Towards Autonomic Detection of SLA Violations in Cloud Infrastructures

Vincent C. Emeakaroha[a], Marco A. S. Netto[b], Rodrigo N. Calheiros[c], Ivona Brandic[a], Rajkumar Buyya[c], César A. F. De Rose[b]

[a]*Vienna University of Technology, Vienna, Austria*
[b]*Faculty of Informatics, Catholic University of Rio Grande do Sul, Porto Alegre, Brazil*
[c]*CLOUDS Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, Australia*

Abstract

Cloud computing has become a popular paradigm for implementing scalable computing infrastructures provided on-demand on a case-by-case basis. Self-manageable Cloud infrastructures are required in order to comply with users' requirements defined by Service Level Agreements (SLAs) and to minimize user interactions with the computing environment. Thus, adequate SLA monitoring strategies and timely detection of possible SLA violations represent challenging research issues. This paper presents the Detecting SLA Violations infrastructure (DeSVi) architecture, sensing SLA violations through sophisticated resource monitoring. Based on the user requests, DeSVi allocates computing resources for a requested service and arranges its deployment on a virtualized environment. Resources are monitored using a novel framework capable of mapping low-level resource metrics (e.g., host up and down time) to user-defined SLAs (e.g., service availability). The detection of possible SLA violations relies on the predefined service level objectives and utilization of knowledge databases to manage and prevent such violations. We evaluate the DeSVi architecture using two application scenarios: i) image rendering applications based on ray-tracing; and ii) transactional web applications based on the well-known TPC-W benchmark. These applications exhibit heterogeneous workloads for investigating optimal monitoring interval of SLA parameters. The achieved results show that our architecture is able to monitor and detect SLA violations. The architecture output also provides a guideline on the appropriate monitoring intervals for applications depending on their resource consumption behavior.

Keywords: Service Level Agreement, Resource Monitoring, SLA Violation Detection, SLA Enactment, Cloud Architecture

1. Introduction

Cloud computing represents a novel paradigm for the implementation of scalable computing infrastructures combining concepts from virtualization, distributed application design, Grid, and enterprise IT management [1, 2, 3]. Service provisioning in the Cloud relies on Service Level Agreements (SLAs) representing a contract signed between the customer and the service provider including non-functional requirements of the service specified as Quality of Service (QoS) [4, 5]. SLA considers obligations, service pricing, and penalties in case of agreement violations.

Flexible and reliable management of SLA agreements is of paramount importance for both Cloud providers and consumers. On the one hand, prevention of SLA violations avoids penalties providers have to pay and on the other hand, based on flexible and timely reactions to possible SLA violations, user interaction with the system can be minimized, which enables Cloud computing to take roots as a flexible and reliable form of on-demand computing.

Although, there is a large body of work considering development of flexible and self-manageable Cloud computing infrastructures [6, 7, 8], there is still a lack of adequate monitoring infrastructures able to predict possible SLA violations. Most of the available monitoring systems rely either on Grid [9, 10] or service-oriented infrastructures [11], which are not directly compatible to Clouds due to the difference of resource usage model, or due to heavily network-oriented monitoring infrastructures [12]. In Grids [13] resources are mostly owned by different individuals/enterprises, and in some cases, as desktop Grids for instance, resources are only available for usage when the owners are not using them [14]. Therefore, resource availability varies much and this impacts its usage for application provisioning, whereas in Cloud computing, resources are owned by an enterprise (Cloud provider), provisioning them to customers in a pay-as-you-go manner. Therefore, availability of resources is more stable and resources can be provisioned on-demand. Hence, the monitoring strategies used for detection of SLA violations in Grids cannot be directly applied to Clouds.

Furthermore, another important aspect for the usage of SLAs is the required elasticity of Cloud infrastructures. Thus, SLAs are not only used to provide guarantees to end user, they are also used by providers to efficiently manage Cloud infrastructures, considering competing priorities like energy efficiency and attainment of SLA agreements [15, 16] while delivering sufficient elasticity. Moreover, SLAs are also recently used as part of novel Cloud engineering models like Cloud federation [17, 18] where provider can in- or outsource their infrastructure depending on the current load. Thus, since SLA parameters are usually defined by Cloud providers and can comprise various user-defined attributes, current monitoring infrastructures lack appropriate solutions for adequate SLA monitoring. The first challenge is to facilitate mapping of measured metrics by low level tools to application based SLAs. The second challenge is to determine appropriate monitoring intervals at the application level keeping the balance between the early detection of possible SLA violations and system intrusiveness of the monitoring tools.

In this paper we present the novel concept for mapping low-level resource metrics to high-level SLAs—*LoM2HiS* [19], where system metrics (e.g., system up and down time) are translated to high-level SLAs (e.g., system availability). Thus, LoM2HiS facilitates efficient monitoring of Cloud infrastructures and early detection of possible SLA violations. Furthermore, LoM2HiS framework enables user-driven mappings between the resource metric and SLA parameters by utilizing mapping rules defined with Domain Specific Languages (DSLs). However, determination of optimal measurement intervals of low-level metrics and their translation to SLAs is still an open research issue. Short measurement intervals may negatively affect the overall system performance, whereas long measurement intervals may cause heavy SLA violations.

In order to assist Cloud providers in detecting SLA violations through resource monitoring, we developed the DeSVi architecture [20]. This architecture represents a core step towards achieving flexible and autonomic SLA management. The main components of the DeSVi architecture are: (i) *the automatic VM deployer*, (ii) *application deployer*, and (iii) *the LoM2HiS framework*. Based on user requests, the *automatic VM deployer* allocates necessary resources for the requested service and arranges its deployment on a virtual machine (VM). After service deployment, LoM2HiS framework monitors the VMs and translates the low-level metrics into high-level SLAs using the specified mapping rules. To realize autonomic SLA management DeSVi utilizes a knowledge database for the evaluation of the monitored information in order to propose reactive actions in case of SLA violation situations.

The main contributions of the paper are: (i) definition of the motivation scenario for the development of the architecture aimed at detecting SLA violations, (ii) conceptual design of the *DeSVi* architecture for the prediction of SLA violations, (iii) discussion of the implementation choices for the DeSVi, and (iv) extensive evaluation of the architecture in a real computing infrastructure using various SLA parameters and two Cloud applications: an image rendering service based on POV-Ray[1] and the TPC-W transactional web e-Commerce benchmark[2].

The rest of this paper is organized as follows: Section 2 presents the related work. Section 3 presents the architecture for the autonomic management of Cloud services and the motivating scenario for the development of the *DeSVi* architecture. Section 4 introduces the *DeSVi* architecture. In particular we discuss the automatic VM deployer, application deployer, and the monitoring components. Section 5 discusses our implementation choices, whereas Section 6 discusses experimental evaluation of the *DeSVi* architecture. Section 7 presents our conclusions and describes future work.

2. Related Work

We classify related work into (i) resource monitoring [12, 21, 22], (ii) SLA management including violation detection [23, 24, 25, 26, 27], and (iii) mapping techniques of monitored metrics to SLA parameters [11, 28]. Currently, there is little work in the area of resource monitoring, low-level metrics mapping, and SLA violation detection in Cloud computing. Because of that, we look into the related areas of Grid and Service-Oriented Architecture (SOA) based systems.

Fu *et al.* [21] propose GridEye, a service-oriented monitoring system with flexible architecture that is further equipped with an algorithm for prediction of the overall resource performance characteristics. The authors discuss how resources are monitored with their approach in Grid environment but they consider neither SLA management nor low-level metric mapping. Gunter *et al.* [12] present NetLogger, a distributed monitoring system, which can monitor and collect information of networks. Applications invoke NetLogger's API to survey the overload before and after some request or operation. However, it monitors only network resources. Wood *et al.* [22] developed a system,

[1]`http://www.povray.org`

[2]`http://www.tpc.org/tpcw/`

called Sandpiper, which automates the process of monitoring and detecting hotspots and remapping/reconfiguring VMs whenever necessary. Their monitoring system is reminiscent of our in terms of goal: avoid SLA violation. Similar to our approach, Sandpiper uses thresholds to check whether SLAs can be violated. However, it differs from our system by not considering the mapping of low level metrics, such as CPU and memory, to high-level SLA parameters, such as response time for SLA enactment.

