DeSVi: An Architecture for Detecting SLA Violations in Cloud Computing Infrastructures

Vincent C. Emeakaroha[1], Rodrigo N. Calheiros[3], Marco A. S. Netto[2], Ivona Brandic[1], and César A. F. De Rose[2]

[1] Vienna University of Technology, Vienna, Austria
{vincent, ivona}@infosys.tuwien.ac.at

[2] PUCRS, Faculty of Informatics, Porto Alegre, Brazil
{marco.netto,cesar.derose}@pucrs.br

[3] Department of Computer Science and Software Engineering, University of Melbourne, Australia
rodrigoc@csse.unimelb.edu.au

Abstract. Cloud computing is a promising paradigm for the implementation of scalable on-demand computing infrastructures. Self-manageable Cloud infrastructures are required in order to comply with users' requirements specified by Service Level Agreements (SLAs) on one hand and to minimize user interactions with the system on the other hand. Adequate SLA monitoring strategies and timely detection of possible SLA violations represent challenging research issues. In this paper we present DeSVi—an architecture for detecting SLA violations through resource monitoring in Cloud computing infrastructures. Based on the user requests DeSVi allocates necessary resources for a requested service and arranges its deployment on a virtualized environment. Resources are monitored using an efficient framework that is also capable of mapping low-level resources metrics to user-defined SLAs. The detection of possible SLA violations is based on the predefined service level objectives and we utilize knowledge databases to manage the SLA violations. Knowledge databases are implemented using techniques like case-based reasoning, where reactive actions are defined based on the past system experience. For the evaluation of our approach, we developed image rendering services, which exhibit heterogeneous workloads for investigating the optimal monitoring interval of SLA parameters. The achieved results show that our architecture is able to monitor and prevent SLA violations considering different costs, measurement intervals, and heterogeneous workloads.

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

1 Introduction

Cloud computing represents a novel paradigm for implementation of scalable computing infrastructures combining concepts from virtualization, distributed application design, Grid, and enterprise IT management [6, 29, 31]. Service provisioning in the Cloud relies on Service Level Agreements (SLAs) representing a contract signed between the customer and the service provider including the non-functional requirements of the service specified as Quality of Service (QoS). 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 one hand, prevention of SLA violations avoids costly penalties provider has to pay in case of violations. On the other hand, based on flexible and timely reactions to possible SLA violations, users 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 [4, 7, 16], there is a lack of adequate monitoring infrastructures, which can predict possible SLA violations. Most of the available systems either rely on Grid or service-oriented infrastructures [11], which are not directly compatible to Clouds due to the difference of resource usage model, or are network-oriented monitoring infrastructures [20]. In our previous work we devised a novel concept for mapping low-level resource metrics to high-level SLAs—*LoM2HiS* [14], where measured metrics like system up and down time can be easily translated to high-level SLAs (e.g. system availability). Thus, *LoM2HiS* facilitates efficient monitoring of Cloud infrastructures and early detection of possible SLA violations. Moreover, LoM2HiS framework enables user-driven mappings between the resource metric and SLA parameters by utilizing 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. Too frequent measurement intervals may negatively affect the overall system performance, whereas too infrequent measurement intervals may cause heavy SLA violations. In this paper we extend our work on *LoM2HiS* and devise a novel architecture suitable for detection of SLA violations through resource monitoring in Cloud computing infrastructures—the *DeSVi* architecture.

The main components of the DeSVi architecture are: (i) *the automatic VM deployer*, (ii) *application deployer*, and (iii) *the LoM2HiS framework*. Based on the user requests the *automatic VM deployer* allocates necessary resources for the requested service and arranges its deployment on a virtual machine (VM). After the service deployment *LoM2HiS framework* monitors VMs and translates the low-level metrics into the high-level SLAs as specified by the users using DSLs. We utilize knowledge databases for the evaluation of the monitored SLAs parameters. Knowledge databases are implemented using traditional knowledge management techniques like case-based reasoning, which allows prediction of SLA violation based on previous system experience. For the evaluation of our approach, we developed image rendering services based on POV-Ray[4] [19]. POV-Ray, together with our application deployer is suitable for the evaluation of

[4] POV-Ray —The Persistence of Vision Raytracer: http://www.povray.org

Cloud-like applications with characteristics such as client-server architecture, reconfigurable load, and parallel execution mode via parameter sweeping [8]. It can be seen as a Cloud service similar to Animoto[5] for raytracing-based videos.

