Elastic Monitoring Framework for Cloud Infrastructures

Benjamin König[1], Jose M. Alcaraz Calero[1,2,*] , Johannes Kirschnick[1]

[1] Cloud and Security Lab

Hewlett-Packard Laboratories, United Kingdom.

{benjamin.koenig, jose.alcaraz-calero, johannes.kirschnick}@hp.com

[2] Department of Information and Communications Engineering

University of Murcia, Spain

jmalcaraz@um.es

Abstract

This paper presents a scalable and elastic distributed system for monitoring Cloud infrastructure based on a pure peer-to-peer architecture. Its distributed nature enables to deploy long-living queries across the network to monitor a diverse set of entities and metrics, spanning across all layers of a Cloud stack which can change rapidly. This allows for aggregating low-level metrics from operating systems, to higher level application-specific metrics derived from services, databases, or application log files. The observed metrics and information can be evaluated and used to reliably trigger policies to automate complex management tasks within a Cloud environment. The architecture incorporates a query framework for obtaining high-level information and a policy framework to provide self-management capabilities to monitored Cloud infrastructure. The system has been implemented as a proof of concept. Details and statistical results are provided to validate the scalability of the underlying architecture.

Keywords: Cloud Computing, Monitoring, Elastic Architecture, Network Measurement and Monitoring, Network Management, Distributed Processing,

* Corresponding Author

1. Introduction

Cloud Computing [1] is changing how businesses implement their services. Instead of using own dedicated infrastructure, there is a growing trend to utilize on-demand virtualized infrastructure, provided by a third-party and paid for in a pay-per-usage model [2] This paradigm allows for truly elastic services which change allocated infrastructure dynamically, based on changing business objectives and user demands. Cloud Computing also imposes several challenges, which have to be addressed in order to manage deployed services efficiently. These challenges, e.g. data security and privacy, infrastructure control and monitoring, and efficient user models are open issues nowadays.

A Cloud infrastructure is usually defined as a stack composed of three service layers [3]: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The IaaS layer uses physical processing and storage capabilities to provide on-demand virtual IT infrastructure in the form of virtual machines (VM). The *PaaS* layer relies on the underlying *IaaS* layer to provide different middleware services, such as security services and data processing services. Finally, the *SaaS* layer provides consumable services to the end user by utilizing the *IaaS* and *PaaS* layers to deliver these services efficiently.

Traditional monitoring solutions do not fit well within Cloud architectures. They are not designed to monitor both physical and virtual IT resources. They rarely are adaptive and cannot take advantage of additional computing resources themselves, or provide easy integration into the infrastructure to monitor elastic services. Usually, monitoring solutions for distributed environments are based on a client-server architecture in which information is collected by a variable number of clients and forwarded to a central master repository hosted at a dedicated server to be processed and managed at a later stage [4], [5]. This architecture can produce processing bottlenecks when the number of data sources and information gathered is being increased, for example, due to additional resources being created and added to a monitored service to provide elasticity. Also, client-server architectures suffer from a central point of failure, the server, which deems them unsuitable for Cloud infrastructures, in which monitoring is a key requirement for automatically scaling services. For the above reasons, providing a monitoring framework for a Cloud stack is challenging. This contributes to the fact that monitoring elastic and distributed layers, in which there are both physical and virtual resources with hundreds of heterogeneous data sources, is an open issue nowadays.

The main contribution of this paper is to provide a suitable architecture for monitoring Cloud infrastructures. This architecture has been implemented as a fully pure distributed, peer-to-peer (P2P) system which provides reliable and elastic monitoring services for gathering information across all layers of a Cloud stack.

2. Related Work

Several distributed monitoring solutions have appeared during the last few years. Zanikolas and Sakellairou [6] provide a complete taxonomy for monitoring Grid architectures. They differentiate four levels of distributed monitoring solutions ranging from level 0, being monolithic monitoring software, up to level 3, which is highly flexible in which all involved components can be deployed in a distributed environment. Monitoring software uses sensors or data publishers to collect and originate information. Further, data processors provide aggregation and evaluation of monitored data, and consumers are receivers of the processed information. Level 3 is associated to monitoring software in which all these components are designed modularly and can be deployed distributedly.

Nowadays, several level 3 monitoring solutions are available. For example, Nagios [4] is a well-known, enterprise-class monitoring and alerting solution that provides organizations extended insight into their IT infrastructure and helps to identify imminent problems which could affect critical business processes. Ganglia [5] is a scalable distributed monitoring system for high-performance computing systems based on a hierarchical design targeted at groups of clusters. Both solutions are widely used and are designed using a central repository for later queries over aggregated information.

An alternative to centralized evaluation is the distributed execution of queries across the network, avoiding the necessity of a central repository. This is the approach followed in this paper. In this sense, R-GMA [7] is a distributed monitoring framework for Grid architectures which provides the concept of *virtual table* for querying aggregated information. These virtual tables enable complex queries over aggregated data which will be distributed across the network and evaluated in parallel on-line. Although R-GMA does not have a central repository, it still has a central point of failure as it requires a dedicated central registry which manages the location of data input for rendering virtual tables.

MDS4 [8] is the monitoring system implemented in Globus Toolkit 4. It is a hybrid solution with a central repository for monitored information, but also has the capability to distribute aggregators across the network to provide basic data processing over information flowing between data sources and consumers.

These solutions all have a central point of failure which is potentially susceptible of denial-of-service attacks. To address this issue, MonALISA [9] provides a multi-agent based approach without central point of failure, powerful monitoring capabilities, control and global optimization services for complex systems. However, this solution lacks on providing efficient query distribution services over monitored data and it does not provide any policy-based framework to manage the monitored systems.