Boniface *et al.* [23] discuss dynamic service provisioning using GRIA SLAs. The authors describe provisioning of services based on agreed SLAs and the management of the SLAs to avoid violations. Their approach considers only Grid environments and not Clouds. Moreover, they do not detail how the low-level metric are monitored and mapped to high-level SLAs to enforce the SLA objectives at runtime. Koller *et al.* [24] discuss autonomous QoS management using a proxy-like approach. Their implementation is based on WS-Agreement. Thereby, SLAs can be exploited to define certain QoS parameters that a service has to maintain during its interaction with a specific customer. However, their approach is limited to Web services and does not consider other applications types. Frutos *et al.* [25] discuss the main approach of the EU project BREIN [29] to develop a framework that extends the characteristics of computational Grids by driving their usage inside new target areas in the business domain for advanced SLA management. BREIN applies SLA management to Grids, whereas we target SLA management in Clouds. Dobson *et al.* [27] present a unified QoS ontology applicable to QoS-based Web services selection, QoS monitoring, and QoS adaptation. However they do not consider application deployment and provisioning strategies. Comuzzi *et al.* [26] define the process for SLA establishment adopted within the EU project SLA@SOI framework. The authors propose the architecture for monitoring SLAs considering two requirements introduced by SLA establishment: the availability of historical data for evaluating SLA offers and the assessment of the capability to monitor the terms in an SLA offer. But they do not consider monitoring of low-level metrics and mapping them to high-level SLA parameters for ensuring the SLA objectives.

Rosenberg *et al.* [28] deal with QoS attributes for Web services. They identify important QoS attributes and their composition from resource metrics. They present mapping techniques for composing QoS attributes from resource metrics to form SLA parameters for a specific domain. However, they do not deal with monitoring of resource metrics. Bocciarelli *et al.* [11] introduce a model-driven approach for integrating performance prediction into service composition processes carried out by BPEL. In their approach, service SLA parameters are composed from system metrics using mapping techniques. Nevertheless, they consider neither resource metric monitoring nor SLA violation detection.

To the best of our knowledge, none of the discussed approaches deals with mapping of low-level resource metrics to high-level SLA parameters and SLA violation detection at runtime, which are desirable features for enforcing SLAs in Cloud-like environments.

3. Background and Motivation

The processes of service provisioning based on SLA and efficient management of resources in an autonomic manner have been identified as major research challenges in Cloud environments [1, 30]. FoSII project (Foundations of Self-governing Infrastructures) is developing models and concepts for autonomic SLA management and enforcement in Clouds. FoSII components manage the whole lifecycle of self-adaptable Cloud services [6] as explained next.

SLA are used to guarantee customers a certain level of quality for their services. In a situation where this level of quality is not met, the provider pays penalties for the breach of contract. In order to save Cloud providers from paying penalties and increase their profit, providers have to monitor the current status or resource and check frequently whether the established SLAs are violated. Thus, in order to facilitate appropriate monitoring of SLAs we developed the Low Level Metrics to High Level SLA *(LoM2HiS framework)* [19] that maps the low-level resource metrics to high-level SLA parameters and detects SLA violations as well as future SLA violation threats so as to react before actual SLA violations occur.

DeSVi architecture utilizes LoM2HiS framework to detect application SLA objectives violations at runtime and extends FoSII with application deployment component, virtual machine configuration and deployer components.

3.1. FoSII Infrastructure Overview

Figure 1 depicts the components of the FoSII infrastructure. There are two core components of the FoSII infrastructure . The first part comprises *the monitoring aspect* and it is intended to provide information to the second part, which comprises *the knowledge management aspect.* As shown in Figure

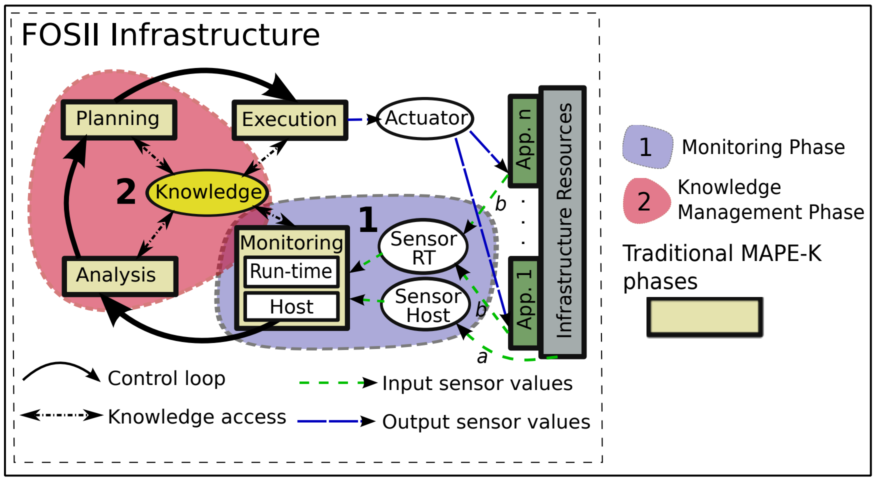


Figure 1: FoSII Infrastructure Overview.

1, each FoSII service implements three interfaces: (i) negotiation interface necessary for the establishment of SLA agreements, (ii) application management interface necessary to start the application, upload data, and perform similar management actions, and (iii) self-management interface necessary to devise actions in order to prevent SLA violations.

The self-management interface shown in Figure 1 is implemented by each Cloud service and specifies operations for sensing changes of the desired state and for reacting to those changes [6]. The host monitor sensors continuously monitor the infrastructure resource metrics (input sensor values arrow a in Figure 1) and provide the autonomic manager with the current resource status. The run-time monitor sensors sense future SLA violation threats (input sensor values arrow b in Figure 1) based on resource usage experiences and predefined threat thresholds.

In this paper we give a brief description of the knowledge management component first, but our focus is on the LoM2HiS framework since it implements monitoring strategies relevant for the realization of the DeSVi architecture.

```
1.  (
2.   (App,  1),
3.    (
4.    ((Incoming Bandwidth, 12.0),
5.      (Outgoing Bandwidth, 20.0),
6.      (Storage, 1200),
7.      (Availability, 99.5),
8.      (Running on PMs, 1)),
9.      (Physical Machines, 20)
10.    ),
11.  "Increase Incoming Bandwidth share by 5%",
12.  (
13.   ((Incoming Bandwidth, 12.6),
14.     (Outgoing Bandwidth, 20.1),
15.     (Storage, 1198),
16.     (Availability, 99.5),
17.     (Running on PMs, 1)),
18.     (Physical Machines, 20)
19.   ),
20.  0.002
21. )
```

Figure 2: Case Based-Reasoning example.

3.2. Knowledge Databases

Knowledge management in FoSII is performed based on knowledge databases and case-based reasoning [31] for proposing of reactive actions. *Case-Based Reasoning (CBR)* is the process of solving problems based on past experience. It tries to solve a *case* (a formatted instance of a problem) by looking for similar cases from the past and reusing the solutions of these cases to solve the current one. In general a typical CBR cycle consists of the following phases assuming that a new case has just been received: (i) retrieving the most similar case or cases to the new one, (ii) reusing the information and knowledge in the similar case(s) to solve the problem, (iii) revising the proposed solution, and (iv) retaining the parts of this experience likely to be useful for future problem solving.

Considering the SLA depicted in Table 1 and as shown in Figure 2, a complete case consists of (a) the application ID being considered (line 2, Figure 2); (b) the initial case measured by the monitoring component and mapped to the SLAs consisting of the SLA parameter values of the application and global Cloud information like number of running virtual machines (lines 4-9); (c) the executed action (line 11); (d) the resulting case measured some time interval later (lines 13-18) as in (b); and (e) the resulting utility (line 20).

We distinguish between two working modes of the knowledge DB: *active*

Table 1: Sample SLA parameter objectives.

SLA Parameter	Value
Incoming Bandwidth (IB)	> 10 Mbit/s
Outgoing Bandwidth (OB)	> 12 Mbit/s
Storage (St)	> 1024 GB
Availability (Av)	$\geq 99\%$

and *passive* [31]. In the active mode, system states and SLA values are periodically stored into the DB. Thus, based on the observed violations and correlated system states, cases are obtained as input for the knowledge DB. Furthermore, based on the utility functions, quality of the reactive actions are evaluated and threat thresholds are generated.

However, definition of the measurement intervals in the active mode is far from trivial. An important parameter to be considered is the period on which resource metrics and SLA parameters are evaluated (e.g. every two seconds or every two minutes). Too frequent measurement intervals may negatively affect the overall system performance, whereas too infrequent measurement intervals may cause heavy SLA violations. Even though the knowledge database component is essential for the achievement of autonomic and self-management behavior in the FoSII infrastructure, it does not relate directly to the architectural components described in this paper, and so it is not discussed further.