The main contributions of the paper are: (i) definition of the motivation scenario for the development of the architecture for detecting SLA violations, (ii) conceptual design of the *DeSVi* architecture for prediction of SLA violations, (iii) discussion on the implementation choices for the *DeSVi*, and (iv) evaluation of the architecture using a real-world application for image rendering.

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 the future work.

2 Related Work

We classify our related work into (i) resource monitoring [18, 20, 36], (ii) SLA management including violation detection [3, 10, 12, 17, 23], and (iii) mapping techniques of monitored metrics to SLA parameters [11, 32]. 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.* [18] 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.* [20] 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.* [36] developed a system, called Sandpiper, which automates the process of monitoring and detecting hotspots and remapping/reconfiguring VMs whenever necessary. Their monitoring system reminds ours 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 allowing the mapping of low level metrics, such as CPU and memory, to high-level SLA parameters, such as response time.

Boniface *et al.* [3] 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 is limited to Grid

[5] Animoto website: http://animoto.com/

environments. Moreover, they do not detail how the low-level metric are monitored and mapped to high-level SLAs. Koller *et al.* [23] 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. Frutos *et al.* [17] discuss the main approach of the EU project BREIN [5] 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.* [12] present a unified QoS ontology applicable to QoS-based Web services selection, QoS monitoring, and QoS adaptation. Comuzzi *et al.* [10] 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.

Rosenberg *et al.* [32] 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 necessary 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 are major research challenges in Cloud-like environments [6, 22]. We are currently developing an infrastructure called FoSII (Foundations of Self-governing Infrastructures), which proposes models and concepts for autonomic SLA management and enforcement in the Cloud. The FoSII infrastructure is capable of managing the whole lifecycle of self-adaptable Cloud services [4].

The essence of using SLA in Cloud business is 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 costly penalties and increase their profit, we devised the Low Level Metrics to High Level SLA—*LoM2HiS framework* [14], which is a building block of the FoSII infrastructure for monitoring Cloud re-

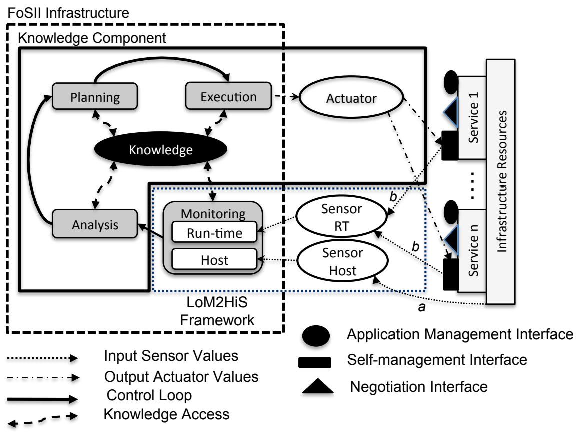


Fig. 1. FoSII Infrastructure Overview.

sources, mapping the low-level resource metrics to high-level SLA parameters, and detecting SLA violations as well as future SLA violation threats so as to react before actual SLA violations occur.

3.1 FoSII Infrastructure Overview

Figure 1 depicts the components of the FoSII infrastructure. 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 [4]. 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.

Logically, FoSII infrastructure consists of multiple components working together to achieve a common goal. In this paper we focus on the knowledge management component and the LoM2HiS framework since they are responsible for system monitoring and detection of SLA violations.

3.2 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

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)	$\geq 99\%$

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) as 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 Ganglia [27]. 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 SLA parameters. 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* and is defined by the following the mapping rule:

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

The mapping rules are defined by the provider using appropriate Domain Specific Languages (DSLs). 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 3.3, is part of our ongoing work and includes sophisticated methods for the system state management.