Cloud infrastructure is elastic by nature. Due to scaling infrastructure, the amount of data having to be monitored changes rapidly. Moreover, monitoring becomes a critical service in Cloud Computing due to the strong control requirements imposed by users, which is one of the main challenges in the field, nowadays. Thus, dedicated points of failure are not acceptable. Users, administrators, and operations staff require monitoring services which enable advanced correlation among the monitored data. This eases debugging of the whole Cloud infrastructure as well as Cloud services, and gives complete insight and control. Self-management is another requirement associated to Cloud infrastructures, which requires high-level policy rules to be applied over the infrastructure, services and monitored data. None of the above mentioned monitoring solutions fulfil these requirements. The main contribution of this paper is to provide a flexible monitoring framework which satisfies such requirements providing a step forward in the monitoring of the cloud stack. The architecture makes suitable the extension of current cloud-enabled solutions with monitoring and basic self-scaling capabilities. For example, our architecture can act as PaaS for the cloud-based online video playing system provided by Jiang et al [10], the cloud architecture for multimedia traffic in the Internet of Things provided by Zhou and Chao [11] and the architecture for enabling cloud-based IPTV provided by Lai et al [12] providing automatic scale up capabilities according to monitored bandwidth parameters. Other example is the Cloud-based recommendation system provided by Lai et al [13] which require intensive data processing. This system can automatically scale up according the CPU information provided by our monitoring system or can use our architecture to use metrics to determine the recommendations. Our monitoring architecture can also acts as internal cross-layer IaaS service providing correlated data acquired among the entire cloud stack. For

example, fast agreements between customers can be achieved by using the monitored data, extending the approach provided by Wang et al [14].

3. System Architecture

Figure 1 shows an overview of the proposed monitoring architecture. The architecture is composed of three individual layers, *Data*, *Processing,* and *Distribution* layers*.,* These layers are interfacing on different levels with the Cloud stack.

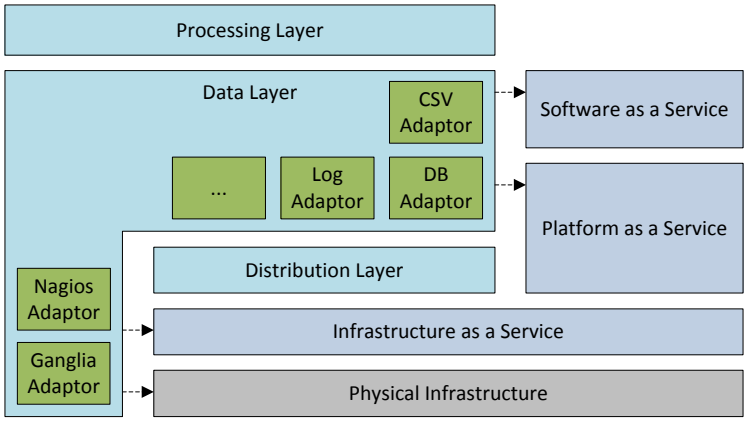


Figure 1: Layers of the System Architecture

The *Data Layer* provides an extensible package of adaptors used to integrate heterogeneous data from different sources. These diverse adaptors are spread across the Cloud infrastructure and across all layers of the Cloud stack. Common examples are: log files, adaptors for software components like databases or web servers, adaptors for other monitoring solutions (Nagios, Ganglia, etc.), and for protocols like *SNMP and NetConf.* Adaptors are in charge of retrieving and homogenising information from the source in order to be used within the monitoring system. Note that our architecture complements current monitoring solutions making them suitable for an application within Cloud environments.

The Processing Layer exposes a powerful and concise language for describing complex queries over data integrated by the *Data Layer*. *The Processing Layer* also enables the definition of policy rules which are triggered when pre-defined objectives are fulfilled, creating a policy-based management system. Queries are expressed in a SQL-like syntax and parsed generating a tree structure of data processing operators, which determines the evaluation order of such operators necessary to monitor the results. Operators in the query can be adaptors reading from data sources or higher level data aggregator operations, such as joins or filters.

The operators are software elements which process data hierarchically from their child nodes and forward processed results upwards in the hierarchy to the parent operators up to a consumer. The key is that each operator in the tree structure of a query is deployed and executed in a distributed way across the Cloud network environment being able to retrieve data remotely from distributed sources (if necessary). Note that in this approach for distributing operators, data locality can be exploited since operators can be placed close to the actual monitored data. Operators can be deployed in any computation resource along the Cloud infrastructure enabling distributed data processing without requiring a central repository of data.

The *Distribution Layer* is in charge of performing an automated deployment of operators in the correct place across the distributed Cloud environment. For this task, this architecture relies on existing automated deployment services such as *SmartFrog* [15], SLIM [16], or Puppet [17]**.** Although the proposed monitoring architecture does not have a direct dependency on a specific deployment solution, the design requires a pure peer-to-peer deployment solution (e.g. *SmartFrog*) in order to avoid any point of failure during the deployment phase.

SmartFrog creates an unstructured peer-to-peer overlay network across physical and virtual machines within a Cloud infrastructure. Each machine runs a SmartFrog daemon which serves as an entry point for the management of this computational resource. SmartFrog daemons facilitate the installation, configuration, execution and overall life cycle management of distributed applications and services deployed into these resources. The desired configuration of a distributed application is expressed in a *Component Description Language* (CDL). Each query in the monitoring architecture generates a tree structure of operators, which in turn, produce a set of component descriptions (one per operators) in CDL language necessary to deploy and execute distributedly the query. In case the reader is interested in the use of SmartFrog for deploying services within a Cloud stack, Kirschnick et al. [18] provide a comprehensible description. The *Distribution Layer* has been designed to provide elasticity to the monitoring architecture. It has to deploy adaptors and operators, and can decide on a suitable deployment destination to make use of Cloud resources efficiently, optimizing for data throughput or minimal network consumption.

The architecture enables users to monitor distributed and heterogeneous data sources integrated into a coherent model by means of adaptors. It is the basis of a fully distributed and scalable framework for monitoring Cloud infrastructures without central point of failure, neither in the deployment phrase nor in the

execution phrase of the query. Monitoring reliability is determined by the fault tolerance of the VMs used for deploying and executing the queries (feature provided by the Cloud provider). Moreover, the monitoring architecture is elastic by nature since it allows for occupying additional VMs for monitoring purposes, and also scales when new virtual resources or data sources become available in the Cloud infrastructure. This is a clear differentiating point with respect to current monitoring solutions which is necessary in the design of a suitable monitoring platform for cloud computing.

4. Monitoring Workflow

Three sequential steps are associated to the execution of the monitoring framework: *Metadata Definition, Query and Policy Definition, and Distribution (and Execution)*. Figure 2 shows an overview of this process and the required artefacts associated with each step. Note that steps are labelled with step numbers.

