CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **25**:1462–1481
Published online 10 October 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2944

SPECIAL ISSUE PAPER

Cloud resource provisioning and SLA enforcement via LoM2HiS framework

Vincent C. Emeakaroha*,†, Ivona Brandic, Michael Maurer and Schahram Dustdar

Information Systems Institute, Distributed System Group, Vienna University of Technology, Vienna, Austria

SUMMARY

Cloud computing represents a novel on-demand computing technology where resources are provisioned in compliance to a set of predefined non-functional properties specified and negotiated by means of service level agreements (SLAs). Currently, cloud providers strive to achieve efficient SLA enforcement strategies to avoid costly SLA violations during application provisioning and to timely react to failures and environmental changes. These strategies include advanced application deployment mechanisms and appropriate resource monitoring concepts. In terms of cloud resource monitoring, providers tend to adopt existing monitoring tools, such as those from grid environments. However, those tools are usually restricted to locality and homogeneity of monitored objects, are not scalable, and do not support mapping of low-level resource metrics (e.g., system uptime and downtime) to high-level application-specific SLA parameters (e.g., system availability). In this paper, we present a novel low-level metrics to high-level SLA (*LoM2HiS*) framework for managing the monitoring of low-level resource metrics and mapping them to high-level SLAs and an application deployment mechanism for scheduling and provisioning applications in clouds. The LoM2HiS framework provides the application deployment mechanism with monitored information and SLA violation prevention techniques, thereby ensuring the performance of the applications and thus increasing the revenue of the cloud provider by avoiding SLA violation penalty cost. This framework is the building block of the Foundations of Self-governing ICT Infrastructures project, which intends to facilitate autonomic SLA management and enforcement. Thus, the LoM2HiS framework detects future SLA violation threats and can notify the knowledge component to act so as to avert the threats. We discuss in detail the conceptual design of the LoM2HiS framework and the application deployment mechanism including their implementations. Finally, we present our evaluation results based on a use-case scenario demonstrating the usage of the LoM2HiS framework in a real cloud environment. Copyright © 2012 John Wiley & Sons, Ltd.

Received 5 January 2011; Revised 13 June 2012; Accepted 6 September 2012

KEY WORDS: cloud computing; resource monitoring; SLA mapping; SLA enforcement; SLA management; performance evaluation

1. INTRODUCTION

In recent years, cloud computing has become a key information technology megatrend that will take root, although it is in its infancy in terms of market adoption. Cloud computing is a promising technology that evolved out of several concepts such as virtualization, distributed application design, grid, and enterprise information technology management to enable a more flexible approach for deploying and scaling applications at low cost [1].

Service provisioning in clouds is based on service level agreements (SLAs), which are sets of non-functional properties specified and negotiated between the customer and the service provider.

*Correspondence to: Vincent C. Emeakaroha, Information Systems Institute, Distributed System Group, Vienna University of Technology, Vienna, Austria.

†E-mail: vincent@infosys.tuwien.ac.at

It states the terms of the service including the quality of service (QoS), obligations, service pricing, and penalties in case of agreement violations. Flexible and reliable management of SLAs is of paramount importance to both cloud providers and consumers. On the one hand, the prevention of SLA violations avoids penalties that are costly to providers, and on the other hand, on the basis of flexible and timely reactions to possible SLA violation threats, administrator interaction with the system can be minimized, enabling cloud computing to take root as a flexible and reliable form of on-demand computing.

To guarantee an agreed SLA in clouds, the cloud provider must be capable of monitoring its infrastructure (host) resource metrics to enforce the agreed service level objectives. However, most of the existing monitoring technologies rely on either grid [2, 3] or service-oriented infrastructures [4], which are not directly compatible to clouds because of the difference of resource usage model or because of heavily network-oriented monitoring infrastructures [5]. The differences between grids and cloud can be explained from different viewpoints such as resource scheduling and availabilities. For example, in grids [6], resources are mostly owned by different individuals/enterprises, and in some cases, desktop grids for instance, resources are only available for usage when the owners are not using them [7]. 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. Moreover, there is a gap between monitored metrics, which are usually low-level entities, and SLA agreements, which are high-level customer guarantee parameters.

In this paper, we present a novel framework for the monitoring and mapping of low-level resource metric to high-level SLA parameters named the LoM2HiS framework, which is also capable of evaluating application SLAs at run time to detect SLA violation situations to ensure the agreed applications' QoS level, and an application deployment mechanism for scheduling and provisioning applications in clouds. Furthermore, we discuss a knowledge management technique based on case-based reasoning (CBR) that is responsible for proposing reactive actions to prevent or correct the detected SLA violation situations. In preventing the SLA violations, we do not reserve resources for the applications, rather, we provision resources on demand.

The LoM2HiS framework is embedded into the Foundations of Self-governing ICT Infrastructures (FoSII) project, which aims at developing an infrastructure for autonomic SLA management and enforcement in cloud environments. Thus, LoM2HiS represents the first building block of the FoSII [8]. We present the conceptual design of this framework including the run-time monitor, host monitor, and the SLA mapping rules. We discuss a novel communication mechanism for the LoM2HiS framework, which supports its scalability. This mechanism is based on queuing networks. Moreover, we demonstrate sample mappings from the low-level resource metrics to the high-level SLA parameters.