As described in a previous work [14], we implemented a highly scalable framework for mapping Low Level Resource Metrics to High Level SLA Parameters (LoM2HiS framework) facilitating the exchange of large numbers of messages. We designed and implemented a communication model based on the Java Messaging Service (JMS) API [21], which is a Java Message Oriented Middleware API for sending messages between two or more clients. We use Apache ActiveMQ as a JMS provider that manages sessions and queues.

3.3 Knowledge Databases

For the decision making we use knowledge databases proposing the reactive actions by utilizing case-based reasoning [28]. *Case-Based Reasoning (CBR)* is

```
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. )
```

Fig. 2. Case Based-Reasoning example.

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) retrieve the most similar case or cases to the new one, (ii) reuse the information and knowledge in the similar case(s) to solve the problem, (iii) revise the proposed solution, and (iv) retain 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 concerned (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* and *passive* [28]. 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.

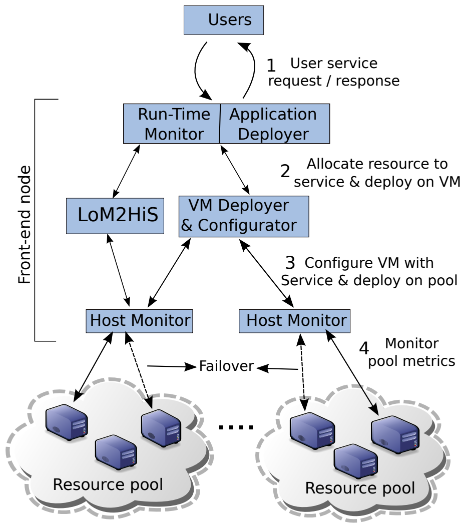

Users
1 User service request / response
Run-Time Monitor
Application Deployer
2 Allocate resource to service & deploy on VM
LoM2HiS
VM Deployer & Configurator
Front-end node
3 Configure VM with Service & deploy on pool
Host Monitor
Host Monitor
4 Monitor pool metrics
Failover
Resource pool
Resource pool

Fig. 3. Overview of the system architecture and component's interaction.

4 DeSVI Architecture

In this section we describe in details the *DeSVi architecture*, its components, and how the components interact with one another. The proposed architecture is designed to handle the complete service provisioning management lifecycle in Cloud environments. The service provisioning lifecycle includes activities such as SLA negotiation, resource allocation to tasks, resource monitoring, and SLA violation detection. Figure 3 depicts the architecture.

The topmost layer represents the users (customers) who request service provisioning (step 1 in Figure 3) from 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 resources for the requested service and arranges its deployment on VMs (step 2). The deployment of VMs and environment configuration are performed by AEF (Automated Emulation Framework) [7] (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.

In Figure 3 the arrow termed *Failover* indicates redundancy in the monitoring mechanism. The host monitor is designed to use monitoring agents like Gmond from Ganglia project [27], 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

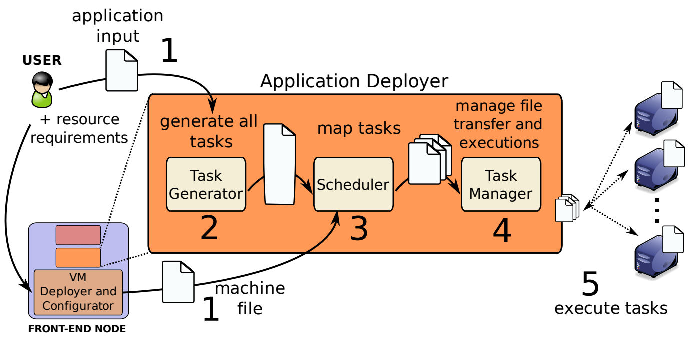


Fig. 4. Application Deployer.

if necessary but it can obviate the problem of a bottleneck master node for accessing the monitored metrics of the resource pool. In the following sections we explain all components 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 [1,13,24]. However, compared to brokers, the Application Deployer has more knowledge and control on the application tasks, being able to perform application-level scheduling, in particular for parameter sweeping executions [9]. It 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.