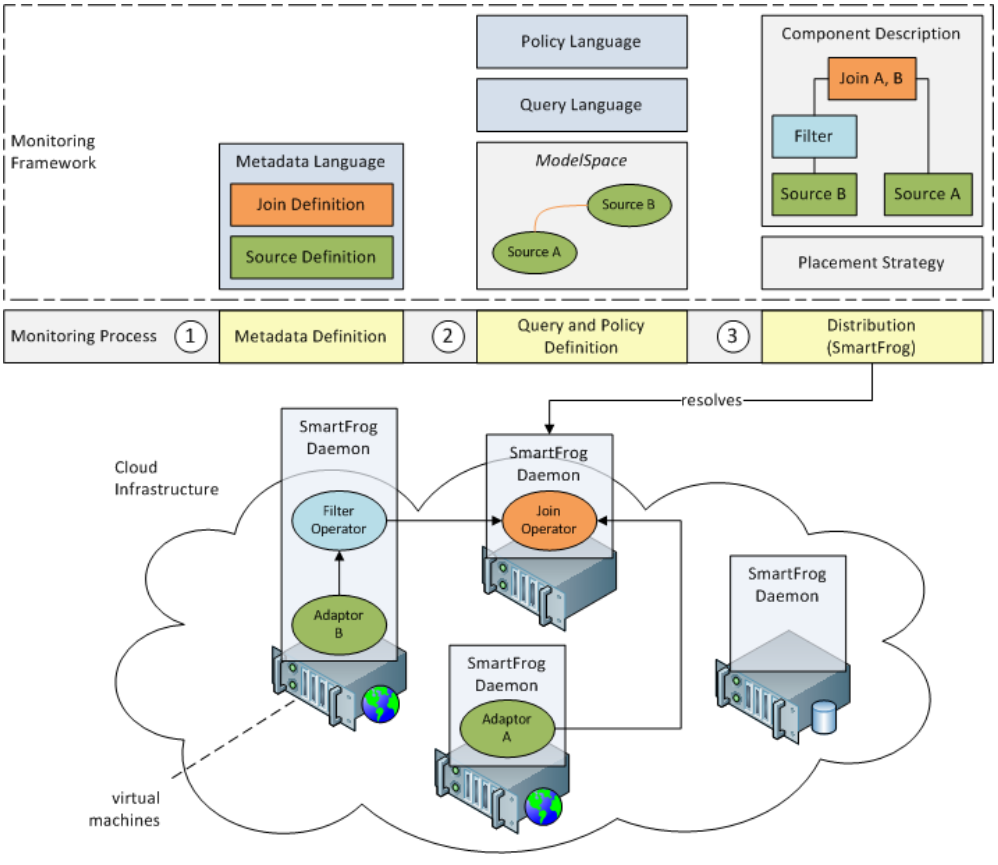


Figure 2: Application of the Monitoring Framework

First, a metadata language is used to define the data sources available in the Cloud infrastructure (step 1). Then, a query and policy language is used to evaluate integrated information over these data sources and

to define system reactions over certain conditions (step 2). Finally, the queries and policies are deployed using a placement strategy across the Cloud infrastructure starting the monitoring life-cycle (step 3).

Let us introduce a scenario used to clarify the different steps involved in the monitoring process explained in the following subsections. An enterprise is running a Web portal with several Web services in the *SaaS* layer. *Customers* information for such Web services is managed in the *PaaS* layer by the Cloud provider using a database service. Moreover, all *Services*, including Web servers and load balancers, are deployed in a variable number of VMs in an elastic way, provided by the *IaaS* layer. The status of all physical and virtualised resources is being monitoring using the proposed monitoring system.

4.1. Metadata Definition

A *metadata language* enables the description of the different data sources available in the Cloud infrastructure. Each data source defined in the *metadata language* is associated to a specific adaptor (software element) for carrying out data retrieval and data homogenisation at run-time stage. Adaptors can be implemented using either pull or push or periodic methods to retrieve data from a local or from a remote source, and convert this data into a common data structure. Adaptors are configured with the information provided in the *metadata language* and they are self-explaining as they provide a description of the monitored data.

Further, this *metadata language* allows for organising data sources hierarchically by defining high-level data sources (or views) composed of an aggregation of basic data sources. These views adapt dynamically according to the current state of the Cloud environment and thus additional data sources can be integrated automatically. In essence, the high-level data sources register new basic data sources when they become available due to, for example, a scaling up of the virtual infrastructure.

Metadata definitions can be represented as a coherent graph-based model of data sources referred to as ModelSpace. The ModelSpace is a read-only view of data sources within a Cloud environment. Within a ModelSpace graph, nodes refer to data sources, while links represent data relationships. By using adaptors which abstract from heterogeneity and data retrieval, diverse data sources may be integrated. The links join data sources and enable navigation between them.

Figure 3 shows a ModelSpace representation of the data sources in our running example: *customers*, *services*, and *hosts*. *Customers* data source is retrieved from the database used in the *PaaS layer*, *service* information is retrieved from log files used by deployed applications (Web servers, load balancer, enterprise applications), and *host* information is retrieved directly from associated VMs and physical machines. It is assumed that *hosts* is a high-level view, so additional hosts can be integrated into the ModelSpace automatically. This example also assumes that each VM and physical machine is running an OS level monitoring solution, such as Ganglia, to retrieve operating system metrics (i.e. CPU, disk, or memory utilisation).

Figure 3: ModelSpace Representation for the Example Scenario

Rather than creating a dedicated language with its own syntax and semantics for describing the *metadata language*, the language proposed relies on *Groovy* [19]. *Groovy* is a dynamic programming language for the Java VM which facilitates the creation of Domain Specific Languages (DSL). Koenig et al. [19] provide a comprehensible explanation. The *metadata language* uses a subset of Groovy and shares its syntax and semantics. The *metadata language* is used to render the *ModelSpace.* For each data source a specific adaptor is instantiated and configured in the *metadata*. For instance in our running example, an adaptor for a database system for the *customers* data source requires information about the concrete implementation class for such adaptor, the database table and columns to be monitored, the credentials used to connect to the database, and the endpoint IP/port in case of remote monitoring. This information later ensures the correct execution of the adaptor and ensures that information is retrieved in a homogeneous way, using a table structure as a homogenization data unit where columns represent the type of monitored information and rows represents samples of monitored values. The navigation relationship between data of different data sources is also specified in the *metadata* by mean of *joins*. A *join* creates a link in the *ModelSpace* and enabling navigation between data. A *join* description is composed by “left” and “right” attributes used to establish the joining condition between the data sources involved.