3.3. LoM2HiS Framework Overview

The LoM2HiS framework comprises two core components, namely *host monitor* and *run-time monitor*. The former is responsible for monitoring low-level resource metrics, whereas the latter is responsible for metric mapping and SLA violation monitoring. In order to explain our mapping approach we consider the Service Level Objectives (SLOs) shown in Table 1, including incoming bandwidth, outgoing bandwidth, storage, and availability.

As shown in Figure 1, we distinguish between *host monitor* and *run-time monitor*. Resources are monitored by the *host monitor* using arbitrary monitoring tools such as Gmond from Ganglia project [32]. Low level resource metrics include downtime, uptime, and available storage. Based on the predefined mapping rules stored in a database, monitored metrics are periodically mapped to the high level SLA parameters. These mapping ideas are similar to those in Grids where workflow processes are mapped to Grid service in order to ensure their quality of service [33]. An example of an SLA parameter is service availability Av, (as shown in Table 1), which is calculated using the resource metrics *downtime* and *uptime* as defined by the following mapping rule:

$$Av = (1 - downtime/uptime) * 100. \tag{1}$$

The mapping rules are defined by the provider using appropriate Domain Specific Languages (DSLs). DSLs are special-purpose languages that can be tailored to a specific problem domain. SLA parameters are specified based on the type of application in question. There are different types of applications that can be grouped into domains based on the composition of their SLA parameters. Thus, the use of DSL to describe the mapping rules. These rules are used to compose, aggregate, or convert the low-level metrics to form the high-level SLA parameter including mappings at different complexity levels, e.g., $1 : n$ or $n : m$. The concept of detecting future SLA violation threats is designed by defining a more restrictive threshold than the SLA violation threshold known as threat threshold. Thus, calculated SLA values are compared with the predefined threat threshold in order to react before SLA violations happen. The generation of *threat thresholds*, described in Section 3.2, is part of our ongoing work and includes sophisticated methods for system state management.

As described in a previous work [19], we designed and implemented a communication model for the LoM2HiS framework based on the Java Messaging Service (JMS) API [34], which is a Java Message Oriented Middleware API for sending messages between two or more clients. We use Apache ActiveMQ [35] as a JMS provider that manages sessions and queues.

Having discussed the FoSII infrastructure, we now present in the next section the DeSVi architecture, which extends the FoSII infrastructure with two new components.

4. DeSVI Architecture

This section describes in detail the Detecting SLA Violation infrastructure–*DeSVi architecture*, its components, and how the components interact with one another (Figure 3). The proposed architecture is designed to handle the complete service provisioning management lifecycle in Cloud environments.

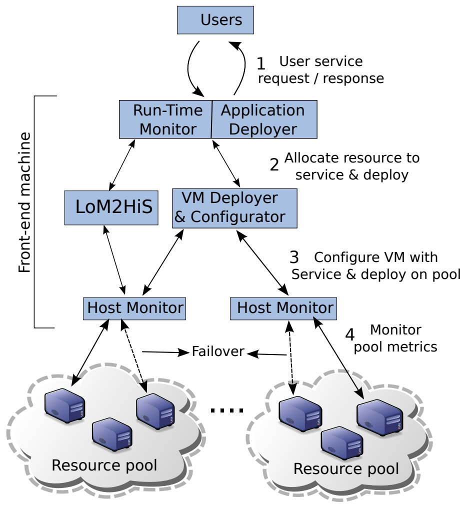


Figure 3: Overview of the DeSVi architecture and component's interaction.

The service provisioning lifecycle includes activities such as service deployment, resource allocation to tasks, resource monitoring, and SLA violation detection.

The topmost layer represents the users (customers) who place service provisioning request through a defined application interface (step 1 in Figure 3) to the Cloud provider. The provider handles the user service request based on the negotiated and agreed SLAs with the user. The application deployer, which is located on the same layer of the run-time monitor, allocates necessary VM resources for the requested service and arranges its deployment on the Cloud environment (step 2). VMs are not the only type of resources in a Cloud environment but we do emphasize them in this work because it is essential to our approach. The deployment of VMs and environmental configurations are performed by AEF (Automated Emulation Framework) [8] (step 3). The host monitor observes the metrics of the resource pool comprising virtual machines and physical hosts (step 4). The relation between the resource metrics (monitored by the host monitor) and SLAs (monitored by the run-time monitor) is managed by the LoM2HiS framework.

The arrow termed *Failover* presented in Figure 3 indicates redundancy in the monitoring mechanism. The host monitor is designed to use monitoring agents as mentioned previously, which are embedded in each node in the resource pool to monitor the metrics of the node. Such monitoring agents broadcast their monitored values to the other agents in the same resource pool, creating the possibility of accessing the whole resource pool status from any node in the pool. The metric broadcasting mechanism is configurable and can be deactivated if necessary but it can obviate the problem of a bottleneck master node for accessing the monitored metrics of the resource pool.

The DeSVi architecture is designed to monitor and detect SLA violation in a single Cloud data center. To be able to manage a Cloud environment with multiple data centers, we intend to apply a decentralization approach where the proposed system will be installed on each data center. The LoM2HiS component in our system is already designed with a scalable communication mechanism, which can be easily utilized to allow communication between data centers. In the following sections we explain all components of our system in detail.

4.1. Application Deployer

The Application Deployer is responsible for managing the execution of user applications; similar to *brokers* in the Grid literature [36, 37, 38]. However, compared to brokers, the Application Deployer has more knowledge and control of the application tasks, being able to perform application-level scheduling, for example, for parameter sweeping executions [39]. It provides an application interface to the users and simplifies the processes of transferring application input data to each VM, starting the execution, and collecting the results from the VMs to the front-end node. The mapping of application tasks to VMs is performed by a scheduler located in the Application Deployer. After deploying application on the VMs, the application deployer stores the VM IDs, which is used by the monitoring component to identify the VMs to monitor.

Figure 4 illustrates the main modules of the Application Deployer. The *task generator* integrated with the application interface receives from the user the application and its parameters, and at the same time the VM deployer generates a machine file based on user requirements (step 1). The *scheduler* uses this machine file and a list of all tasks (step 2) to map tasks to VMs (step 3). Each VM contains an executor, which requests tasks from the *task manager* whenever executors are idle and there are tasks to be executed, thus allowing a dynamic load balancing (step 4). The *task manager* is also responsible for triggering the task executions on VMs (step 5) and collecting the results when tasks complete.

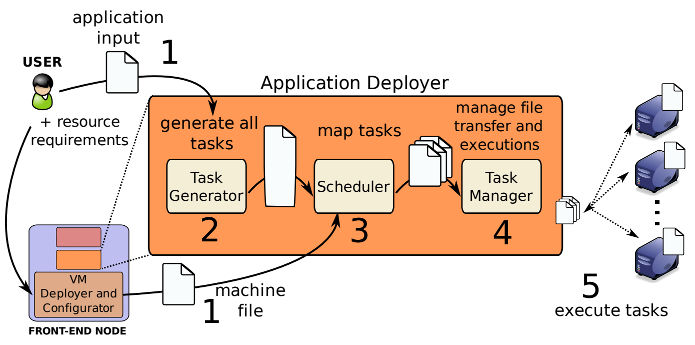


Figure 4: Application Deployer.

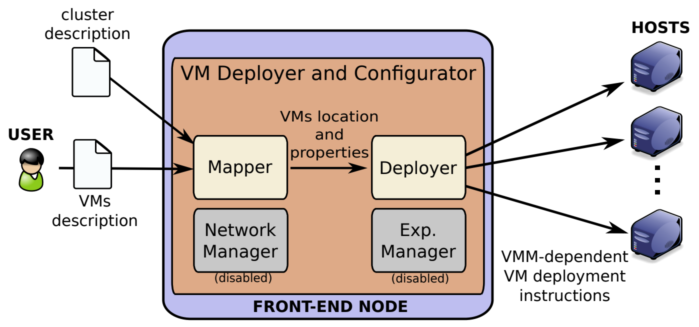


Figure 5: AEF Framework.

4.2. Automated Emulation Framework

The Automated Emulation Framework (AEF) was originally conceived for automated configuration and execution of emulation experiments [8]. Nevertheless, it can also be used to set up arbitrary virtual environments by not activating the emulated wide-area network support. In the latter case AEF works as a virtualized infrastructure manager, similar to tools such as OpenNebula [40], Oracle VM Manager [41], and OpenPEX [42].