Figure 4 illustrates the main modules of the Application Deployer. The *task generator* 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). In our current implementation, the scheduler places an equal number of tasks for each available VM in order to balance the load. The scheduler can also be configured to use a dynamic load balancing, in which each VM asks for tasks whenever there is CPU available. A list of task for each VM is transferred from the *task manager* to the VMs (step 4). The *task manager* is also responsible for triggering the executions on VMs (step 5) and collecting the results when tasks complete execution.

4.2 Automated Emulation Framework

The Automated Emulation Framework (AEF) was originally conceived for automated configuration and execution of emulation experiments [7]. Nevertheless, it also can be used to set up arbitrary virtual environments by not activating

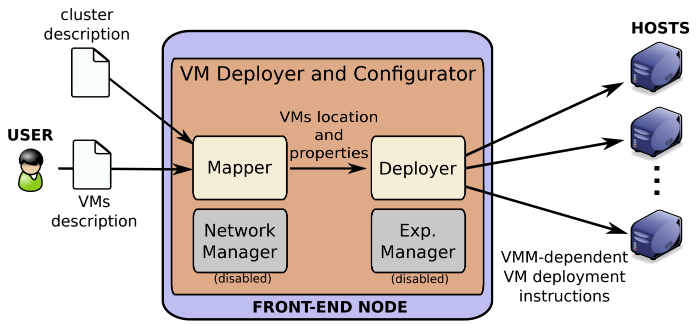


Fig. 5. AEF Framework.

the emulated wide-area network support. In the latter case AEF works as a virtualized infrastructure manager, similarly to tools like OpenNebula [34], Oracle VM Manager [30], and OpenPEX [35].

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.

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 by either 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 by the former approach. 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 our proposed architecture of Figure 3 is done by the LoM2HiS framework. Figure 6 presents the architecture of the LoM2HiS framework. 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 mappings rules for the LoM2HiS mappings (step 1 in Figure 2) using Domain Specific Languages (DSLs). 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

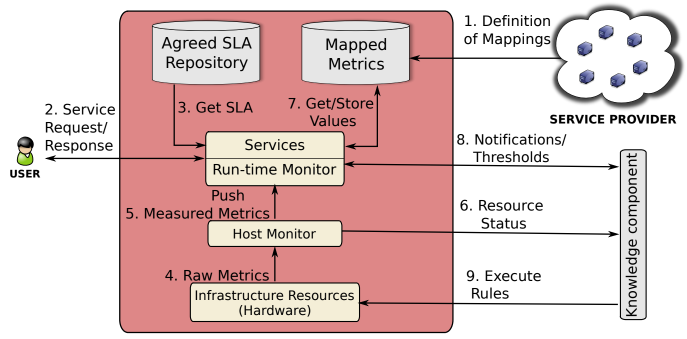


Fig. 6. LoM2HiS framework.

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 designed communication model.

Upon reception of the measured metrics, the run-time monitor maps the low-level metrics based on predefined mapping rules to form an equivalent 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 preventive 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 preventive action to avert future SLA violation threats based on the resource status (step 6) and defined rules. The knowledge component's decisions (e.g. assign more CPU to a virtual host) are executed on the infrastructure resources (step 9).

5 Implementation Issues

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 only well-established and tested open source tools in the implementations.

5.1 Application Deployer

The Application Deployer is written in Python 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 script to split the work to be done into lists of tasks—one list per VM. For a rendering application, for instance, such a script includes a list of frames and the command to render them. The division of work per VM is performed by the Application Deployer's scheduler as described in Section 4.1.

Once the mapping of tasks to VMs is completed, the Application Deployer uses the *scp* command, a standard Linux 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 then responsible for triggering task execution on each VM specified in the machine file. During the user application execution, the Application Deployer generates log files with the time required to execute each task. After tasks complete execution, results are transferred to the front-end node via *scp*.

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. The former description may contain different subnetworks, connected by emulated WANs (e.g. virtual links with high latency and low bandwidth). Nevertheless, this feature is not required in the current experiments where the target environment is a virtualized cluster. Thus, from AEF point of view the environment comprises a set of virtual machines in a single Gigabit Ethernet network.