The main contributions of this paper are as follows: (i) the design of the low-level resource monitoring and communication mechanisms; (ii) the definition of mapping rules using domain-specific languages; (iii) the mapping of the LoM2HiS objectives; (iv) the design of the application deployment mechanism; (v) the evaluation of the SLA objectives at run time to detect violation threats or real violation situations; (vi) the evaluation of the LoM2HiS framework in a real cloud testbed with a use-case scenario consisting of an image-rendering application such as Persistence of Vision Raytracer (POV-Ray) [9]; and (vii) the analysis of the achieved results to determine the optimal measurement intervals for efficient monitoring of different application types.

The rest of the paper is organized as follows. Section 2 presents related work. In Section 3, we present the background and motivation for this research work including some detail about the knowledge management component. The conceptual design and implementation issues of the LoM2HiS framework together with the application deployment mechanism are presented in Section 4. Section 5 deals with the framework evaluation based on a real cloud testbed using POV-Ray applications and the discussion of the achieved results. Section 6 presents the conclusion of the paper and our future research work.

DOI: 10.1002/cpe

2. RELATED WORK

We classify related work on SLA management and enforcement of cloud-based services into the following: (i) cloud resource monitoring [5, 10, 11]; (ii) SLA management including QoS management [12–17]; and (iii) mapping techniques of monitored metrics to SLA parameters and attributes [4, 18, 19].

The focus of the FoSII project and the LoM2HiS framework is on cloud environments. However, because there is very little work on monitoring, SLA management, and metrics mapping in cloud environments, we look particularly into related areas such as grid and service-oriented-architecture-based systems for available strategies.

Fu *et al.* [10] proposed 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 discussed how resources are monitored with their approach in the grid environment, but they considered neither SLA management nor low-level metric mapping. Gunter *et al.* [5] presented NetLogger, a distributed monitoring system that monitors and collects information from networks. Applications can invoke NetLoggers API to survey the overload before and after some request or operation. However, it monitors only network resources. Wood *et al.* [11] developed a system called Sandpiper, which automates the process of monitoring and detecting hotspots and remapping/reconfiguring virtual machines (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.* [17] discussed dynamic service provisioning using GRIA SLAs. The authors described provisioning of services according to agreed SLAs and the management of the SLAs to avoid violations. Their approach is limited to grid environments. Moreover, they did not detail how the low-level metrics are monitored and mapped to high-level SLAs. Theilman *et al.* [14] discussed an approach for multi-level SLA management, where SLAs are consistently specified and managed within a service-oriented infrastructure. They presented the run-time functional view of the conceptual architecture and discussed different case studies including enterprise resource planning and financial services. But they did not address low-level resource monitoring and SLA mappings. Koller *et al.* [12] discussed autonomous QoS management using a proxy-like approach. The 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 requirements of cloud computing infrastructures such as scalability. Frutos *et al.* [13] discussed the main approach of the EU project BREIN [20] 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. However, BREIN applies SLA management to grids, whereas we target SLA management in clouds. Dobson *et al.* [15] presented a unified QoS ontology applicable to the main scenarios identified such as QoS-based Web services selection, QoS monitoring, and QoS adaptation. However, they did not consider low-level resource monitoring in their approach to ensure QoS. Comuzzi *et al.* [16] defined the process for SLA establishment adopted within the EU project SLA@SOI framework. The authors proposed an architecture for the monitoring of 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. Moreover, they did not detail their process of evaluating SLA objectives to detect violation situations at run time.

Brandic *et al.* [19] presented an approach for adaptive generation of SLA templates. Thereby, cloud customers can define mappings from their local SLA templates to the remote templates to facilitate communication with numerous cloud service providers. However, they did not investigate mapping of monitored metrics to agreed SLAs. Rosenberg *et al.* [18] dealt with QoS attributes for Web services. They identified important QoS attributes and their composition from resource metrics. They presented some mapping techniques for composing QoS attributes from resource metrics to form SLA parameters for a specific domain. However, they did not deal with monitoring of

DOI: 10.1002/cpe

resource metrics. Bocciarelli *et al.* [4] introduced a model-driven approach for integrating performance prediction into service composition processes carried out using Business Process Execution Language (BPEL). In their approach, they compose SLA parameters from resource metrics using some mapping techniques. But they considered neither resource metrics nor SLA monitoring.

To the best of our knowledge, none of the discussed approaches deal with mappings of low-level monitored metrics to high-level SLA guarantees, such as those necessary in cloud environments.

3. BACKGROUND AND MOTIVATIONS

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 [1, 21]. We are currently developing an infrastructure called FoSII [8], which proposes models and concepts for autonomic SLA management and enforcement in clouds. The FoSII are capable of managing the whole life cycle of self-adaptable cloud services [22].