4.2. Query and Policy Definition

The ModelSpace acts as abstracting layer providing the underlying schema for the *query language*, which has been developed to retrieve data from data sources, filter and evaluate retrieved data, and join data from different data sources following an SQL fashion.

The *query language* uses a dot notation [20] to express queries, chaining references to data sources and performing evaluations along the way. A query is composed by a set of two different kinds of operators: *sources* and *features*. A *source* is a reference to a data source defined in the ModelSpace. It may be followed by (“.” and) either another *source* or a *feature*. A *feature* is a reference to an *operator*, which provides capabilities for evaluating and processing data. The framework provides different *feature* operators, e.g. selections, projections, filter for duplicated and empty data, aggregations, and aggregations over time. Further, each “.” within a query is translated into a *join operator*, which joins sets of data received from two child operators by evaluating a join predicate. Note that this join predicate is already defined in the *metadata*. Thus, the semantics for a given query is a straight-forward evaluation of the ModelSpace where *sources* represent data sources, *“.”* represent joins between data sources, and each *feature* represents data processing actions over the previously obtained results.

For example, `customers.eq(level:gold).services` monitors all running Cloud services associated to “gold” customers in our running example. Note that the query does not specify where and how the query operators are evaluated. This information is determined by the *Distribution Layer*. Note as well how cross layer monitoring is achieved since both PaaS and *SaaS* information is being monitored and combined in this example.

Different *feature operators* are supported to address a wide set of use cases for data processing. These features give extended insight into monitored data sources as they provide elaborated information rather than mere monitored data. For example, the *project* and *select features* are inspired by SQL. They isolate a subset of data columns or data rows respectively, according to a given criteria. Several basic evaluators are provided for defining criteria such as *equals*, *not-equals*, *less-than*, *etc*. Other evaluators have been also provided for filtering duplicated data and removing *null* values. Regarding aggregation *features* over the monitored data, a *groupBy feature* can be applied to group rows with similar values applying an aggregation. A more complex aggregation is done using *time aggregation feature* in which the information to be aggregated is previously

filtered (*select* operation) in order to isolate a time window of the monitored data and then the results are aggregated in the same way as *groupBy* operator. After an *aggregation*, a query usually requires an aggregation function to process the aggregated information according to a given criteria. Different aggregation functions have been included to enable the calculation of the average, minimum, maximum of aggregated rows, as well as the sum of the values. This set of *features* enable the monitoring and processing of the monitored information providing to the end-user a high level overview and information of the current status of the system.

Users may also define a policy which is attached to a query in a post-fix notation. Policies are translated into *policy operators* which execute user-defined blocks of code for data received from query operators. They can be employed to automate the handling of query results and error messages, and ease the automation of complex management tasks.

In essence, a policy defines actions which are executed when given criteria are fulfilled. Thus, they implement policy-driven system adaptation. All actions are expressed using Groovy/Java code and may be triggered by the underlying query. A policy can define up to four code blocks: *added*, *removed*, *done*, and *error*. The *added* code block is called for new query results, *removed* is called for out-dated results, and the *done* block is called when all query results have been retrieved. *Error* code block can be used to determine fault locations and diagnosis. Note that a query is executed in a long-living fashion and these code blocks control the life-cycle of a running query.

Listing 2 shows a query with an additional policy attached. The query calculates the average CPU load each service of a “gold” customer produces over a period of 10 minutes. This policy triggers a service scale up if the average CPU load exceeds 70%. Likewise, the policy also scales the running service down again, once user demand descends and CPU load falls below 20%. Note that the code used for scaling a Cloud service depends on the particular Cloud infrastructure in which the monitoring framework is applied. In this example, the *IaaSAPI* utility class has been used for isolating the vendor specific invocation methods.

```
1   customers.eq(level:gold)
2       .services.hosts
3           .select(cpu_load, service_id).minutes(10).avg(cpu_load) (
4       added: { /* code for scaling infrastructure */
5           if (cpu_load > 70) {
6               IaaSAPI.createServiceVM(service_id)
7           }
8           if (cpu_load < 20 &&
9               IaaSAPI.vms(service_id) > IaaSAPI.minVM(service_id) {
10              IaaSAPI.removeVM(service_id)
11          }
12      },
13      error: { /* code for managing errors */ }
14  )
```

Listing 2: Example Query and Policy for Scaling a Cloud Service

4.3. Query Distribution and Deployment

After queries and policies have been defined, they are parsed generating an operator tree, similar to *query execution plans* [21] found in relational database systems. The operator tree is constructed as a straight forward left-to-right evaluation of the query. For instance, Figure 4 illustrates the tree associated to the example query `customers.eq(level:gold).services`.

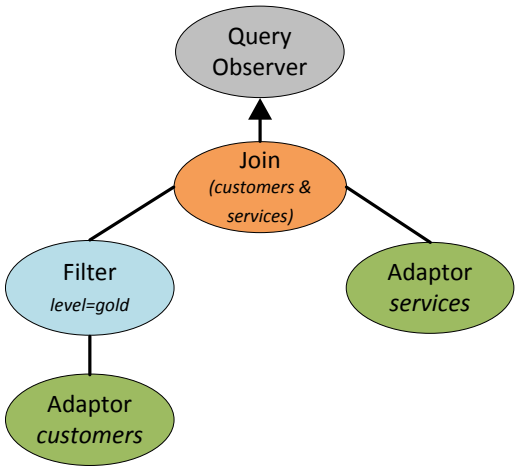


Figure 4: Tree Representation for the Example Query

Note that the *join* operator receives monitored information from both child branches, executes the join functionality and passes joined data on to an optional *query observer* or *policy operator* depending on the scenario. The *query observer* node represents a component that receives up-to-date monitoring information. In a production scenario, this node can be a model of a graphical interface or any other external software to receive updates from a deployed query.