Such an environment is described in the XML file and processed by AEF. Once the cluster and virtual environment 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 standalone version of the Deployer is used, because it does not require external tools or systems for its execution. This deployer is written in Java and is part of AEF source code. It works as follows. First, a base image file of the virtual machines is copied, via *scp*, to each cluster node that takes part in the experiment, as determined by the Mapper. This image contains all the software and configuration required by the experiment. After the base image is copied to each machine, it is replicated so that an image for each virtual machine is created on each node. These new images are configured with VM-specific settings required by the experiment, such as hostname and static IP address. Finally, virtual machines are created on each host from each image file replicated in the previous step.

5.3 LoM2HiS Components

The *host monitor* implementation uses the standalone Gmond module from the Ganglia open source project [27] as monitoring agent. 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 [33] 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 [2] for this purpose.

The *run-time monitor* implementation passes the received metric-value pairs into ESPER engine [15], 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

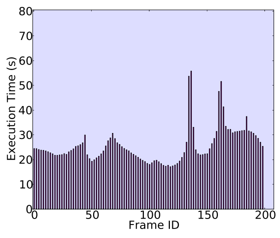
In this section we discuss the evaluation of the *DeSVi architecture* and SLA violation detection strategy. In Section 6.1 we describe our experimental set up and in Section 6.2 we discuss the achieved experimental results.

6.1 Experimental Configuration

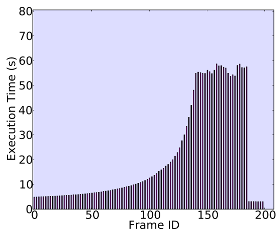
The virtualized cluster used in the experiments comprises five Pentium 4 2.8GHz machines with 1MB of cache and 2.5GB of RAM memory running Xen 3.4.0 on top of Oracle VM Server. One of the machines is used as the system front-end node. It runs AEF, the application deployer, and LoM2HiS. The other machines are used as cluster computing nodes and run virtual machines deployed via AEF as described previously. AEF is used to deploy and configure two VMs on each machine, with the VMs allocating all the available resources from the host.

Virtual machines are worker nodes able to execute POV-Ray applications submitted via Application Deployer. In order to achieve heterogeneous load, three POV-Ray workloads are tested, each one with a different characteristic of time for rendering frames, as illustrated in Figure 7:

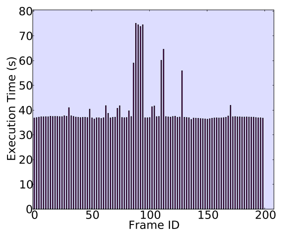
– **Fish:** rotation of a fish on water. Time for rendering frames is variable.
– **Box:** approximation of a camera to an open box with objects inside. Time for rendering frames increases during execution.



(a) Fish.



(b) Box.



(c) Vase.

Fig. 7. Behavior of execution time for each POV-Ray application.

– **Vase:** rotation of a vase with mirrors around. Time for processing different frames is constant.

Three SLA documents are negotiated 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 2 presents the SLA objectives for each application. Based on these SLA objectives, the applications are monitored to detect SLA violations. These violations may happen 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 2. SLA objective definitions for the three POV-Ray applications.

SLA Parameter	**Fish**	**Box**	**Vase**
CPU	20%	15%	10%
Memory	297MB	297MB	297MB
Storage	2.7GB	2.6GB	2.5GB

The virtual machines are continuously monitored by Gmond. Thus, LoM2HiS has access to resource utilization during execution of 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. Figure 8 illustrates our testbed and the information flow from the moment the users negotiate SLAs with the provider, until the moment the frames are processed and returned to the users in a form of a movie. As described in Figure 8, users supply the QoS requirements in terms of SLOs (step 1 in Figure 8). 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.* [4]. Thereafter, VM deployer starts configuration and allocation of the required VMs whereas