Figure 5 depicts the architecture of the AEF framework. AEF input consists of two configuration files providing XML description of both the physical and virtual infrastructures. Using this information, AEF maps VMs to physical hosts. AEF supports different algorithms for VM mapping. The algorithm used in this work tries to reduce the number of hosts used by consolidating VMs as long as one host has enough resources to host several VMs. At the end of the mapping process, the resulting mapping is sent to the Deployer, which creates VMs in the hosts accordingly.

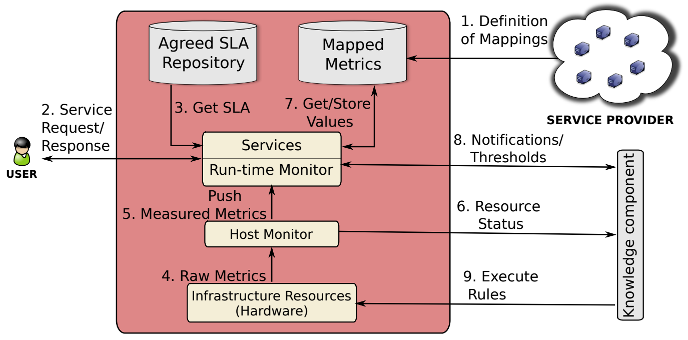


Figure 6: LoM2HiS framework.

If network configuration is required in the environment (e.g. to create virtual networks), the Network Manager component of AEF performs this activity. Execution of the applications may be triggered either by the user, in case of interactive applications, or directly by AEF in case of non-interactive applications. In the experiments presented in this paper we opted for the former approach where the execution is triggered by the application deployer. VMs can be accessed via cluster front-end and then users can log in the machine and interact with the application.

4.3. Monitoring

Monitoring in DeSVi is performed by the LoM2HiS framework, whose architecture is presented in Figure 6. The run-time monitor is designed to monitor the services based on the negotiated and agreed SLAs. After agreeing on SLA terms, the service provider creates mapping rules for the LoM2HiS mappings (step 1 in Figure 2) using Domain Specific Languages (DSLs) to define specific rules for different application domains. An example rule is presented in Equation 1. Once the customer requests the provisioning of an agreed service (step 2), the run-time monitor loads the service SLA from the agreed SLA repository (step 3). Service provisioning is based on the infrastructure resources, which represent the hosts and network resources in a data center for hosting Cloud services. The resource metrics are measured by monitoring agents, and the measured raw metrics are accessed by the host monitor (step 4). The host monitor extracts metric-value pairs from the raw metrics and transmits them periodically to the run-time monitor (step 5) and to the knowledge component (step 6) using our novel communication model as presented in [19].

Upon receipt of the measured metrics, the run-time monitor maps the low-level metrics based on predefined mapping rules to form an equivalence of the agreed SLA objectives. The resulting mapping is stored in the mapped metric repository (step 7), which also contains the predefined mapping rules. The run-time monitor uses the mapped values to monitor the status of the deployed services. In case future SLA violation threats occur, it notifies (step 8) the knowledge component for preventative actions. The knowledge component also receives the predefined threat thresholds (step 8) for possible adjustments due to environmental changes at run-time. This component works out an appropriate preventative action to avert future SLA violation threats based on the resource status (step 6) and defined rules. Finally, knowledge component's decisions (e.g. assign more CPU to a virtual host) are executed on the infrastructure resources (step 9).

5. Implementation Issues

In this section, we describe the implementation choices for each *DeSVi* component. The implementation of the *DeSVi* components targets the fulfillment of some fundamental Cloud requirements such as scalability, efficiency, and reliability. To achieve these goals, we incorporated, whenever possible, well-established and tested open source tools in the implementation. Results presented in Section 6 where obtained with utilization of the components presented in this section.

5.1. Application Deployer

The Application Deployer is written in Java and has as input a machine file (in plain ASCII format), which contains the list of hostnames or IPs of the VMs allocated to the user application and a task generator Java class to split the work to be done into a lists of tasks. For a rendering application, for instance, such a class includes a list of frames and the command to render them. The division of tasks per VM is performed by the Application Deployer's scheduler as described in Section 4.1.

The Application Deployer uses *scp*, a standard tool for copying files among multiple machines, in order to transfer the application-related files from the front-end node to VMs responsible for executing tasks. The *ssh* command is responsible for triggering an executor on each VM specified in the machine file. Each executor requests tasks to be executed from the task manager. During the user application execution, the Application Deployer generates log files with the time required to execute each task. After tasks executions are completed, the results are transferred back to the front-end node via *scp*. This model was chosen because it provides a reliable mechanism for file transferring (*scp*) together with persistent logging information that does not depend on a DBMS to archive results. The overall result of the approach is a reliable and lightweight mechanism for managing tasks that has an insignificant overhead on the platform, what is a requirement of a system aiming at managing QoS of resources.

5.2. Virtual Machine Deployer and Configurator

The automated emulation framework used to deploy and configure the virtual machines is implemented in Java. The framework inputs are XML files describing the characteristics of both, the required virtual machines and the cluster. Once these files are parsed, the Mapper component maps the virtual machines to cluster nodes. During this stage, AEF ensures that the resources required by all virtual machines assigned to a cluster node do not exceed the node's available resources.

Once the mapping is finished, the resulting configuration is applied in the cluster by the VM Deployer component. Here, a parallel standalone deployer, which is part of the AEF core, is used. This parallel deployer module does not require external tools or systems for its operation, and it works as follows. First, a base image file of the virtual machines is copied, via *scp*, to each cluster node (as determined by the Mapper) simultaneously. This image contains all the software and configuration required by the application. After the base image is copied to each physical machine, it is replicated there to achieve the number of virtual images intended to be deployed on this specific physical machine. This step is also carried out simultaneously on each physical machine.

The replicated images are configured with VM-specific settings, such as hostname and static IP address. Finally, virtual machines are simultaneously created on each host from each image file replicated in the previous step. Furthermore, the deployer checks if an image is already present in a host before performing the transfer. Thus, if the image is already present in the host, the transfer process is skipped in such a host, saving bandwidth for the transfer of images in other hosts. Moreover, if the replicated VM images on each host are newer than the base image in use, the replication process is skipped. AEF was used because it is lightweight and supports deployment of systems based on Xen with negligible overhead. Moreover, its parallel transfer of VMs and selective replication of images reduces the amount of time required for building and deployment of the virtual environment.

5.3. LoM2HiS Components

The *host monitor* implementation uses the standalone Gmond module from the Ganglia open source project [32] as monitoring agent, as it is a widely used, open source monitoring software. We use it to monitor the low-level resource metrics. The monitored metric results are presented in an XML file and written to a predefined network socket. With our implemented Java routine, the host monitor listens to this network socket where Gmond writes the XML file containing the monitored metrics to access them. Furthermore, we implemented an XML parser using the well-known open source SAX API [43] to parse the XML file in order to extract the metric-value pairs. These metric-value pairs are sent to the run-time monitor using our implemented communication model.

Our *communication model* exploits the capabilities of the Java Messaging Service API, which is a Java message oriented middleware for sending message between two or more clients. In order to use JMS, there is a need for a JMS provider that is capable of managing the sessions and queues. We used the well-established open source Apache ActiveMQ [35] for this purpose.

The *run-time monitor* implementation passes the received metric-value pairs into ESPER engine [44], which provides a filter to remove identical monitored values so that only changed values between measurements are delivered for further processing. This strategy drastically reduces the number of messages processed in the run-time monitor. The received metric-value pairs are stored in MySQL DB from where the mapping routine accesses them and applies the appropriate mappings. The agreed service SLA is also stored in the same DB accessible to the run-time monitor. Furthermore, we implemented a Java routine that checks for SLA violations by comparing the mapped SLA against the agreed service level objectives.

6. Evaluation

This section discusses the evaluation of our approach using two use-case scenarios. The use-case scenarios represent the most dominant application domains provisioned in Clouds today, namely (i) high performance computing applications, which include image processing and scientific simulations;

and (ii) transactional applications, which include web applications, social network sites, and media sites. The first use-case scenario comprises three types of ray-tracing applications based on POV-Ray, and the second one comprises executions of TPC-W, which is a well-known web application benchmark that simulates a web server for on-line shopping. The goal of our evaluation is to determine the efficiency of the proposed architecture in detecting SLA violations at runtime and, based on its output, suggest optimal measurement intervals for monitoring applications considering the application resource consumption behavior.