The essence of using SLA in the 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. To save cloud providers from paying costly penalties and to enhance their revenue, we devised the LoM2HiS framework [23], which is a core component of the FoSII for monitoring cloud resources, mapping the low-level resource metrics to high-level SLA parameter objectives and detecting SLA violations as well as future SLA violation threats so as to react before actual SLA violations occur. The *LoM2HiS* framework represents a major step toward achieving the goals of the FoSII project.

3.1. *Foundations of Self-governing ICT Infrastructures overview*

Figure 1 presents an overview of the FoSII. 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 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 [22].

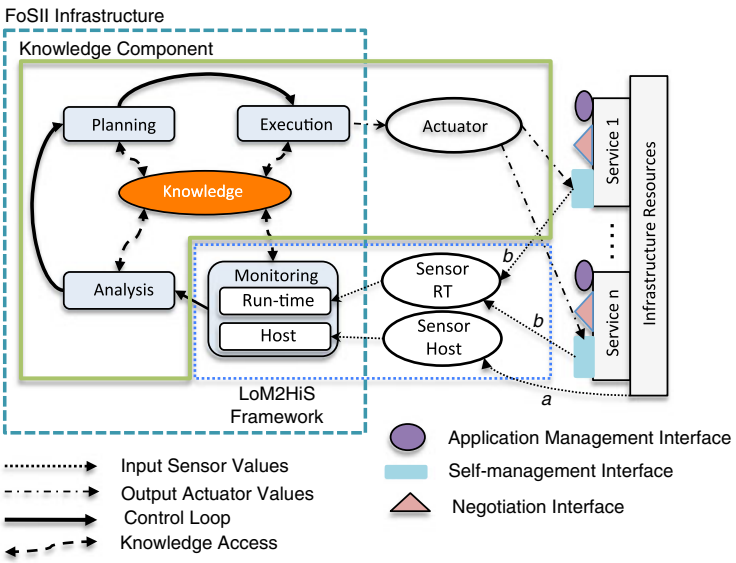


Figure 1. Foundations of Self-governing ICT Infrastructures overview. LoM2HiS, low-level metrics to high-level service level agreement.

DOI: 10.1002/cpe

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) from resource usage experiences and predefined threat thresholds.

Logically, FoSII consist of multiple components working together to achieve a common goal. In this paper, we give some details of the knowledge management component and focus on the LoM2HiS framework because they are responsible for system monitoring, detecting SLA violations, and proposing reactive actions for possible prevention or correction of violation situations.

3.2. Knowledge management component

As shown in Figure 1, we rely on a knowledge management technique to realize autonomic behaviors in FoSII. We are currently investigating different techniques including the rule-based approach and CBR for this purpose [24, 25]. In this section, we give some details about our research work on the CBR technique.

The CBR was first built on top of FreeCBR [26] but is now a completely independent Java framework, taking into account, however, basic ideas of the FreeCBR. It can be defined as the process of solving problems from past experiences [27]. In more detail, it tries to solve a *case*, which is 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.

Figure 2 presents an overview of the CBR process. The ideas of using CBR in SLA management are to have rules stored in a database that engage the CBR system once a threshold value has been reached for a specific SLA parameter. The notification information is fed into the CBR system as new cases by the monitoring component. Then, the CBR, prepared with some initial meaningful cases stored in DB2 (Figure 2), chooses the set of cases that is most similar to the new case on the basis of various factors as described in [24]. From these cases, we select the one with the highest utility measured previously and trigger its corresponding action as the proposed action to solve the new case. Finally, we measure in a later time interval the result of this action in comparison with the initial case and store it with its calculated utilities as a new case in the CBR. Doing this, we can constantly learn new cases and evaluate the usefulness of our triggered actions.

In general, a typical CBR cycle consists of the following phases, assuming that a new case was just received:

1. Retrieve the most similar case or cases to the new one.
2. Reuse the information and knowledge in the similar case(s) to solve the problem.
3. Revise the proposed solution.
4. Retain the parts of this experience likely to be useful for future problem solving. (Store the new case and corresponding solution into the knowledge database.)

Furthermore, we present here a practical demonstration of a CBR system, considering the SLA objectives depicted in Table I and as shown in Figure 3. A complete case consists of the following: (i) the ID of the application being provisioned (line 2, Figure 3); (ii) the initial case measured by the monitoring component and mapped to the SLAs including the application SLA parameter values and global cloud information such as the number of running VMs (lines 4–9); (iii) the executed

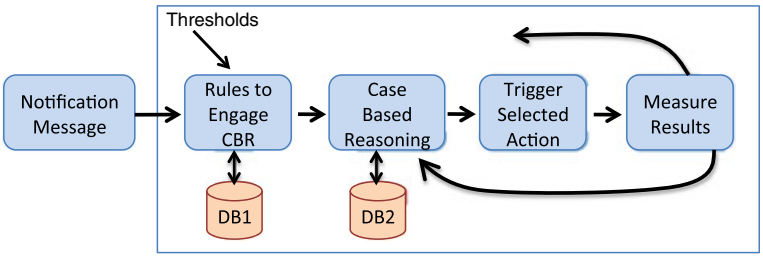


Figure 2. Case-based reasoning (CBR) process overview.