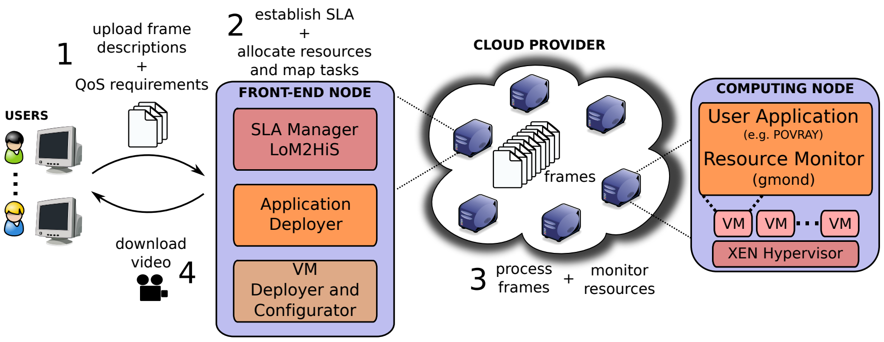


Fig. 8. Experimental testbed and information flow.

application deployer maps the tasks to the appropriate VMs (step 3). In step 4 the application execution is triggered.

6.2 Results and Analysis

The experiments are carried out using three POV-Ray applications with varying characteristics in terms of frame rendering as explained in Section 6.1. The applications are provisioned using our described testbed. Application execution is monitored by LoM2HiS using five different monitoring intervals. Table 3 presents the results of the experiments for 20-minute execution time for each application and monitoring interval.

Table 3. Experimentation results.

	Intervals	5s	10s	20s	30s	1min	2min
	Nr. of Measurements	240	120	60	40	20	10
Fish POV-Ray Application							
		Nr. of Violations					
SLA Parameter	**CPU**	105	90	42	28	16	8
	Memory	100	91	43	29	16	9
	Storage	97	91	43	29	17	9
Box POV-Ray Application							
SLA Parameter	**CPU**	85	37	22	17	12	7
	Memory	70	38	23	18	12	8
	Storage	65	38	23	18	11	8
Vase POV-Ray Application							
SLA Parameter	**CPU**	50	18	13	10	8	5
	Memory	45	19	14	11	9	6
	Storage	40	19	14	11	9	6

To determine the best monitoring interval for each application, we define a cost function (C) taking into account the cost of doing the measurement and the cost for the provider if it fails to detect SLA violations. The ideas of defining this cost functions are derived from utility functions discussed in [26]. The reference measurements taken with 5 seconds interval as shown in Table 3 for the three applications with the same execution length of 20 minutes form the basis of the cost function. The measurements represent the results of the current interval used by the provider for the monitoring of service and resources in the Cloud environment. Equation 2 presents the cost function.

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

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 cost of the measurement is defined by considering the intrusiveness of the measurements on the overall performance of the system. Based on our system architecture and intrusive test performed, we were able to find out that measurements have minimal effects on the computing nodes. This is because measurements and their processing take place in the front-end node while the services are hosted in the computing node. The monitoring agents running on computing nodes have minimal impact on resource consumption. The cost of missing violations is a parameter that is set by the provider considering the applicable penalties for SLA violations and historical data of SLA enactment processes.

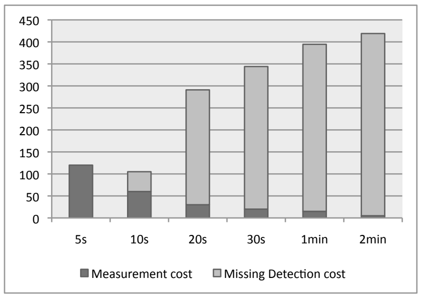
Applying the cost function on the achieved results of Table 3, with a measurement cost of 0.5 dollar and missing violation cost of 1.5 dollar, we achieve the monitoring costs presented in Table 4. The cost values specified here are derived based on the cost function approaches presented in literature [25, 37].

Table 4. Monitoring cost.

