representative Cloud applications:

- **Cassandra DB** [6]: a distributed storage system for managing very large amounts of structured data, stored as *key/value* pairs, spread out across many nodes. Each Cassandra node can be queried to obtain the read and/or write latency on a specified column family. The CPU utilization of the cluster nodes increases when Cassandra is stressed by a large workload. We take advantage of this factor to decide when a scaling action should be enforced.
- **YCSB** [13]: an open-source and extensible benchmarking framework developed by *Yahoo!* which can be utilized to generate realistic workloads, with the goal of facilitating performance comparisons of Cloud data serving systems (e.g. Cassandra). We have instrumented the YCSB Clients to expose latency and throughput of the targeted database over time which is hit by the generated workload. The workload imposed to the database cluster is *update heavy*, consisting of 50% reads and 50% writes.
- **HASCOP** [27]: a generic parameter-free attributed multi-graph clustering application based on a distributed algorithm. We instrument HASCOP to expose 5 metrics regarding the current number of clusters, the iteration elapsed time, and the time to update each of its data structures (graph weights, probability table and centroids). The iteration time decreases linearly with the number of processing units dedicated to the clustering algorithm.

To monitor the resource utilization of the VMs in our testbed and the performance of the applications that reside on them, we have developed using the *JCatascopia Java Probe API* several Probes. Table I presents the metrics each Probe is responsible for and the default collecting period for each metric. We selected to compare JCatascopia with two Monitoring Systems which follow a similar agent-based architecture: (i) Ganglia [20], which is an open source, production-ready, general purpose monitoring tool and (ii) Lattice [11], which is a monitoring framework that can be used to monitor elastically adaptive application environments and has a prototype available online. In order for the comparison to be meaningful we configured each Monitoring System to report the same metrics at the same frequency. Both, Ganglia and Lattice offer the CPU, memory and network metrics that JCatascopia offers as well. For the disk usage and application level metrics, we extended both Ganglia's metric library and Lattice, by implementing Python modules and Java Probes respectively.

Probe	Metrics	Period (s)
CPU	cpuUserUsage, cpuNiceUsage, cpuSystemUsage, cpuIdle, cpuIOWait	10
Memory	memTotal, memUsed, memFree, memCache, memSwapTotal, memSwapFree	15
Network	netPacketsIN, netPacketsOUT, netBytesIN, netBytesOUT	20
Disk Usage	diskTotal, diskFree, diskUsed	60
Disk IO	readkbps, writekbps, iotime	40
Cassandra	readLatency, writeLatency	20
YCSB	clientThroughput, clientLatency	10
HASCOP	clustersPerIter, iterElapTime, centroidUpdTime, pTableUpdTime, graphUpdTime	20

TABLE I: Available Probes

Our experiments were conducted utilizing VMs, of various sizes and operating systems, originating from 4 different Cloud infrastructures. Specifically, our testbed consists of the following:

- 15 VMs residing on GRNET Okeanos Public Cloud [26] with the following characteristics: 1 VCPU, 1GB RAM, 10GB Disk, Ubuntu Server 12.04.2 LTS. On 12 VMs we deployed Cassandra, while on the remaining 3 VMs we deployed YCSB Clients.
- 60 VMs residing on UCY Nephelae Private Cloud with the following characteristics: 2 VCPU, 2GB RAM, 10GB Disk, Ubuntu Server 12.04.2 LTS. HASCOP was deployed on all the VMs.
- 10 VMs residing on Flexiant FlexiScale platform [17] with the following characteristics: 2 VCPU, 2GB RAM, 10GB Disk, Debian 6.07 (Squeeze). On all 10 VMs we deployed HASCOP.
- 10 m1.small (1VCPU, 1.7GB RAM, 160GB Disk) Amazon EC2 [4] instances with CentOS 6.4. On all 10 VMs we deployed HASCOP.

On all the acquired virtual instances we have deployed JCastascopia Monitoring Agents, Ganglia gmonds and Lattice DataSources[2].

B. Runtime Impact Evaluation

In this section we evaluate the runtime impact of Monitoring Agents on *user paid* application VMs and compare JCatascopia to Ganglia and Lattice.