The *Distribution Layer* regards operator trees as distributed applications and deploys them within the Cloud environment. This service is in charge of locating, deploying, executing and orchestrating all query operators in the network. Using *SmartFrog* for deployment, a tree of *SmartFrog* component descriptions is created first. Each component description contained in such tree represents a different operator associated to the query. The Distribution Layer then determines the place (e.g. virtual or physical machine) where processing pieces have to be installed and executed. This location can be inferred from information provided in the metadata, or determined automatically based-on deployment strategies if such information is not explicitly defined. While *source* operators can adapt a simply strategy of being deployed into the same virtual or physical machine that also provides the monitored data (data source), the location for other *processing operators* may be determined by applying different strategies. The deployment strategies can be used in combination with the *Distribution Layer* through a plug-in architecture, enabling the utilization of different algorithms.

Without considering network traffic, an efficient placement strategy is a simple algorithm which chooses its deployment destinations for query processing operators based on the current workload of virtual or physical machines. Note that information about CPU load may be provided by the monitoring system itself and this information can be used as part of the deployment policies. This algorithm can reuse existing VMs as a destination, but may also request additional VMs or destroy existent ones according to an elasticity limitation parameter for controlling the trade-off between cost associated to the usage of additional resources and the process time associated to the monitoring task.

Listing 3 shows a fragment of the *SmartFrog* component descriptions for the example query. These component descriptions contain all required configuration to ensure correct run-time behaviour of operators. In this case, necessary information for instantiating the correct operator/adaptor (*sfClass*), data filtering (*attributes* attribute), and information about where has to be deployed and executed (*sfProcessHost*). Note how this information has been previously provided by the different layers of the monitoring architecture.

```
1   eqOperator {
2       sfClass “mon.operators.feature.Eq” // operator class to instantiate
3       sfProcessHost “vm2.cloudiaas.hp.com” // deployment destination
4       attributes [ level : “gold” ] // level has to be “gold”
5       source customersOperator {
6           sfClass “mon.operators.source.SQLMonitor” // adaptor implementation
7           sfProcessHost ... // deployment destination for operator
8           ... // configuration for data retrieval (e.g. DB table and credentials)
9       }
10  }
```

Listing 3: Except of the SmartFrog Component Description for the Example Query

Once all component descriptions have been generated, they are sent to a running *SmartFrog* daemon in the Cloud environment. This initial SmartFrog daemon is in charge of distributing these component descriptions among all the other involved daemons – each component description is sent to its specified location (using *sfProcessHost* values). Each daemon may receive component descriptions for different operators and has to install the necessary software artefacts. *SmartFrog* not only provides a framework for automated provisioning of services but also ensures an orchestrated deployment and full life-cycle management. This enables an orchestrated instantiation of the different query operators which matches the evaluation order associated to the query operators. This orchestration ensures correct data processing, enables parallel processing when possible, controlling synchronization points in which operators require information provided by other operators to process data.

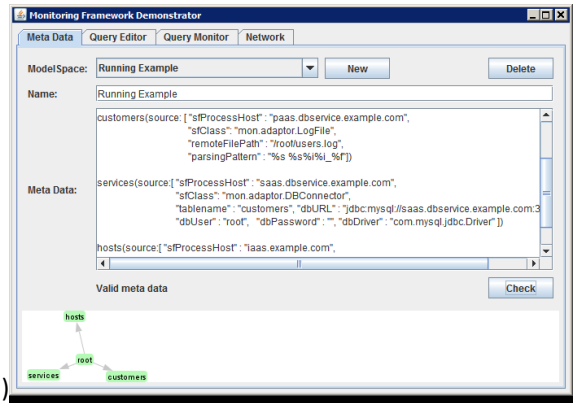
Due to their application for monitoring purposes, queries are long-living and continuously evaluated to supply *query observers* and *policies* with up-to-date monitored information automatically. Query evaluation is triggered by source operators when data changes at its source. Source operators translate any retrieved data and push it to their parent. This established a data flow within the deployed tree of operators, as each operator evaluates received data and forwards the evaluation result. In order to restrict the amount of data sent between query operators, only result changes are distributed. As a result, inner operators in the operator tree (e.g. feature, join, and policy operators) need to evaluate data only when necessary.

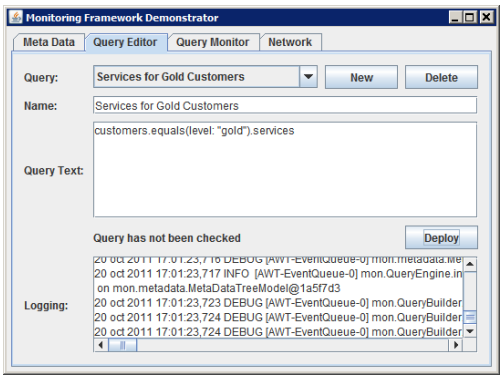
This *push-based* query evaluation produces very little overhead compared to a top-down evaluation, which is implemented by the traditional *iterator* data type found in modern database systems [21].

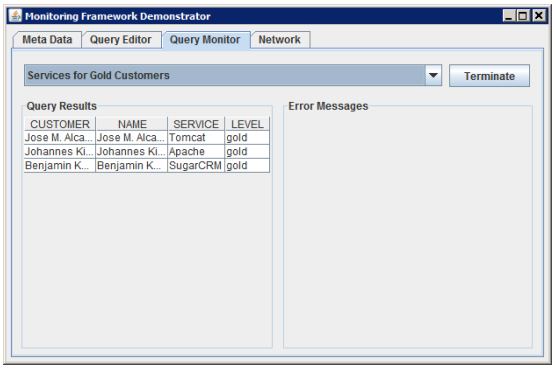
5. Implementation

The architecture proposed has been implemented as a proof-of-concept system and is being used for monitoring our internal test bed Cloud infrastructure. The prototype has been developed in Groovy and Java. Several adaptors have been implemented to retrieve information from a variety of sources such as log files, JDBC sources, e.g. Oracle and MySQL, and monitoring solutions, e.g. Nagios and Ganglia. Both adaptors and operators share the same interface. This interface allows for an easy extension of the framework with additional adaptors and data processing features. It exposes all the methods related to the life-cycle management, distributed deployment, query observer registration (also used for policies) and data structure description.