DOI: 10.1002/cpe

Table I. Sample service level agreement parameter objectives.

SLA parameter	Value
Incoming bandwidth	>10 Mbit/s
Outgoing bandwidth	>12 Mbit/s
Storage	> 1024 GB
Availability	⩾ 99%

```
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 3. Case-based reasoning example.

action (line 11); (iv) the resulting case measured some time interval later (lines 13–18) as in (ii); and (v) the resulting utility (line 20). In applying CBR, we assume infinite resource availability. Therefore, the allocation of more resources to one customer does not affect the provisioning of the other customer applications.

We distinguish between two working modes of the knowledge management: *active* and *passive* [28]. In the active mode, system states and SLA values are periodically stored into the database. Thus, on the basis of the observed violations and correlated system states, cases are obtained as input for the knowledge database. Furthermore, on the basis of the utility functions, the quality of the reactive actions are evaluated, and threat thresholds (explained in Section 4) are generated.

However, the definition of measurement intervals in the active mode is far from trivial. An important parameter to be considered is the period at which resource metrics and SLA parameters are evaluated (e.g., every 2 s or every 2 min). Too frequent measurement intervals may negatively affect the overall system performance, whereas too infrequent measurement intervals may cause heavy SLA violations.

Having discussed the overviews of FoSII and knowledge management, we now present in the next section the LoM2HiS framework, which is the building block of the FoSII.

4. LOW-LEVEL METRICS TO HIGH-LEVEL SERVICE LEVEL AGREEMENT FRAMEWORK: DESIGN AND IMPLEMENTATION

The LoM2HiS framework is the first step toward achieving the goals of the FoSII. In this section, we give the details of the LoM2HiS framework and describe its components and their implementations.

DOI: 10.1002/cpe

We present the design descriptions and the implementation details of this framework together in this section to give the reader a compact and comprehensive overview.

In this framework, we assumed that the SLA negotiation processes are completed and that the agreed SLAs are stored in the database for service provisioning. Besides the SLAs, the predefined threat thresholds for guiding the SLA objectives are also stored in this database. The concept of detecting future SLA violation threats is designed by defining more restrictive thresholds that are known as threat thresholds, which are stricter than the normal SLA objective violation thresholds. In this paper, we assume predefined threat thresholds because the autonomic generation of threat thresholds is far from trivial and is part of our ongoing research work.

Figure 4 presents the architecture of our LoM2HiS framework. The application deployment component including the run-time monitor represents the application layer where services are scheduled for deployment on the cloud resources. The run-time monitor is designed to monitor the services according to the negotiated and agreed SLAs. After agreeing on SLA terms, the service provider creates mapping rules for the low-level to high-level SLA mappings (step 1 in Figure 4) using domain-specific languages. Domain-specific languages are small languages that can be tailored to a specific problem domain. Once the customer requests the provisioning of an agreed service (step 2), the run-time monitor loads the SLA from the agreed SLA database (step 3). The provisioning of service applications (step 4) is based on the infrastructure resources, which represent the physical machines and VMs and the 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. The host monitor extracts the metric value pairs from the raw metrics and transmits them periodically to both the run-time monitor (step 5) and the knowledge component (step 6) using our designed communication mechanism.

Upon receiving the measured metrics, the run-time monitor maps the low-level metrics from predefined mapping rules to form an equivalent of the agreed SLA objectives. The mapping results

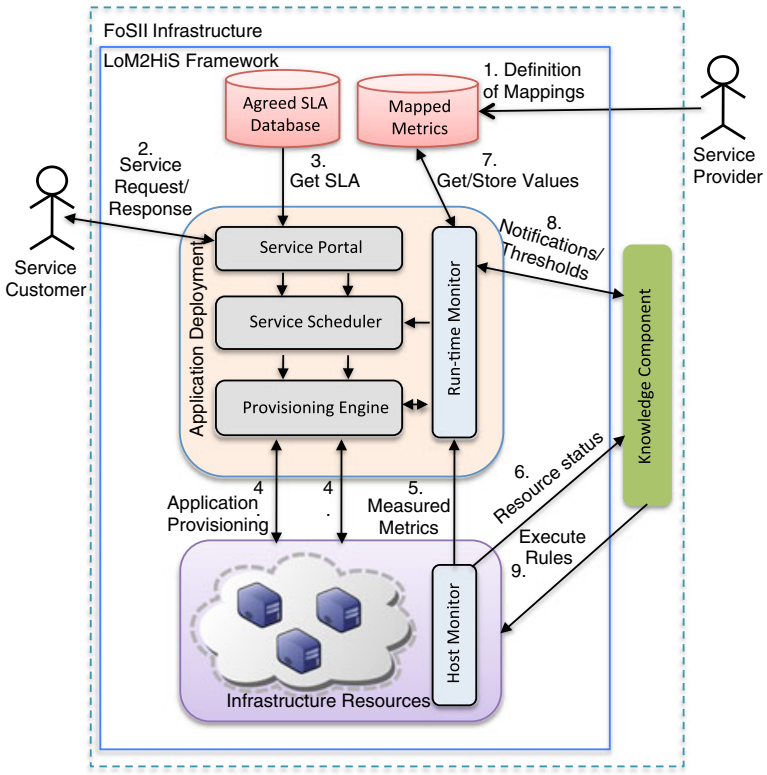


Figure 4. Low-level metrics to high-level service level agreement (LoM2HiS) framework architecture. FoSII, Foundations of Self-governing ICT Infrastructures; SLA, service level agreement.

DOI: 10.1002/cpe

are stored in the mapped metric database (step 7), which also contains the predefined mapping rules. The run-time monitor uses the mapped values and the predefined thresholds 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 on the basis of the resource status (step 6) and defined rules [28]. The knowledge component decisions (e.g., assign more CPU to a virtual host) are executed on the infrastructure resources (step 9).