Experiment 1. Elastically Adapting Cassandra Cluster:

This experiment uses 15 VMs in Okeanos cluster. Initially the topology consists of 1 Monitoring Server, 3 YCSB VMs which are responsible for generating an increasing workload and the Cassandra cluster which at first, only consists of 2 VMs. The other 10 VMs are shutdown. For the YCSB VMs we only collect client-side related metrics by deploying a YCSB Probe. For the Cassandra nodes we collect VM level metrics by deploying a CPU, Memory, Network and DiskIO Probe and application level metrics by deploying a Cassandra Probe. To cope with the increasing workload, the Cassandra cluster must scale by *dynamically* adding more nodes to the cluster until all 12 VMs have been initialized. To calculate when a scaling action should be taken[3], we add a subscription rule to the Monitoring Server. The subscription notifies the Resource Provisoner[4] when the average CPU usage ($cpuTotal = 1 - cpuIdle$) of the Cassandra cluster is over 75% at which point we initialize another Cassandra VM and add it to the cluster. After each scaling action we add the id of the new VM to the subscription via a RESTful request to the Monitoring Server. The subscription rule used is the following:

```
cassCPUTotalUsage = AVG(1 - cpuIdle)
    MEMBERS = [id1, ... ,idN]
      ACTION = NOTIFY(>=75%)
```

Experiment 2. Monitoring a Cloud Federation Environment:

This experiment uses 30 VMs originating from the Amazon, Flexiant and Nephelae cluster to monitor a topology spread across muliple Cloud platforms. On each VM we have deployed HASCOP and utilized a CPU, Memory, Disk Usage and HASCOP Probe to monitor the topology for 5 hours. It must be noted that JCatascopia is out of the box *interoperable* whereas Lattice required configuration to make this feasible.

To evaluate the runtime footprint that JCatascopia, Ganglia and Lattice impose on all of the application VMs, for both experiments, we measure the average CPU and memory utilization for each Monitoring System. The results are depicted in Figures 7, 8 and 9. From these figures, we notice that JCatascopia's runtime footprint, both CPU and especially memory, is lower than Lattice.

In contrast to Lattice, Ganglia's memory footprint is lower than JCatascopia, though we must take into consideration that: (i) Ganglia is lightweight when utilizing its built-in default metrics but its runtime footprint increases (noticable in Fig. 8 and 9) when deploying user-developed Python modules which target application level metrics. (ii) Ganglia offers less functionality than JCatascopia. For instance, JCatascopia offers filtering and allows users to add custom metric checks which may impose a slight overhead when collecting metrics but gains much more by reducing the communication and storage overhead of the overall system. To justify this conclusion we measured the outgoing network utilization for the second experiment. From Figure 10 we observe that JCatascopia has inherently a smaller network footprint than Ganglia and when we enable adaptive filtering on the CPU, Memory and DiskUsage Probes with their default parameters set to $N = 15$, $A = 10\%$, $minR = 1\%$, $maxR = 3\%$ and $step = 1\%$, we observe that the network overhead drops even more (the large deviations in Fig. 10 are due to sent messages not having fixed lengths). Finally, to make our experimental results clearer, we used small VMs, whereas if slightly larger VMs were used the distinction between the two systems would be neglectable.

[2]The versions used were 3.1.7 for Ganglia and 0.6.4 for Lattice

[3]More advance decision making policies can be used which involve throughput, latency, etc. to decide when a scaling should occur, but this is not the target of this paper

[4]Our Resource Provisioner is a set of scripts responsible for enforcing elasticity actions and overseeing the experiments

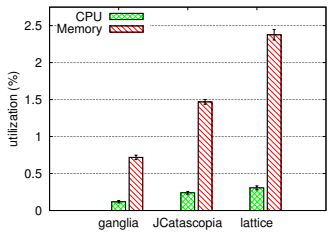

Fig. 7: Agent Utilization YCSB

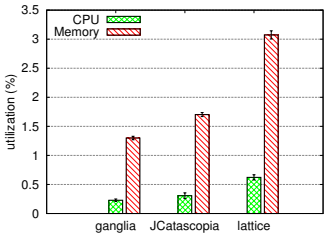

Fig. 8: Agent Utilization Cassandra

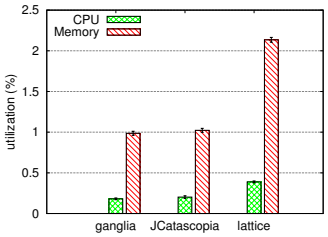

Fig. 9: Agent Utilization HASCOP

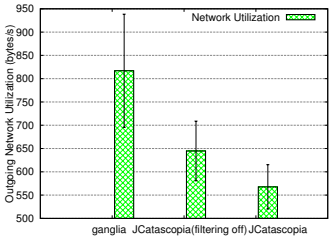

Fig. 10: Agent Network Utilization

C. Scalability Evaluation

In this section we study how JCatascopia scales while adding at runtime Monitoring Agents to an elastically adaptive application topology. The performance of a Monitoring Agent is not affected while scaling, since each Agent is an independent entity responsible of collecting metrics originated only from a single instance. This does not apply to the performance of a Monitoring Server which highly depends on the number of Agents in the topology and the amount of metrics it receives.