Section 6.1 describes the experimental environment setup. Next, Section 6.2 presents the definition of a cost function, which is used to analyze the achieved results of the two use-case scenarios. Sections 6.3 and 6.4 respectively discuss the two experimental use-case scenarios including their achieved results, the results analysis and the derived conclusions of the results.

6.1. Experimental Environment

Our basic Cloud experimental testbed is shown in Table 2. The table shows the resource capacities of the physical and the virtual machines being used in our experimental testbed. We use Xen virtualization technology in the testbed, precisely we run Xen 3.4.0 on top of Oracle Virtual Machine (OVM) server.

Table 2: Cloud Environment Resource Setup Composed of 36 Virtual Machines.

Machine Type = Physical Machine

OS	**CPU**	**Cores**	**Memory**	**Storage**
OVM Server	AMD Opteron 2 GHz	2	8 GB	250 GB

Machine Type = Virtual Machine

OS	**CPU**	**Cores**	**Memory**	**Storage**
Linux/Ubuntu	AMD Opteron 2 GHz	1	1024 MB	5 GB

We have in total nine physical machines and, based on the resource capacities presented in Table 2, we host 4 VMs on each physical machine. AEF deploys the VMs onto the physical hosts, thus creating a virtualized Cloud environment with up to 36 computing nodes capable of provisioning resources to applications and one front-end node responsible for management activities.

The front-end node serves as the control entity. It runs the automated emulation framework, the application deployer, and the LoM2HiS framework, which are the core components of the DeSVi architecture. The first two components are the supporting blocks of the experiments, whereas the third component is the main responsible for the results obtained in this section. Nevertheless, their integration is required in order to enable the experiments. We use this virtualized environment to evaluate the two use-case scenarios presented in the rest of this section.

6.2. Cost Function Definition

To suggest an optimal measurement interval for detecting applications' SLA objectives violations at runtime, we discuss the following two determining factors i) cost of making measurements; and ii) the cost of missing SLA violations. The acceptable trade-off between these two factors defines the optimal measurement interval.

Using these two factors and other parameters we define a cost function (C) based on which we can derive an optimal measurement interval. The ideas of defining this cost functions are derived from utility functions discussed by Lee *et al.* [45]. Equation 2 presents the cost function.

$$C = \mu * C_m + \sum_{\psi \epsilon \{cpu, memory, storage\}} \alpha(\psi) * C_v \qquad (2)$$

where μ is the number of measurements, C_m is the cost of measurement, $\alpha(\psi)$ is the number of undetected SLA violations, and C_v is the cost of missing an SLA violation. The number of undetected SLA violations are determined based on the results of the reference measurement interval, which is assumed to be an interval capturing all the violations of an application SLA objectives.

This cost function now forms the basis for analyzing the achieved results of our two use-case scenarios in the later sections. Regarding the two determining factors, we explain for each use-case scenario how we obtained these cost values.

6.3. Image Rendering Application Use-Case

We developed an image rendering application based on the Persistence of Vision Raytracer (POV-Ray), which is a ray tracing program available for several computing platforms. In order to achieve heterogeneous load in this use-case scenario, we experiment with three POV-Ray workloads, each one with a different characteristic of time for rendering frames, as described below and illustrated in Figures 7 and 8:

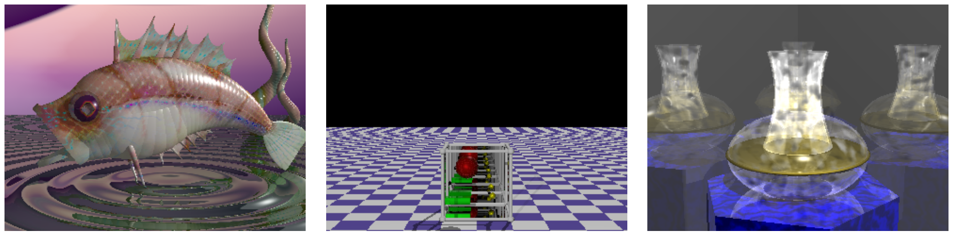
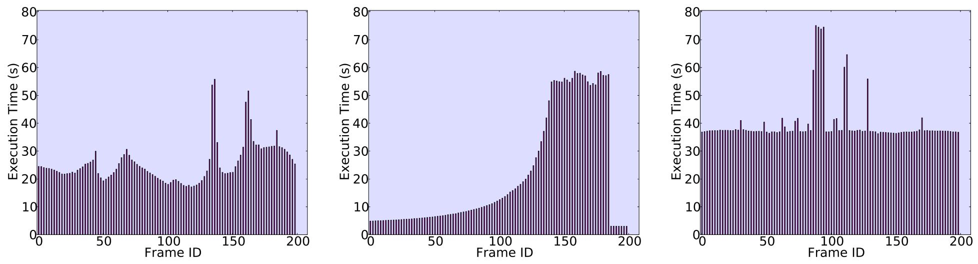

(a) Fish. (b) Box. (c) Vase.

Figure 7: Example of images for each of the three animations.



(a) Fish. (b) Box. (c) Vase.

Figure 8: Behavior of execution time for each POV-Ray application.

- **Fish:** rotation of a fish on water. Time for rendering frames is variable.
- **Box:** approximation of a camera to an open box with objects inside. Time for rendering frames increases during execution.
- **Vase:** rotation of a vase with mirrors around. Time for processing different frames is constant.

Three SLA documents are specified for the three POV-Ray applications. The SLA documents specify the level of Quality of Service (QoS) that should be guaranteed for each application during its execution. Table 3 presents the SLA objective thresholds for each of the applications. It should be noted that we are not addressing the issues of SLA definition and formalization, rather we specify SLA parameters relevant to the Cloud provider in order to manage the users' applications. These SLA objective thresholds are defined based on historical data and experiences with these specific type of applications in terms of resource consumption [46]. With the historical data, the Cloud provider can determine the amount and type of resources the application requires. Thus, the provider can make better resource provisioning plan for the applications.

Based on these SLA objective thresholds, the applications are monitored to detect SLA violations. These violations may happen either because of unforeseen resource consumptions or because SLAs are negotiated per application and not per allocated VM considering the fact that the service provider may provision different application requests on the same VM.

Table 3: SLA objective thresholds for the three POV-Ray applications.

SLA Parameter	**Fish**	**Box**	**Vase**
CPU	98.5 %	97.5 %	99.3 %
Memory	1.28 GB	1.32 GB	1.31GB
Storage	2.16 GB	2.169 GB	2.157 GB

Figure 9 presents the evaluation configurations for the POV-Ray applications. We instantiate 36 virtual machines that execute POV-Ray frames submitted via Application Deployer. The virtual machines are continuously monitored by Gmond. Thus, LoM2HiS has access to resource utilization during execution of the applications. Similarly, information about the time taken to render each frame in each virtual machine is also available to LoM2HiS. This information is generated by the application itself and is sent to a location where LoM2HiS can read it. As described in Figure 9, users supply the QoS requirements in terms of SLOs (step 1 in Figure 9). At the same time the images with the POV-Ray applications and input data (frames) can be uploaded to the front-end node. Based on the current system status, SLA negotiator establishes an SLA with the user. Description of the negotiation process and components is out of scope of this paper and is discussed by Brandic *et al.* [6]. Thereafter, VM deployer starts configuration and allocation of the required VMs whereas application deployer maps the tasks to the appropriate VMs (step 3). In step 4 the application execution is triggered.

6.3.1. Image Rendering Application Use-Case Results

We defined and used seven measurement intervals to monitor the POV-Ray applications during their executions. Table 4 shows the measurement intervals and the number of measurements made in each interval. The applications run for about 30 minutes for each measurement interval.

The 10 seconds measurement interval is a reference interval meaning the

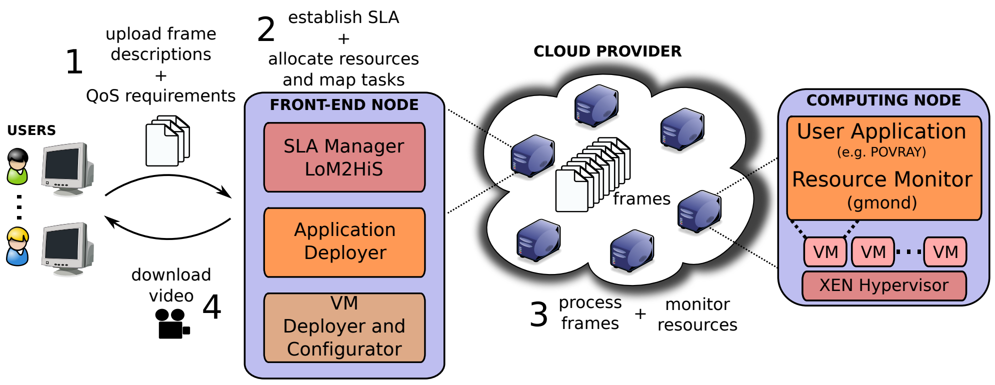


Figure 9: Pov-Ray Evaluation Configuration.