The LoM2HiS framework is designed to be scalable. In its design, the separation of the host monitor and the run-time monitor makes it possible to deploy these two components on different hosts. This decision is focused toward increasing the scalability of the framework and facilitating its usage in distributed and parallel environments. We evaluate the usage of the LoM2HiS framework in this paper for a single cloud environment. However, we are already making efforts to extend its usage in federated clouds.

4.1. Application deployment mechanism

The design and implementation details of the application deployment mechanism are presented in this section. We first describe the scheduling design concepts and later discuss its implementation. The run-time monitor described in Section 4.4 is an integral part of this mechanism.

4.1.1. Design issues. The application deployment mechanism is responsible for managing the execution of customer service applications. Its functions are similar to those of a broker in grid technology [29–31]. But compared with a broker, our deployment mechanism has more knowledge and considers SLA objectives in scheduling the customer application. Figure 5 presents a detailed overview of the application deployment mechanism.

The core part of the application deployment is the scheduler, which is responsible for accepting the customer requests through the service portal and scheduling them on the basis of the previously agreed SLA objectives stored in the provider database. The scheduler uses the SLA objectives to determine on which host in the cloud environment to map the execution of the customer application. Once the scheduler decides on a scheduling plan, it passes it to the provisioning engine for execution. The provision engine manages the provisioning of the customer applications on the cloud environment resources. It copies the application data to the VMs, triggers their execution there, and collects the outputs of the executions. The run-time monitor interacts with this module to monitor the application SLA objectives during their executions to detect SLA violation threats or real SLA violation situations.

4.1.2. Implementation decisions. The application deployment mechanism is written in Java. It takes customer service request as input, which contains the details of the application data wrapped as a service. This request information is passed to the scheduler for analyzing and processing. The scheduler extracts the SLA objectives from the database using a Java routine that determines the appropriate SLA agreement using their IDs.

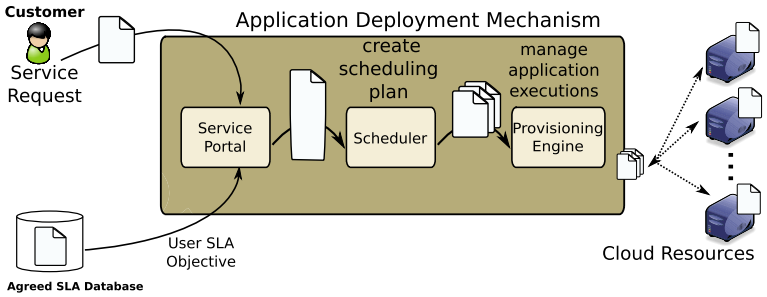


Figure 5. Application deployment mechanism. SLA, service level agreement.

DOI: 10.1002/cpe

The scheduler communicates with the run-time monitor to receive the available resource status in the cloud environment, which it then uses to create the scheduling plan. The provisioning engine is implemented as a Java class; it receives the scheduling plan as input and organizes for the execution of the customer service. It uses the *scp* command (a standard Linux tool for copying files among multiple machines) to copy the application input files to the appropriate VMs for execution. The actual execution of the applications on the VMs is triggered by the provisioning engine through an *ssh* connection. When the application provisioning is completed, the results are copied back to the provisioning engine using the *scp* command, and it delivers them to the customers.

4.2. Host monitor

This section describes the host monitor component, which is located at the cloud infrastructure resource level. We first explain its design strategy and later present the implementation details.

4.2.1. Host monitor design. The host monitor is responsible for processing monitored values delivered by the monitoring agents embedded in the infrastructure resources. The monitoring agents are capable of measuring both hardware and network resources. Figure 6 presents the host monitoring system.

As shown in Figure 6, the monitoring agent embedded in device 1 (D1) measures its resource metrics and broadcasts them to D2 and D3. Equally, the monitoring agent in D2 measures and broadcasts its measured metrics to D1 and D3. Thus, we achieve a replica management system in the sense that each device has a complete result of the monitored infrastructure. The host monitor can access these results from any device. It can be configured to access different devices at the same time for monitored values. In case one fails, the result will be accessed from the others. This eradicates the problem of a bottleneck system and offers fault-tolerant capabilities. Note that a device can be a physical machine, a VM, a storage device, or a network device. Furthermore, it should be noted that the broadcasting mechanism described earlier is configurable and can be deactivated in a cloud environment where there are many devices within resource pools to avoid communication overheads, which may consequently lead to degraded overall system performance.