Experiment 3: Scaling a HASCOP cluster:

This experiment was performed on the 60 VMs which comprise the UCY Nephelae cluster. The goal of this experiment is to evaluate the scalability of a Monitoring Server while the number of Agents, and consequently the number of reported metrics increases. As previously stated, HASCOP and a JCatascopia Monitoring Agent were deployed on all 60 VMs. Each Agent reports to the Monitoring Server the same metrics (19 in total) as the second experiment (Monitoring a Cloud Federation Environment). We add a subscription rule to the Monitoring Server that reports the average elapsed time of each iteration:

```
hascopIterElapsedTime = AVG(iterElapTime)
        MEMBERS = [id1, ... ,idN]
          ACTION = NOTIFY(ALL)
```

Experiment 3a. Using a Single Monitoring Server:

The first scenario of this experiment is the following: Initially, the topology consists of 1 application VM and 1 Monitoring Server. The other 59 VMs are shutdown. The Monitoring Server was deployed on a VM with the same characteristics (2VCPU, 2GB RAM, Ubuntu Server 12.04.2) as the VMs in the cluster. Randomly, every 2 to 5 minutes, a VM was selected from the cluster and initialized automatically without the need to restart or reconfigure any part of the Monitoring System. This process is performed until all the VMs in the cluster have been initialized and are reporting metrics to the Monitoring Server.

To evaluate JCatascopia's performance while scaling, we measure *archiving time* which is the average time required for the Monitoring Server to process and store a received metric to the Monitoring Database. Figure 11 depicts the average archiving time observed when scaling the application topology from 1 virtual instance to 60. At first, for a small number of VMs archiving time can be considered as stable. As the number of VMs increases, we observe that JCatascopia achieves an archiving time that

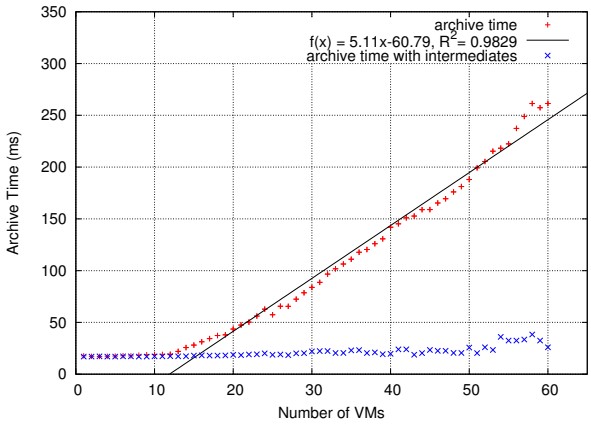

Fig. 11: Monitoring Server Average Archiving Time

grows linearly which is desired when scaling a distributed system. We apply a linear regression fit to the collected data and calculate the R^2 coefficient[5] to determine the accuracy of the regression which indicates a near perfect fit (0.9829). It should be noted that throughout the experiment, the highest observed CPU and Memory utilization of the Monitoring Server was 0.3% and 2.9% respectively.

Experiment 3b. Using 2 Monitoring Servers as Intermediates:

In the second scenario, we use an hierarchy of Monitoring Servers by adding 2 Monitoring Servers to the previous topology as Intermediates which are utilized to process, aggregate and distribute monitoring metrics originating from the underlying Agents to the root Monitoring Server. In turn, the root Monitoring Server will store the metrics in the Monitoring Database. As in the previous scenario the topology initially consists of 1 VM. Every 2 to 5 minutes a new VM is randomly selected, initialized and assigned to one of the Intermediate Servers (eventually each will be responsible for 30 Agents) until all 60 VMs are running and reporting metrics to their assigned Intermediate Server. Figure 11 depicts the average archiving time when using two Intermediate Monitoring Servers. From this figure we observe that archiving time is relatively stable, with the highest reported value to be 37ms. This is a significant performance gain compared to the previous scenario with uses one Monitoring Server. We conclude that when the average observed archiving time is considerably high we can redirect monitoring metric traffic through Indermediate Monitoring Servers, which will result in a significant performance gain, allowing the monitoring system to scale.