A graphical interface has been implemented allows users to specify metadata, enter queries and policies and receive results from submitted queries, everything at run-time stage. This graphical interface is designed as 4 different tabs enabling the execution of all the processes explained in previous sections. Figure 5 shows a series of screenshots of the different tabs for the executing of the running example. In particular, Figure 5.a shows the metadata language for describing the three data sources used in the running example. Note how *hosts* and *services* are remotely monitored using a database adaptor whereas *customers* is retrieved from a parsed log file. The example query *customers.eq(level : gold)*.services is shown in Figure 5.b. Then, when "deploy" button is pressed, the query is deployed using a selectable placement strategy and the run-time monitoring monitor is shown with up-to-date information (see Figure 5.c which shows the current results of such query). For this example, three users are gold users and it is shown the associated services to such users. Finally, Figure 5.d illustrates the deployed query tree in order to show where the different operators are physically deployed and executing at run-time stage.

a) 

b) 

c) 

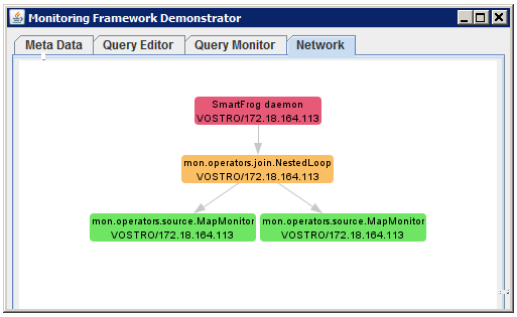
d) 

Figure 5: Screenshots of the Graphical Interface of Monitoring Framework Demonstrator

The graphical interface also enables to select different placement strategies for query deployment. Several strategies have been implemented. The last screenshot shows an application of a *mock* strategy used for testing purposes, which allocates every operator onto the local machine. Further, a *cloud-based* strategy which places operators onto on-demand virtual IT resources has been developed. These resources can be pre-existing or dynamically created according to observed parameters such as: query priority, maximum and minimum number of VMs, response time of monitoring framework, to name but a few. The latter strategy can be used to request for new VMs via an existing *IaaS API*, which enables the elasticity of the monitoring architecture.

Among others, an adaptor for *Ganglia*, a monitoring software, has been implemented which is used to gather operating system metrics from both the virtual and physical machines in our landscape. This adaptor enables to correlate between virtual and physical workloads. In fact, the adaptor has been extended with plug-ins to be able to retrieve additional application-specific metrics where needed. Moreover, log and database adaptors have been created to retrieve information available in the *IaaS* and *PaaS* layers related to user accounting, security, Service Level Agreements, among others. Finally, the prototype system can also use as a platform service itself which enables customers of the Cloud infrastructure to use the implemented monitoring architecture to retrieve a subset of the monitored information related to their deployed virtual infrastructure. It enables Cloud providers to control what information is exposed to cloud users whereas give control to users to monitor their infrastructure. This is another differentiating key with respect to other monitoring solutions.

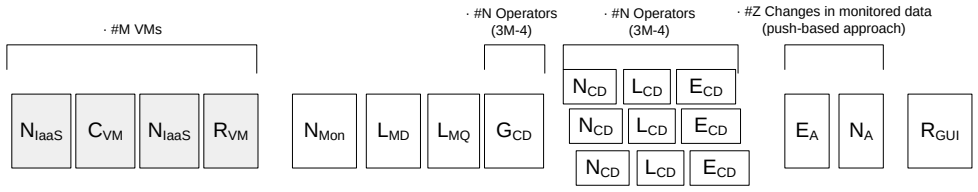
6. Statistics

A test environment has been established to validate the proposed architecture. Within a dedicated, internal Cloud test bed, a query which retrieves information from VMs running Ganglia has been executed in various constellations. Data retrieved from Ganglia has been consolidated and logged to a text file by a query observer for further analysis.

To analyse the scalability of the proposed architecture, the same query was repeatedly executed while varying the number of monitored data sources. Each test run creates a different number of VMs running Ganglia in the IaaS and monitors the VMs. VM 0 has been reserved to initialise query deployment and aggregate data delivered periodically by Ganglia adaptors running on VM 1 to 17. This allows to record the load different customers create on VMs and act accordingly. This scenario provides an overview about how the proposed architecture deals with the elasticity of the Cloud environment. The Cloud architecture is composed of 6 homogeneous physical Dual-Core servers running, based on XEN virtualization which exposes an IaaS API used to create dynamic computing resources.

This scenario contains up to 17 Ganglia adaptors for virtual resources (1 per each VM). Each test run following the analytical model shown in Figure 6 where all the times involved in the testbed are carefully indicated. T_0 is the time required to create and execute the virtual infrastructure in the IaaS and to run the operating system and the services available therein. This time is out of scope of the proposed architecture due to it can be assumed that services to be monitored are already running and only is shown for completeness. T_0 assumes a sequential creation of the infrastructure however parallel requests can reduce this time significantly, this parallel model is analytically approximated using the notation $\approx_p$. T_1 is the time for creating the component descriptions. Note how it directly depends on the number of operators (N) available in the query which in turn directly depends on the number of VMs in the testbed. In particular, the query used in the testbed is a tree composed of *3n-1* operators where *n* is the number of data sources available (matches with M-1, where M is the number of *VM*, thus it can be defined as *3M-4*). T_2 is the time for deploying all Ganglia adaptors (source operators) and other processing operators. It includes the distribution, remote installation, configuration and execution of all the software component description at the destination. Note that *SmartFrog* support efficient parallel deployment and thus the sequential deployment can be significantly reduced as shown in the parallel model analytically approximated. Finally, the time executing the query over the data

sources, yielding the results to the graphical interface is modelled analytically in T_3. It depends on the number of data changes in monitored data sources (Z) (all are considered changes at the first time), and the level of the operator in the tree in which such changes occur (L_i).