Table 4: Measurement Intervals.

Intervals	10s	15s	20s	25s	30s	60s	120s
Nr. of Measurements	180	120	90	72	60	30	15

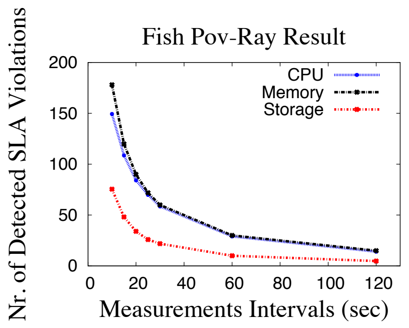
current interval used by the provider to monitor application executions on the Cloud resources. Its results show the present situation of the Cloud provider.

Figure 10 presents the achieved results of the three POV-Ray applications with varying characteristics in terms of frame rendering as explained in Section 6.3. We use the 36 virtual machines in our testbed to simultaneously execute the POV-Ray frames. The load-balancer integrated in the application deployer ensures that the frame executions are balanced among the virtual machines.

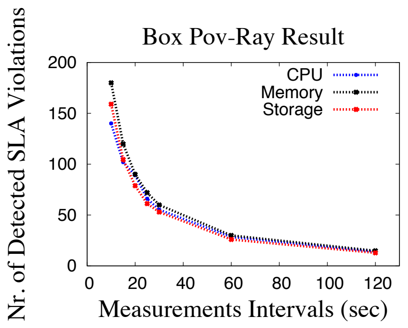
The *LoM2HiS* framework monitors the resource usage of each virtual machine to determine if the SLA objectives are met and reports violations otherwise. Since the load-balancer balances the execution of frames among the virtual machines, we plot in Figure 10 the average numbers of violations encountered in the testbed for each application with each measurement interval. We analyze and interpret these results in the next section.

6.3.2. Image Rendering Application Use-Case Results Analysis

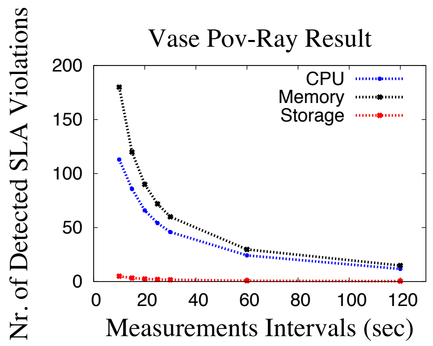
POV-Ray results presented in Figure 10 show that as the measurement interval increases, the number of detected SLA violation decreases. This effect is straightforward because with larger measurement interval the system misses detection of some SLA violations. The figures also reflect the resource consumption behavior of the POV-Ray applications.



(a) Fish.



(b) Box.



(c) Vase.

Figure 10: POV-Ray Experimentation Results.

We carried out an intrusiveness test in our testbed to find out the processing overhead of a measurement. This will determine the cost of taking measurements. Measurement processing includes monitoring of all the virtual machines, processing of monitored data, mapping of low-level metrics to high-level SLA, and evaluation of SLA objectives. Figure 11 presents the achieved result.

Figure 11 shows the amount of overhead found in the system and how they decrease as the measurement intervals increases. This means high cost for measurements with small intervals and low cost for measurement with larger intervals.

The cost of missing SLA violation detection is an economic factor, which depends on the SLA penalty cost agreed for the specific application and

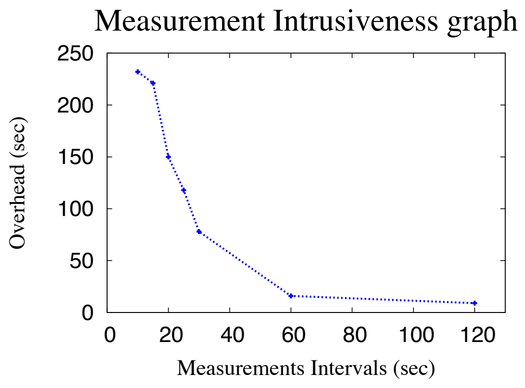


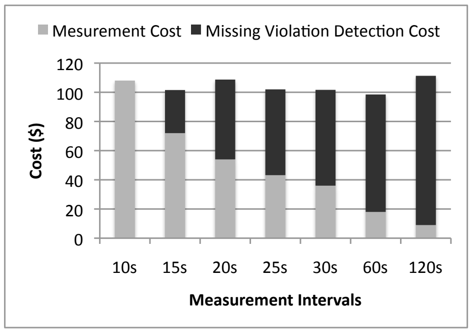
Figure 11: Intrusiveness Test Results.

the effects the violation will have to the provider for example in terms of reputation or trust issues.

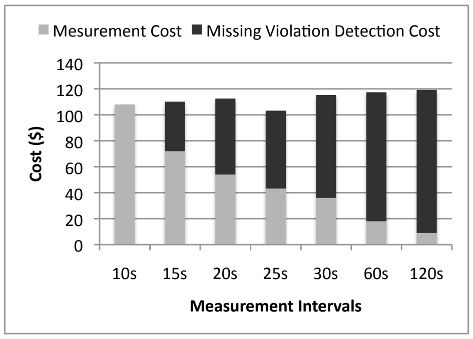
By applying the cost function presented in Section 6.2 to the achieved results of Figure 10, with a measurement cost of \$0.6 and missing violation cost of \$0.25, we achieve the monitoring costs presented in Figure 12. These cost values are example values for our experimental setup. They neither represents nor suggests any standard values. The approach used here is derived from the cost function approaches presented in literature [47, 48].

It should be noted about the results of Figure 12 that the reference measurement is assumed to capture all SLA violations for each application, thus it only incurs measurement cost. From the figures, it can be noticed on the one hand that the lower the number of measurements, the smaller the measurement cost and on the other hand, the higher the number of undetected SLA violations, the higher the cost of missing violations. This implies that to keep the detection cost low, the number of undetected SLA violations must be low.

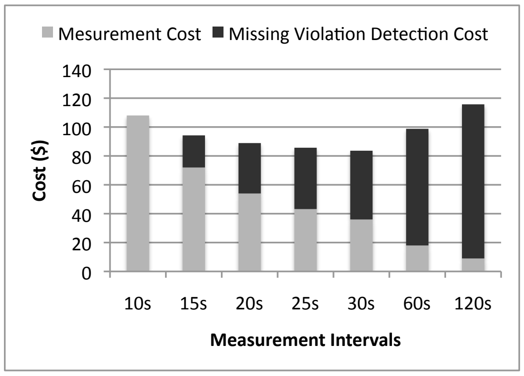
Considering the total cost of monitoring the fish POV-Ray application in Figure 12(a), it can be seen that the reference measurement is not the cheapest although it does not incur any cost of missing SLA violation detection. In this case the 60-second interval is the cheapest and in our opinion the most suited measurement interval for this application. In the case of box POV-Ray application the total cost of monitoring, as depicted graphically



(a) Fish.



(b) Box.



(c) Vase.

Figure 12: POV-Ray application cost relations.

in Figure 12(b), indicates that the lowest cost is incurred with the 25-second measurement interval. Thus we conclude that this interval is best suited for this application. Also from Figure 12(c), it is clear that the reference measurement by far is not the optimal measurement interval for the vase POV-Ray application. Thus, from the experiments the 30-second measurement interval is considered best suited for this application group.

Based on our experiments, it is observed that there is no best suited measurement interval for all applications. Depending on how steady the resource consumption is, the monitoring infrastructure requires different measurement intervals. Notwithstanding, definition of these intervals is important to allow estimation of impact of missed violations in applications. Note that the architecture can be configured to work with different intervals. In this case, specification of the measurement frequencies depends on policies agreed by customer and Cloud providers.