The decision to separate the monitoring agents from the data processing units aims to make this framework scalable and less intrusive. Processing the monitored data in a separate node frees the resources on the computing nodes for computational tasks, thereby reducing the monitoring effects on these nodes to a minimum and allowing the computational tasks to achieve high performance.

4.2.2. Host monitor implementation. The host monitor implementation uses the GMOND module from the GANGLIA open-source project [32] as the monitoring agent. The GMOND module is a stand-alone component of the GANGLIA project. The monitoring agents are embedded in each of the computing nodes, and we use them to monitor the resource metrics of these nodes. The monitored results are presented in an XML file and written to a predefined network socket. We

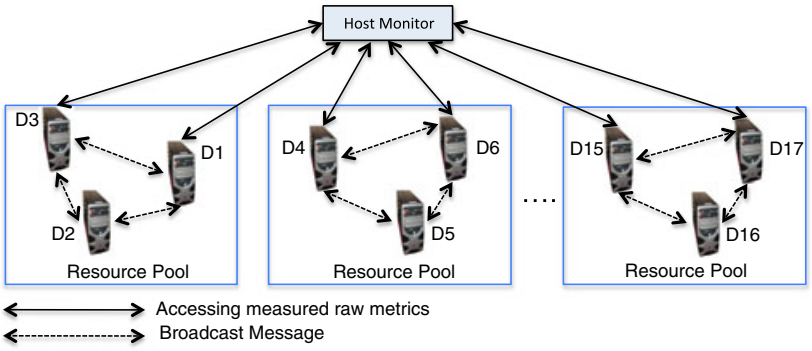


Figure 6. Host monitoring system.

DOI: 10.1002/cpe

implemented a Java routine to listen to this network socket where the GMOND writes the XML file containing the monitored metrics to access the file for processing.

The processing of the monitored data is carried out in a separate host. For this purpose, we implemented an XML parser using the well-known open-source SAX API [33] to parse the XML file to extract the metric value pairs. The measured metric value pairs are sent to the run-time monitor using our implemented communication mechanism. These processes can be performed once or repeated periodically depending on the monitoring strategy being used.

4.3. Communication mechanism

The components of the FoSII exchange large number of messages with each other and within the components. Thus, there is a need for a reliable and scalable means of communication.

To satisfy this need of communication means, we designed and implemented a communication mechanism based on the Java Messaging Service (JMS) API, which is a Java message-oriented middleware API for sending messages between two or more clients [34]. So that a JMS can be used, there is a need for a JMS provider that is capable of managing the sessions and the queues. In this case, we use the well-established open-source Apache ActiveMQ [35] for this purpose.

The implemented communication model is a sort of queuing network. It realizes an inter-process communication for passing messages within the FoSII and between the components of the LoM2HiS framework because of the fact that the components can run on different machines at different locations. This queue makes the communication mechanism highly efficient and scalable.

Figure 7 presents an example scenario expressing the usage of the communication mechanism in the LoM2HiS framework. The scenario of Figure 7 depicts the processes of extracting the low-level metrics from the monitoring agents embedded in the computing nodes, processing the gathered information in the host monitor, and passing the derived metric value pairs to the run-time monitor for mapping and SLA objective monitoring. The communication mechanism is very essential for communicating among the components of the LoM2HiS framework. However, it is not a core component. Thus, we did not structure its descriptions into the design and implementation sections.

4.4. Run-time monitor

The run-time monitor component is an integral part of the application deployment mechanism. In this section, we present the detailed description of this component. We first describe the design of the component and later explain its implementation detail.

4.4.1. Run-time monitor design. The run-time monitor carries out the LoM2HiS mappings. Thus, on the basis of the mapped values, the SLA objectives, and the predefined thresholds, it continuously monitors the customer application status and performance. Its operations are based on three information sources: (i) the resource metric value pairs received from the host monitor; (ii) the SLA parameter objective values stored in the agreed SLA database; and (iii) the predefined threat threshold values. The metric value pairs are low-level entities, and the SLA objective values are high-level entities, so for the run-time monitor to work with these two values, they must be mapped into common values.

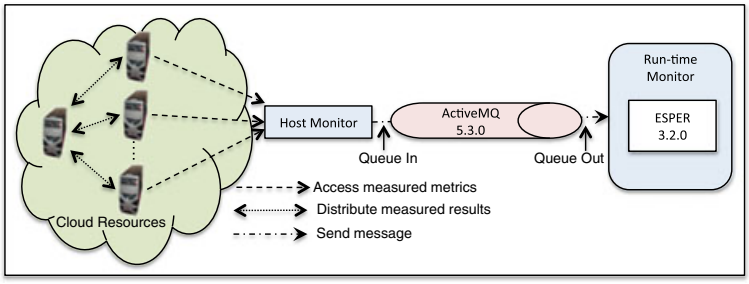


Figure 7. Communication mechanism scenario.

DOI: 10.1002/cpe

Table II. Complex mapping rules.

Resource metrics	SLA parameter	Mapping rule
downtime, *uptime*	Availability (A)	$A = (1 - (downtime/uptime + downtime)) * 100$
inbyte, *outbytes*, *packetsize*, *bandwidthin*, *bandwidthout*	Response time (R_{total})	$R_{total} = R_{in}\ (s) + R_{out}\ (s)$