N_{IaaS} – Network Delay to invoke IaaS method
C_{VM} – Time for creating VM in the IaaS
R_{VM} – Time for booting/running VM and its services
N_{MON} – Network Delay to invoke Monitoring Software
L_{MD} – Time for loading metadata language
L_{MQ} – Time for loading the monitoring query
G_{CD} – Time for generating SmartFrog component descriptions

M - Number of VMs
N - Number of Operators
Z - Number of Changes in monitored Data

N_{CD} – Network Deplay to distribute Component Description
L_{CD} – Time for loading Component Description at destination
E_{CD} – Time for executing Component Description – Including Installation and Configuration
E_A – Time for executing the adaptor/operator
N_A – Network Deplay for sending the result to the graphical interface
R_{GUI} – Information Receive at the graphical interface

L_i – Level in which the change of monitored data occur
P – Number of Physical Machines
(M/P) – It is the integer division between the number of VMs and the number of associated Physical Machines

$$T_0 = M \cdot (N_{IaaS} + C_{VM} + R_{VM} + N_{MON}) \approx_P (M/P) \cdot N_{IaaS} + C_{VM} + R_{VM} + N_{MON}$$
$$T_1 = N_{MON} + L_{MD} + L_{MQ} + N \cdot G_{CD}$$
$$T_2 = N \cdot (N_{CD} + L_{CD} + E_{CD}) \approx_P (M/P) \cdot N_{CD} + L_{CD} + E_{CD}$$
$$T_3 = \sum_{i=0}^{z} L_i \cdot (E_A + N_A)$$

Figure 6: Analytical Model for the proposed testbed

Figure 7 show the real times achieved in the testbed ranging the number of VMs and the correspondence with the analytical times previous described. For statistical relevance, the results shown in Figure 7 are the average time for 25 executions of each testbed. An immediate observation is the correlation between the time taken for deploying the operators and the number of VMs in the scenario. This is an artefact of the limited resources in our Cloud test bed, as additional VMs create an additional management overhead. Note that 1 VM requires everything running on the same machine, between 2 and 5 VMs there is 1 VM per physical machines, between 6 and 11 VMs there are 2 VM per physical machine, and so on. This fluctuation in the overhead is clearly indicated by the *(M/P)* factor available in T_0 and T_2 of the analytical model. Having this in mind, it can be seen that executions are almost constant when the ratio VM/physical machine keep constants. This shows that the monitoring framework can scale well within its bounds.

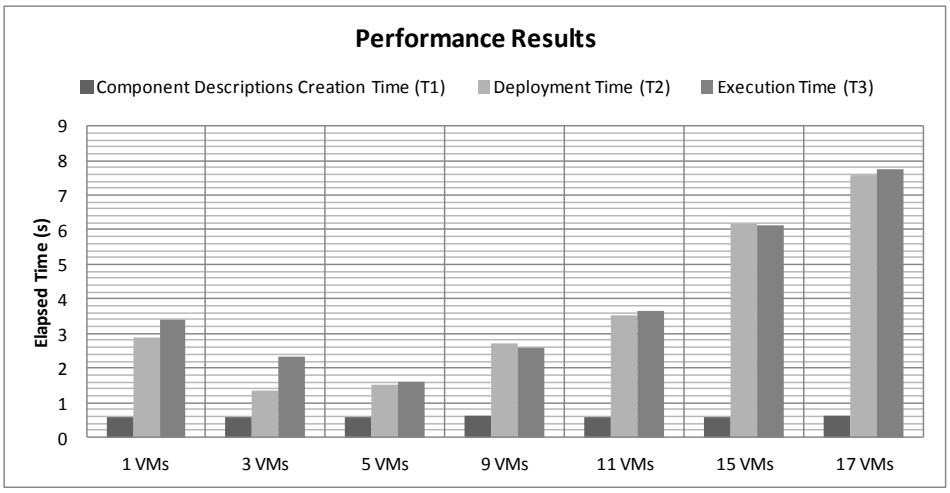


Figure 7: Time for Initialising a Monitoring Query for up to 17 Data Sources

To further analyse the impact of the monitoring framework on the overall resource consumption, the average CPU load of all VMs has been captured over a 1 hour period. It enables us to determine the CPU overhead imposed by the monitoring system. For a better understanding of these results, it is necessary to described how adaptors and operators have been implemented. For instance, the log and database adaptors are polling a log file, or a database table respectively, on a configurable time interval. This causes an update message to be sent, when data has changed between two polling invocations. In the same way, the adaptor for Ganglia is querying for new status information on a 5 second interval. Further, adaptors and query operators are pushing information to parent operators and registered query observers only when changes in the monitored data occur. Having this in mind, the left part of Figure 8 shows the observed load average over all VMs in each scenario. Note that this also includes the base load of the operating system, which is not incurred by our monitoring framework. The average CPU load is around 2.7% when is ranging between 1 and 5 monitored VMs and then constantly increases due to the ratio VM/physical previously described. These results are very promising.

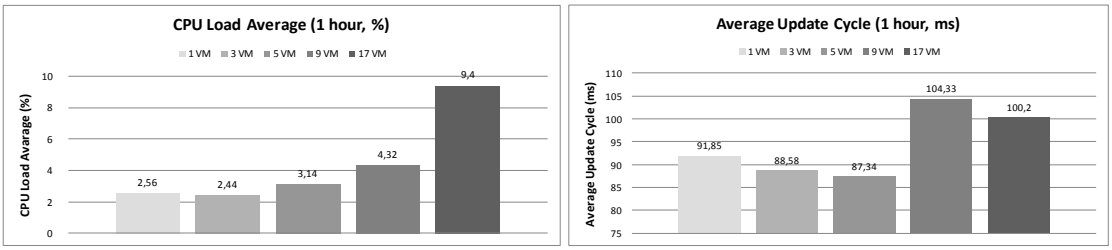


Figure 8: (a) Average CPU-Workload per Hour and (b) Query Update Times

Finally, the times required for a data update to reach its destination, the *test observer,* were determined. It can be considered as the T_3 analytical model after the initial stabilization of the long-living query. In this scenario, an entire update cycle includes data retrieval and homogenisation from its source by a Ganglia adaptor and network transfer of all updated data. The average update cycle has been determined over the course of one hour. Note that each query update has to transfer changed data along all the edges of the query processing tree, triggering individual processing in each individual nodes. The results are shown on the right side of Figure 8. These results show a constant time is spent for updating information across the network, when the number of VMs is increased. These statistics provide promising performance figures, showing that our monitoring approach scales as expected, while having a minimal performance impact on the monitored infrastructure. The results analyse the prototype in different circumstances and not only during the initial convergence time.