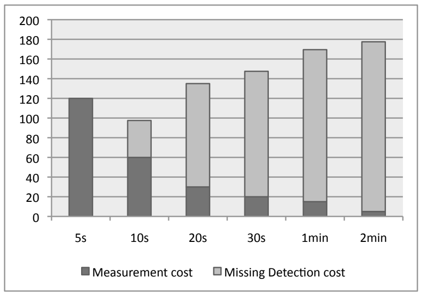
Intervals / SLA Parameter	**Reference**	**10s**	**20s**	**30s**	**1min**	**2min**
Fish POV-Ray Application						
CPU	0	22.5	94.5	115.5	133.5	145.5
Memory	0	13.5	85.5	106.5	126	136.5
Storage	0	9	81	102	120	132
Cost of Measurements	120	60	30	20	15	5
Total Cost	120	105	291	344	394.5	419
Box POV-Ray Application						
CPU	0	19.5	42	49.9	58.5	64.5
Memory	0	10.5	33	40.5	49.5	55.5
Storage	0	9	81	102	120	132
Cost of Measurements	120	60	30	20	15	5
Total Cost	120	97.5	135	147.5	169.5	177.5
Vase POV-Ray Application						
CPU	0	10.5	18	22.5	25.5	30
Memory	0	6	13.5	18	21	25.5
Storage	0	7.5	15	19.5	22.5	27
Cost of Measurements	120	60	30	20	15	5
Total Cost	120	84	76.5	80	84	87.5

The monitoring cost presented in Table 4 represents the cost of measurement for each frequency and for missing to detect SLA violation situation for each application. The reference measurement captures all SLA violations for each application, thus it only incurs measurement cost. Taking a closer look at Table 4, it is clear that the values of the shorter measurement interval are closer to the reference measurement than those of the longer measurement interval. This is attributed to our novel architecture design, which separates management activities from computing activities in our Cloud testbed.

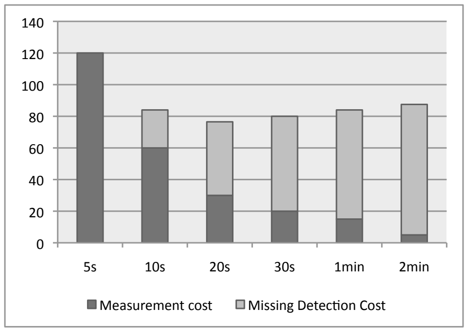
The relations of the measurement cost and the cost of missing SLA violation detection is graphically depicted in Figure 9 for the three POV-Ray applications. From the figures, it can be noticed in terms of measurement cost that the longer the measurement interval, the smaller the measurement cost and in terms of detection cost, the higher the number of missed SLA violation detection, the higher the detection cost rises. This implies that to keep the detection cost low, the number of missed SLA violation must be low.



(a) Fish.



(b) Box.



(c) Vase.

Fig. 9. POV-Ray application cost relations.

Considering the total cost of monitoring the fish POV-Ray application in Table 4 and Figure 9(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 10-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 shown in Table 4 and depicted graphically in Figure 9(b), indicates that the lowest cost is incurred with the 10-second measurement interval. Thus we conclude that this interval is best suited for this application. Also from Table 4 and Figure 9(c), it is clear that the reference measurement by far does not have the optimal measurement interval for the vase POV-Ray application. Based on the application behavior, longer measurement intervals are better fitted than shorter ones. Therefore, in this case the 20-second measurement interval is best suited for the considered scenario.

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. 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 users and providers.

7 Conclusion and Future Work

Flexible and reliable management of SLA agreements represents an open research issue in Cloud computing infrastructures. Advantages of flexible and reliable Cloud infrastructures are manifold. For example, preventions of SLA violations avoids unnecessary penalties provider has 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 an image rendering service based on POV-Ray with heterogeneous workloads. From our experiments we observed that there is no particular 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. Our architecture can be extended to tackle such applications, which will be the scope of our future work.

Besides our investigation on dynamic measurement intervals, we will evaluate the influence of such intervals in the quality of the reactive actions proposed by the knowledge database. If the effects of measurement intervals are known, best reactive actions may be taken, contributing to our vision of flexible and reliable on-demand computing via fully autonomic cloud infrastructures.

Acknowledgment

This work is supported by the Vienna Science and Technology Fund (WWTF) under grant agreement ICT08-018 Foundations of Self-governing ICT Infrastructures (FoSII).