4.4.2. Mapping of low-level metric to high-level service level agreements. As already discussed in Section 4, the run-time monitor chooses the mapping rules to apply on the basis of the application being provisioned. That is, for each application type, there is a set of defined rules for performing their SLA parameter mappings. These rules are used to compose, aggregate, or convert the low-level metrics to form the high-level SLA parameter. We distinguish between simple and complex mapping rules. A simple mapping rule maps one to one from a low level to a high level, for example mapping a low-level metric *disk space* to a high-level SLA parameter *storage*. In this case, only the units of the quantities are considered in the mapping rule. Complex mapping rules consist of predefined formulae for the calculation of specific SLA parameters using the resource metrics. For our evaluations later in this paper, we execute simple mapping rules. Nevertheless, Table II presents some complex mapping rules.

In the mapping rules presented in Table II, the downtime variable represents the *mean time to repair*, which denotes the time it takes to bring a system back online after a failure situation, and the uptime represents the *mean time between failure*, which denotes the time the system was operational between the last system failure and the next. The unit of the availability equation is in percentage. In this equation, we assume that the *uptime* and *downtime* variables are greater than or equal to 0. R_{in} is the response time for sending a request in the inwards communication direction and is calculated as $packetsize/(bandwidthin - inbytes)$ in seconds. The *packetsize* is the size of the query sent in, and its unit is in megabit. The *bandwidthin* is the total bandwidth for communicating in the inwards direction, and its unit is in megabit per second. The *inbytes* is the amount of bandwidth already in use on this channel, and its unit is in megabit per second. R_{out} is the response time for receiving an answer in the outwards communication directions and is calculated as $packetsize/(bandwidthout - outbytes)$ in seconds. The meaning of the variables in this equation is similar to those in the R_{in} equation, and their units are the same. The response time equation shown in Table II considers only the data transfer time values. It does not include computational time values because in our experiments with Web applications, these values are small and negligible. Therefore, this response time equation is customized to Web application and does not represent a generalized formula for other application types. For example, in the case of data-intensive applications, the computational time cannot be neglected. The mapped SLAs are stored in the mapped metric database for usage during the monitoring phase.

4.4.3. Monitoring service level agreement objectives and notifying the knowledge component. In this phase, the run-time monitor accesses the mapped metrics' database to obtain the mapped SLA parameter values that are equivalent to the agreed SLA objectives, which it uses together with the predefined thresholds in the monitoring process to detect future SLA violation threats or real SLA violation situation. This is achieved by comparing the mapped SLA values against the threat thresholds to detect future violation threats and against SLA objective thresholds to detect real violation situations. In case of detection, it dispatches notification messages to the knowledge component to avert the threats or correct the violation situation. An example of an SLA violation threat is something like an indication that the system is running out of storage. In such a case, the knowledge component acts to increase the system storage. Real violations occur only if the system is unable to resolve the cause of a violation threat notification.

4.4.4. Run-time monitor implementations. The run-time monitor receives the measured metric value pairs and passes them to the Esper engine [36] for further processing. Esper is a component of complex event processing and event stream processing applications, available for Java as Esper and for .NET as NEsper. Complex event processing is a technology to process events and discover

DOI: 10.1002/cpe

complex patterns among multiple streams of event data, whereas event stream processing deals with the task of processing multiple streams of event data with the goal of identifying the meaningful events within those streams and deriving meaningful information from them.

We use this technology because the JMS system used in our communication model is stateless and as such makes it hard to deal with temporal data and real-time queries. From the Esper engine, the metric value pairs are delivered as events each time their values change between measurements. This strategy reduces drastically the number of events/messages processed in the run-time monitor. We use an XML parser to extract the SLA parameters and their corresponding objective values from the SLA document and store them in a database. The LoM2HiS mappings are realized in Java methods, and the returned mapped SLA objectives are stored in the mapped metrics database.