7. Conclusions

This paper has described a distributed and elastic architecture for monitoring Cloud infrastructure. It enables scaling resources used for monitoring when required, thus taking advantage of the on-demand infrastructure provided by Cloud environments. This architecture is pure peer-to-peer being suitable for implementing critical monitoring services. Scalability has been also successfully demonstrated by the statistics provided. Queries can be dynamically modified by adding additional data sources to already observed ones. This dynamic approach to monitoring makes the proposed framework an ideal solution for an ever changing Cloud environment, where data sources are deleted and created on a continuous basis.

As future steps, we would like to improve the self-management capabilities provided in the monitored system by establishing a policy-based network management system. Finally, we would like to investigate new techniques for handling automatic disaster recovery management of Cloud infrastructures, by combining our monitoring solution with a service deployment service.

Acknowledgements

Thanks the Funcion Seneca for sponsoring Jose M. Alcaraz Calero under the post-doctoral grant 15714/PD/10.

8. Bibliography

[1] M. Armbrust , A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. *A view of cloud computing*. Communications of the ACM. Vol. 53, No. 4, pp. 50-58. 2010

[2] R. Buyyaa, C. Shin Yeo, S. Venugopal, J. Broberg, I. Brandic. *Cloud computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility*. Future Generation Computer Systems,Vol. 25, No. 6, pp. 599-616, 2006.

[3] J. Hurwitz, R.Bloor, M. Kaufman, F. Halper. *Cloud Computing for dummies. For Dummies*, 1 Edition. Paperback. 2009. ISSN: 9780470484708

[4] W. Barth. *Nagios: System and Network Monitoring,* 2 Edition. No Starch Press, 2008, ISSN: 1593271794

[5] M.Massie, B. N.Chun, D. E. Culler. *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience*, Parallel Computing , Vol. 30, No. 7, pp. 817-840. 2004.

[6] S. Zanikolas, F. Sakellariou. *A taxonomy of grid monitoring systems*. Future Generation Computer Systems Vol. 21, No. 1, pp. 163-188. 2005

[7] D. Bonacorsi, D. Colling, L. Field, S. Fisher, C. Grandi, P. R. Hobson, P. Kyberd, B. MacEvoy, J. J. Nebrensky, H. Tallini, S. Traylen. *Scalability Test of R-GMA Based Grid Job Monitoring System for CMS Monte Carlo Data Production*, IEEE Transactions on Nuclear Science, Vol. 51, No. 6, pp. 3026-3029. 2004

[8]. J. M. Schopf, L.Pearlman, N.Miller, C. Kesselman, I. Foster, M. D'Arcy, A. Chervenak. *Monitoring the grid with the Globus Toolkit MDS4*, Journal of Physics: Conference Series, Vol. 46, pp. 521. 2006.

[9] H. B. Newman, I. C. Legrand, P. Galvez, C. R. Voicu. *MonALISA : A Distributed Monitoring Service Architecture*. Computing in High Energy and Nuclear Physics, La Jolla, 2003.

[10] J. Jiang, Y. Wu, X. Huang, G. Yang, W. Zheng. *Online Video Playing on Smartphones: A Context-Aware*

Approach Based on Cloud Computing.Journal of Internet Technology, Vol. 11 No. 6, pp. 821-828. 2010

[11] L. Zhou, H.-C. Chao. *Multimedia Traffic Security Architecture for Internet of Things*. IEEE Network, Vol. 25, No. 3, pp. 29-34, May 2011.

[12] Y.-X. Lai, C.-F. Lai, C.-C. Hu, H.-C. Chao, Y.-M. Huang. *A Personalized Mobile IPTV System with Seamless Video Reconstruction Algorithm in Cloud Networks*. International Journal of Communication Systems, Vol. 24, No. 10, pp. 1375–1387, October 2011.

[13] C.-F. Lai, J.-H. Chang, C.-C. Hu, Y.-M. Huang, H.-C Chao, *CPRS: A Cloud-based Program Recommendation System for Digital TV Platform*. Future Generation Computer Systems, Vol. 27, No. 6, pp. 823-835, June 2011.

[14] S.-C. Wang, S.-S. Wang, K.-Q. Yan, L.-H. Chang, C.-P. Huang. *Reaching Fast Agreement in a Generalized Cloud Computing Environment*. Journal of Internet Technology, Vol. 11 No. 7, P.975-984. 2010

[15]. P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, P. Toft. *The SmartFrog configuration management framework*. ACM SIGOPS Operating Systems Review, Vol. 43, pp. 16-25. 2009

[16] J. Kirschnick, J. M. Alcaraz Calero, L. Wilcock, N. Edwards. *Towards an Architecture for the Automated Provisioning of Cloud Services*. IEEE Communications Magazine. Vol. 48, No. 12, pp. 124-131, 2010

[17] J. Turnbull. *Pulling Strings with Puppet*. 1 Edition. FristPress, 2007, ISSN: 1590599780

[18] J. Kirschnick, J. M. Alcaraz Calero, P. Goldsack, J. Guijarro, S. Loughran, N. Edwards, L. Wilcock. *SmartFrog: A Framework for Deploying Services in the Cloud.* Software: Practice and Experience, 2010, (first on-line)

[19] D. Koenig, A. Glover, P. King, G. Laforge, J. Skeet. *Groovy in Action*, 2 Edition, Manning. 2007, ISSN: 1932394842

[20] L. Cardelli, X. Leroy. *Abstract types and the dot notation*. Proceedings IFIP TC2 working conference on programming concepts and methods , 1990, pp.479-504. ISSN: 0444885455

[21] J. M. Hellerstein, M. Stonebraker, J. Hamilton. *Architecture of a Database System*. Foundation and Trends in Databases. Vol. 1, No. 2, pp. 141-259. 2007