[5] the Coefficient of Determination, R^2, indicates how well data points fit a curve. Its values range from 0 to 1 with 1 indicating a perfect fit.

VI. Conclusion and Future Work

In this paper we have presented JCatascopia; a fully automated, multi-layer, interoperable Cloud Monitoring System. JCatascopia aims at supporting automated multi-grained Cloud platforms that offer elastic resource provisioning for deployed Cloud applications. Though still a prototype, JCatascopia has been successfully integrated in such a system [10]. Furthermore, we have presented the key features of JCatascopia: (i) dynamic agent discovery and removal to identify when monitoring instances have been added/removed due to elasticity actions, (ii) adaptive filtering to minimize both the storage and communication overhead by adapting the filter window range based on the percentage of values that were previously filtered, and (iii) a subscription rule language, which can be used to aggregate and group low-level metrics to generate high-level metrics dynamically at runtime. Experiments on public and private Cloud platforms show that JCatascopia is capable of supporting a fully automated Cloud resource provisioning system with proven interoperability, scalability and low runtime footprint. Finally, JCatascopia is able to adapt in a fully automatic manner when elasticity actions are enforced to an application deployment.

As future work, we will further pursue adaptive filtering and also enhance Probes with *adaptive sampling* to adjust the metric collecting period when stable phases are detected in the imposed workload. This results in minimizing the computation overhead. Furthermore, we will continue with the implementation of a Cassandra NoSQL database interface. Finally, we will enhance the subscription rule language and mechanism to enable JCatascopia to be integrated with a Cloud cost evaluation system [21].

VII. Acknowledgement

This work was partially supported by the European Commission in terms of the CELAR 317790 FP7 project (FP7-ICT-2011-8).

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] Amazon AutoScaling, http://aws.amazon.com/autoscaling/.

[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] Apache Cassandra, http://cassandra.apache.org/.

[7] 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.

[8] 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.

[9] AzureWatch, https://www.paraleap.com/AzureWatch.

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

[11] 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, pp. 239–246.

[12] S. Clayman, R. Clegg, L. Mamatas, G. Pavlou, and A. Galis, "Monitoring, aggregation and filtering for efficient management of virtual networks," in *Proceedings of the 7th Int. Conference on Network and Services Management*, 2011, pp. 234–240.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, USA: ACM, 2010, pp. 143–154.

[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. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar, "Low level metrics to high level slas - lom2his framework: Bridging the gap between monitored metrics and sla parameters in cloud environments," in *High Performance Computing and Simulation (HPCS)*, 2010, pp. 48–54.

[16] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

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

[18] N. Hayashibara, A. Cherif, and T. Katayama, "Failure detectors for large-scale distributed systems," in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 2002.

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

[20] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: Design, implementation and experience," *Parallel Computing*, vol. 30, p. 2004, 2003.

[21] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "Mela: Monitoring and analyzing elasticity of cloud services," *5th International Conference on Cloud Computing, CloudCom*, 2013.

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

[23] MySQL, http://www.mysql.com/.

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

[25] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, "Monalisa : A distributed monitoring service architecture," in *Proceedings of CHEP03, LaJolla, USA*, Jun. 2003.

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

[27] A. Papadopoulos, G. Pallis, and M. D. Dikaiakos, "Identifying clusters with attribute homogeneity and similar connectivity in information networks," *IEEE/WIC/ACM International Conference on Web Intelligence*, 2013.

[28] RackSpace CloudKick, http://www.rackspace.com/cloudkick/.

[29] Schubert and K. Jeffery, "Advances in clouds," *Report of the Cloud Computing Expert Working Group*, 2012.

[30] L. Schubert and K. Jeffery, "The future of cloud computing," *Report of the Cloud Computing Expert Working Group*, 2010.

[31] sFlow, http://www.sflow.org/.

[32] 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.

[33] 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.

[34] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt, "Vscope: middleware for troubleshooting time-sensitive data center applications," in *Proceedings of the 13th International Middleware Conference*. NY, USA: Springer-Verlag New York, Inc., 2012, pp. 121–141.

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

[36] ZMQ, http://zmq.org/.