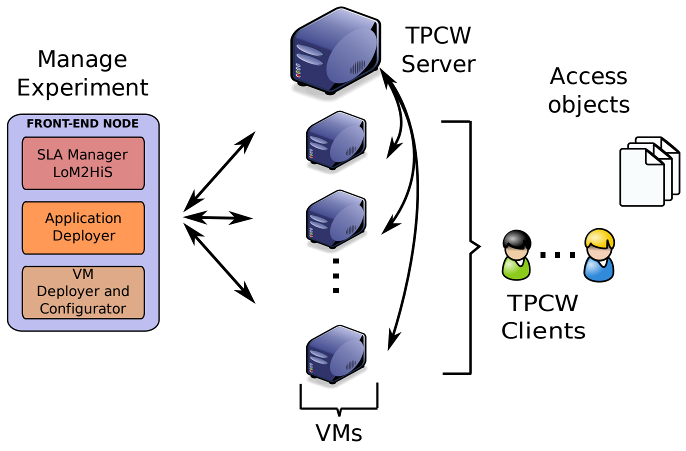


Figure 13: Web Application Evaluation Configuration.

6.4. Web Application Use-Case

As a web application, we performed experiments using the Java implementation[3] of the TPC-W Benchmark [49]. This application simulates the activties of a business oriented transactional web server. The workload used in the server exercises system components related to several issues commonly found in web environments, such as multiple on-line browser sessions, dynamic page generation with database access and update, transaction integrity, and simultaneous execution of multiple transaction types.

We configured TPC-W to run on the 36 VMs in our setup environment. One VM is used as the server and the other 35 VMs are used as clients as shown in Figure 13. The clients generate requests that are handled on the server. We use the LoM2HiS framework to monitor the server and to detect SLA violations.

The quality of service requirement of the web application depends on the amount of available CPU and memory resources. Thus, we define two SLA objectives for these resource parameters to ensure the performance of the application during its execution. The values of the SLA objectives are learned based on historical data and sample runs to examine the behavior of the application in terms of resource consumptions. For the CPU, we set a 10% threshold and for memory we set a 12% threshold. Utilization of resources above these thresholds indicates an SLA violation situation.

[3]http://tpcw.deadpixel.de/

6.4.1. *Web Application Use-Case Results*

The resource usage of the web application server in processing the requests generated by the clients is monitored by the LoM2HiS framework in order to detect and report the SLA violations. Like in the case of POV-Ray application, we experiment here with five measurement intervals to monitor the SLAs during the application execution. The web application is allowed to run for a total length of seven minutes. In this case, small measurement intervals are chosen considering the fact that web application behavior can change drastically within a period of seconds. Table 5 presents the achieved results.

Table 5: TPCW Experimentation Results.

Intervals	5s	10s	15s	20s	30s
Nr. of Measurements	84	42	28	21	14
Nr. of CPU violations detected	77	26	14	12	7
Nr. of Memory violations detected	75	41	26	19	12

Table 5 shows the number of measurements made with each interval and the number of SLA violations detected for the CPU and memory resources. Based on these results, we apply our cost function in the next section (result analysis) to determine the optimal measurement interval.

6.4.2. *Web Application Use-Case Results Analysis*

As presented in Table 5, the number of SLA violations detected decreases as the measurement interval time grows. This is an expected logical behaviour. Therefore, to find the optimal measurement interval we apply the cost function of Equation 2 on the achieved results.

In this use-case scenario, the cost of measurement is low considering the experimental setup shown in Figure 13. With the setup, the processing of client requests are performed on the TPC-W server, thus only this server is monitored to detect SLA violations. Therefore, there is a low overhead in monitoring this single machine. On the other side, the cost of missing SLA violation is high because the web application performance degrades very fast once the SLA objectives are violated, which can frustrate a customer waiting for a response of the application. e.g., waiting for a browser to load.

On the basis that the cost of measurement is low and the cost of missing SLA violation detection is high, we use \$0.15 as the measurement cost and

$0.30 as the cost of missing violation. Note that we use the 5s measurement interval as a reference interval, which means that it detects all SLA violation and acts as the current default measurement interval. Thus, it incurs only measurement cost and no cost for missing SLA violation detection. When these values are applied in the cost function, results depicted in Figure 14 are achieved.

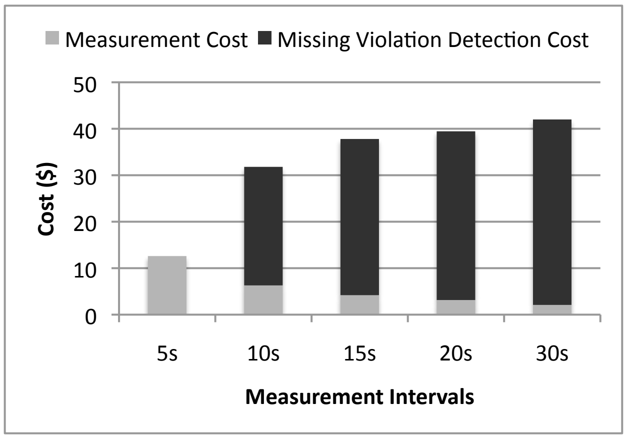


Figure 14: Web application cost relations.

The results show the total cost incurred by each of the measurement intervals. The cost of missing SLA violation detection increases as the measurement interval increases. This is caused by the fact that the larger the measurement interval, the lower the number of measurements made and the higher the number of missed SLA violations. Failure to detect SLA violations means costly SLA penalties for the provider and poor performance of the application.

Therefore, from our experiments we could not find a larger better measurement interval than the 5 seconds reference measurement interval, what confirms our assumptions that web applications are highly sensitive and should be monitored at small interval to ensure their quality of service. Furthermore, there can be a surge in clients request of a web application within short period of time, thus the monitoring mechanism should be able to detect such situations.

The whole set of experiments presented in this section clearly demonstrate the need for fine-tunning of monitoring systems to the specific requirements of Cloud applications. However, different applications have needs for different measurement intervals, and even though some applications are more stable than other in terms of resource requirements, defining methods for finding the optimal measurement interval of each application is a non-trivial problem, and an interesting research topic that we plan to address in the future.

7. Conclusion and Future Work

Flexible and reliable management of SLA agreements represents an open research issue in Cloud computing. Advantages of flexible and reliable Cloud infrastructures are manifold. For example, prevention of SLA violations avoids unnecessary penalties providers have to pay in case of violations. Moreover, based on flexible and timely reactions to possible SLA violations, interactions with users can be minimized. In this paper we presented *DeSVi*—the novel architecture for monitoring and detecting SLA violations in Cloud computing infrastructures. The main components of our architecture are *the automatic VM deployer*, responsible for the allocation of resources and for mapping of tasks, *application deployer*, responsible for the execution of user applications, and *LoM2HiS framework*, which monitors the execution of the applications and translates low-level metrics into high-level SLAs.

We evaluated our system using two use-case scenarios consisting of an image rendering application and a transactional application. In the first use-case scenario we use a heterogeneous workload of three POV-Ray applications. From our experiments with these applications, we observed that there is no particular optimal suited measurement interval for all applications. It is easier to identify the intervals for applications with steady resource consumption, such as the 'vase' POV-Ray animation. However, applications with variable resource consumption require dynamic measurement intervals.

The experiments of the second use-case scenario use workloads generated by the TPC-W benchmark. Based on the experimental results, we noticed that smaller measurement intervals are preferable than larger ones for this application domain due to their sensitive nature and failure intolerance.

The currently proposed system is capable of monitoring a single Cloud data center. In the future, we will extend it with the capability to manage a Cloud environment with multiple data centers. Thus, we will apply a decentralization approach whereby the proposed system is installed on each data center. The scalable communication mechanism realized in LoM2HiS framework will be used to allow communication between data centers.

Based on our investigation of optimal measurement intervals, we will incorporate into DeSVi a knowledge database to propose reactive actions to prevent or correct the SLA violation situations. Knowledge of optimal measurement intervals allows best reactive actions, which contributes to our vision of flexible and reliable on-demand computing via fully autonomic Cloud infrastructures.

Acknowledgments

This work is supported by the Vienna Science and Technology Fund (WWTF) under grant agreement ICT08-018 Foundations of Self-governing ICT Infrastructures (FoSII) and Australian Research Council. This paper is a substantially extended version of a CloudComp 2010 paper [20]. The experiments were performed in the High Performance Computing Lab at Catholic University of Rio Grande do Sul (LAD-PUCRS) Brazil.

References

[1] R. Buyya, C. S. 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 25 (6) (2009) 599–616.

[2] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, L. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, F. Galan, The RESERVOIR model and architecture for open federated cloud computing, IBM Journal of Research and Development 53 (4) (2009) Paper 4.