References

1. D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.
2. ActiveMQ. Messaging and integration pattern provider. http://activemq.apache.org/.
3. M. Boniface, S. C. Phillips, A. Sanchez-Macian, and M. Surridge. Dynamic service provisioning using GRIA SLAs. In *International Workshops on Service-Oriented Computing (ICSOC'07)*, 2007.
4. I. Brandic. Towards self-manageable cloud services. In *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, 2009.
5. Brein. Business objective driven reliable and intelligent grids for real business. http://www.eu-brein.com/.
6. R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and 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):599–616, 2009.
7. R. N. Calheiros, R. Buyya, and C. A. F. D. Rose. Building an automated and self-configurable emulation testbed for grid applications. *Software: Practice and Experience*, 40(5):405–429, 2010.
8. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop (HCW'00)*, 2000.
9. H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS parameter sweep template: User-level middleware for the Grid. In *Supercomputing (SC'00)*, 2000.
10. M. Comuzzi, C. Kotsokalis, G. Spanoudkis, and R. Yahyapour. Establishing and monitoring SLAs in complex service based systems. In *IEEE International Conference on Web Services 2009*, 1009.
11. A. D'Ambrogio and P. Bocciarelli. A model-driven approach to describe and predict the performance of composite services. In *6th International Workshop on Software and Performance (WOSP'07)*, 2007.
12. G. Dobson and A. Sanchez-Macian. Towards unified QoS/SLA ontologies. In *IEEE Services Computing Workshops (SCW'06)*, 2006.
13. E. Elmroth and J. Tordsson. A grid resource broker supporting advance reservations and benchmark-based resource selection. In *Applied Parallel Computing*, 2006.
14. V. C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar. Low level metrics to high level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments. In *High Performance Computing and Simulation Conference (HPCS'10)*, 2010.
15. ESPER. Event stream processing. http://esper.codehaus.org/.
16. FoSII. Foundations of self-governing infrastructures. http://www.infosys.tuwien.ac.at/linksites/FOSII/index.html.
17. H. M. Frutos and I. Kotsiopoulos. BREIN: Business objective driven reliable and intelligent grids for real business. *International Journal of Interoperability in Business Information Systems*, 3(1):39–42, 2009.
18. W. Fu and Q. Huang. GridEye: A service-oriented grid monitoring system with improved forecasting algorithm. In *International Conference on Grid and Cooperative Computing Workshops*, 2006.

19. A. S. Glassner et al. *An introduction to ray tracing*. Academic Press London, 1989.
20. D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A toolkit for distributed system performance analysis. In *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'00)*, 2000.
21. JMS. Java messaging service. http://java.sun.com/products/jms/.
22. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
23. B. Koller and L. Schubert. Towards autonomous sla management using a proxy-like approach. *Multiagent Grid Systems*, 3(3):313–325, 2007.
24. K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software - Practice and Experience*, 32(2):135–164, 2002.
25. C. B. Lee and A. Snavely. On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions. *Int. J. High Perform. Comput. Appl.*, 20(4):495–506, 2006.
26. K. Lee, N. W. Paton, R. Sakellariou, and A. A. F. Alvaro. Utility driven adaptive worklow execution. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 220–227, Washington, DC, USA, 2009. IEEE Computer Society.
27. M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30(7):817–840, 2004.
28. M. Maurer, I. Brandic, V. C. Emeakaroha, and S. Dustdar. Towards knowledge management in self-adaptable clouds. In *4th International Workshop of Software Engineering for Adaptive Service-Oriented Systems (SEASS'10)*, 2010.
29. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *9th International Symposium on Cluster Computing and the Grid (CCGRID'09)*, 2009.
30. Oracle. Oracle virtualization. http://www.oracle.com/technologies/virtualization.
31. 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, and F. Galan. The RESERVOIR model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):Paper 4, 2009.
32. F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping performance and dependability attributes of web services. In *IEEE International Conference on Web Services (ICWS'06)*, 2006.
33. SAX. Simple API for XML. http://sax.sourceforge.net/.
34. B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.
35. S. Venugopal, J. Broberg, and R. Buyya. OpenPEX: An open provisioning and execution system for virtual machines. In *17th International Conference on Advanced Computing and Communications (ADCOM'09)*, 2009.
36. T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.
37. C. S. Yeo and R. Buyya. Pricing for utility-driven resource management and allocation in clusters. *Int. J. High Perform. Comput. Appl.*, 21(4):405–418, 2007.