[3] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, D. Zagorodnov, The Eucalyptus open-source cloud-computing system, in: Proceedings of the 9th International Symposium on Cluster Computing and the Grid (CCGRID'09), 2009.

[4] P. Balakrishnan, T. S. Somasundaram, SLA enabled CARE resource broker, Future Generation Computer Systems 27 (3) (2011) 265 – 279.

[5] A. Litke, K. Konstanteli, V. Andronikou, S. Chatzis, T. Varvarigou, Managing service level agreement contracts in OGSA-based grids, Future Generation Computer Systems 24 (4) (2008) 245 – 258.

[6] I. Brandic, Towards self-manageable cloud services, in: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09), 2009.

[7] FoSII, Foundations of self-governing infrastructures. `http://www.infosys.tuwien.ac.at/linksites/FOSII/index.html`.

[8] R. N. Calheiros, R. Buyya, C. A. F. De Rose, Building an automated and self-configurable emulation testbed for grid applications, Software: Practice and Experience 40 (5) (2010) 405–429.

[9] W.-C. Chung, R.-S. Chang, A new mechanism for resource monitoring in grid computing, Future Generation Computer Systems 25 (1) (2009) 1–7.

[10] S. Reyes, C. Muoz-Caro, A. Nio, R. Sirvent, R. Badia, Monitoring and steering grid applications with grid superscalar, Future Generation Computer Systems 26 (4) (2010) 645 – 653.

[11] A. D'Ambrogio, P. Bocciarelli, A model-driven approach to describe and predict the performance of composite services, in: Proceedings of the 6th International Workshop on Software and Performance (WOSP'07), 2007.

[12] D. Gunter, B. Tierney, B. Crowley, M. Holding, J. Lee, Netlogger: A toolkit for distributed system performance analysis, in: Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'00), 2000.

[13] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, H. Casanova, Characterizing resource availability in enterprise desktop grids, Future Generation Computer Systems 23 (7) (2007) 888 – 903.

[14] C. Li, L. Li, Competitive proportional resource allocation policy for computational grid, Future Generation Computer Systems 20 (6) (2004) 1041 – 1054.

[15] J. L. Berral, I. Goiri, R. Nou, F. Juliá, J. Guitart, R. Gavaldá, J. Torres, Towards energy-aware scheduling in data centers using machine learning, in: 1st International Conference on Energy-Efficiency Computing and Networking, Passau, Germany, 2010.

[16] A. Beloglazov, R. Buyya, Energy efficient resource management in virtualized cloud data centers, in: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 826–831.

[17] A. Celesti, F. Tusa, M. Villari, A. Puliafito, How to enhance cloud architectures to enable cross-federation, in: IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010, 2010, pp. 337 –345.

[18] A. Celesti, F. Tusa, M. Villari, A. Puliafito, Three-phase cross-cloud federation model: The cloud sso authentication, in: Second International Conference on Advances in Future Internet (AFIN), 2010, 2010, pp. 94 –101.

[19] V. C. Emeakaroha, I. Brandic, M. Maurer, S. Dustdar, Low level metrics to high level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments, in: Proceedings of the High Performance Computing and Simulation Conference (HPCS'10), 2010.

[20] V. C. Emeakaroha, R. N. Calheiros, M. A. S. Netto, I. Brandic, C. A. F. De Rose, DeSVi: An architecture for detecting SLA violations in cloud computing infrastructures, in: Proceedings of the 2nd International ICST Conference on Cloud Computing (CloudComp'10), 2010.

[21] W. Fu, Q. Huang, GridEye: A service-oriented grid monitoring system with improved forecasting algorithm, in: Proceedings of the 5th International Conference on Grid and Cooperative Computing Workshops (GCCW'06), 2006.

[22] T. Wood, P. J. Shenoy, A. Venkataramani, M. S. Yousif, Sandpiper: Black-box and gray-box resource management for virtual machines, Computer Networks 53 (17) (2009) 2923–2938.

[23] M. Boniface, S. C. Phillips, A. Sanchez-Macian, M. Surridge, Dynamic service provisioning using GRIA SLAs, in: Proceedings of the 5th International Workshops on Service-Oriented Computing (ICSOC'07), 2007.

[24] B. Koller, L. Schubert, Towards autonomous SLA management using a proxy-like approach, Multiagent Grid Systems 3 (3) (2007) 313–325.

[25] H. M. Frutos, I. Kotsiopoulos, BREIN: Business objective driven reliable and intelligent grids for real business, International Journal of Interoperability in Business Information Systems 3 (1) (2009) 39–42.

[26] M. Comuzzi, C. Kotsokalis, G. Spanoudkis, R. Yahyapour, Establishing and monitoring SLAs in complex service based systems, in: Proceedings of the 7th International Conference on Web Services (ICWS'09), 2009.

[27] G. Dobson, A. Sanchez-Macian, Towards unified QoS/SLA ontologies, in: Proceedings of the 2006 IEEE Services Computing Workshops (SCW'06), 2006.

[28] F. Rosenberg, C. Platzer, S. Dustdar, Bootstrapping performance and dependability attributes of web services, in: Proceedings of the 4th International Conference on Web Services (ICWS'06), 2006.

[29] Brein, Business objective driven reliable and intelligent grids for real business.
`http://www.eu-brein.com/`.

[30] J. O. Kephart, D. M. Chess, The vision of autonomic computing, IEEE Computer 36 (1) (2003) 41–50.

[31] M. Maurer, I. Brandic, V. C. Emeakaroha, S. Dustdar, Towards knowledge management in self-adaptable clouds, in: Proceedings of the 4th International Workshop of Software Engineering for Adaptive Service-Oriented Systems (SEASS'10), 2010.

[32] M. L. Massie, B. N. Chun, D. E. Culler, The Ganglia distributed monitoring system: Design, implementation and experience, Parallel Computing 30 (7) (2004) 817–840.

[33] D. Kyriazis, K. Tserpes, A. Menychtas, A. Litke, T. Varvarigou, An innovative workflow mapping mechanism for grids in the frame of quality of service, Future Generation Computer Systems 24 (6) (2008) 498 – 511.

[34] JMS, Java messaging service, `http://java.sun.com/products/jms/`.

[35] ActiveMQ, Messaging and integration pattern provider.
`http://activemq.apache.org/`.

[36] E. Elmroth, J. Tordsson, A grid resource broker supporting advance reservations and benchmark-based resource selection, in: Proceedings of the Workshop on State-of-the-art in Scientific Computing (PARA'04), 2004.

[37] D. Abramson, R. Buyya, J. Giddy, A computational economy for grid computing and its implementation in the Nimrod-G resource broker, Future Generation Computer Systems 18 (8) (2002) 1061–1074.

[38] K. Krauter, R. Buyya, M. Maheswaran, A taxonomy and survey of grid resource management systems for distributed computing, Software: Practice and Experience 32 (2) (2002) 135–164.

[39] H. Casanova, G. Obertelli, F. Berman, R. Wolski, The AppLeS parameter sweep template: User-level middleware for the Grid, in: Proceedings of the Supercomputing (SC'00), 2000.

[40] B. Sotomayor, R. S. Montero, I. M. Llorente, I. Foster, Virtual infrastructure management in private and hybrid clouds, IEEE Internet Computing 13 (5) (2009) 14–22.

[41] Oracle, Oracle virtualization.
`http://www.oracle.com/technologies/virtualization`.

[42] S. Venugopal, J. Broberg, R. Buyya, OpenPEX: An open provisioning and execution system for virtual machines, in: Proceedings of the 17th International Conference on Advanced Computing and Communications (ADCOM'09), 2009.

[43] SAX, Simple API for XML.
`http://sax.sourceforge.net/`.

[44] ESPER, Event stream processing.
`http://esper.codehaus.org/`.

[45] K. Lee, N. W. Paton, R. Sakellariou, A. A. F. Alvaro, Utility driven adaptive worklow execution, in: Proceedings of the 9th International Symposium on Cluster Computing and the Grid (CCGrid'09), 2009.

[46] S. Seneviratne, D. C. Levy, Task profiling model for load profile prediction, Future Generation Computer Systems 27 (3) (2011) 245 – 255.

[47] C. B. Lee, A. Snavely, On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions, International Journal of High Performance Computer Applications 20 (4) (2006) 495–506.

[48] C. S. Yeo, R. Buyya, Pricing for utility-driven resource management and allocation in clusters, International Journal of High Performance Computer Applications 21 (4) (2007) 405–418.

[49] D. Menascé, TPC-W: A benchmark for e-commerce, IEEE Internet Computing 6 (3) (2002) 83–87.