Unlike in the former version of the LoM2HiS framework [23] where repositories are used to store information, we introduce a MySQL database in this current version for storing and querying information. We use Hibernate to realize an interface between the run-time monitor classes and the MySQL database. Hibernate is a high-performance object/relational persistence and query service. Hibernate, the most flexible and powerful object/relational solution on the market, takes care of mapping from Java classes to database tables and from Java data types to SQL data types. It provides data query and retrieval facilities that significantly reduce development time [37]. With this database, the cloud provider now has the capability to create a graphical interface for displaying the monitored metric values and the SLA objective status including the reported violation threats and real violation situations.

5. LOW-LEVEL METRICS TO HIGH-LEVEL SERVICE LEVEL AGREEMENT FRAMEWORK EVALUATIONS

In this section, we use an image-rendering use-case scenario to evaluate our proposed novel framework. The goals of our evaluations are to provide a proof of concept and to determine the optimal measurement interval for monitoring agreed SLA objectives for applications at run time. We first present the details of the image-rendering application use-case scenario after which we discuss in detail our real cloud experimental environment, and lastly, we present the evaluation results.

5.1. Image-rendering application use-case scenario

For the evaluation of our approach, we used an image-rendering application based on the POV-Ray,[‡] which is a ray tracing program available for several computing platforms [9]. 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 later and illustrated in Figures 8 and 9:

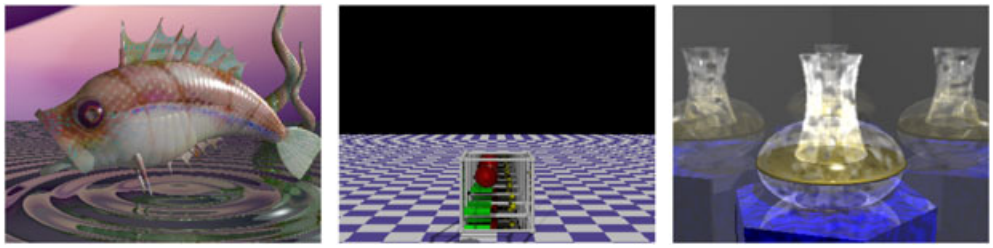
- **Fish:** rotation of a fish on water. Time for rendering frames is variable.
- **Box:** approximation of a camera to an open box with objects inside. Time for rendering frames increases during execution.
- **Vase:** rotation of a vase with mirrors around. Time for processing different frames is constant.

Three SLA documents are negotiated for the three POV-Ray applications. The SLA documents specify the level of QoS that should be guaranteed for each application during its execution. Table III presents the SLA objectives for each of the applications. These SLA objective thresholds are defined from test runs and experiences with these applications in terms of resource consumption. With the test runs, the cloud provider can determine the amount and type of resources the applications require. Thus, the provider can make a better resource provisioning plan for the applications. On the basis of these SLA objectives, the applications are monitored to detect SLA violations.

Each of the POV-Ray application is made of frames, which are being processed and executed at run time. The POV-Ray application frames can be executed in a distributed and parallel manner using prespecified computing nodes in a cloud environment. It is the processing of the frames that

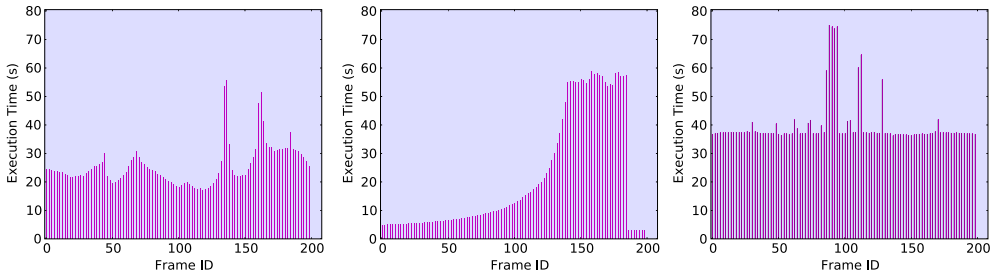
[‡]www.povray.org

DOI: 10.1002/cpe



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

Figure 8. Example of images for each of the three animations.



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

Figure 9. Behavior of execution time for each Persistence of Vision Raytracer application.

Table III. Persistence of Vision Raytracer applications' SLA objective thresholds.

SLA parameter	Fish	Box	Vase
CPU (%)	20	15	10
Memory (MB)	297	297	297
Storage (GB)	2.7	2.6	2.5

SLA, service level agreement.

Table IV. Cloud environment resource setup composed of 10 virtual machines.

Machine type	OS	CPU	Cores	Memory (GB)	Storage (GB)
Physical	OVM server	Pentium 4 2.8 GHz	2	2.5	100
Virtual	Linux/Ubuntu	Pentium 4 2.8 GHz	1	1	5

OS, operating system; OVM, Oracle virtual machine.

is computationally intensive and causes the load on the system at run time. In the next section, we describe our experimental cloud testbed used for the execution of this use-case scenario.

5.2. Experimental environment

The resource capacities of our cloud experimental testbed are shown in Table IV. The table shows the resource compositions of the physical machines and VMs being used in our experimental testbed. We use the Xen virtualization technology in our testbed; precisely, we run Xen 3.4.0 on top of the Oracle VM server.

We have in total five physical machines, and on the basis of their resource capacities as presented in Table IV, we host two VMs on each physical machine. We use an automated emulation framework [38] to deploy the VMs onto the physical hosts, thus creating a virtualized cloud environment

DOI: 10.1002/cpe

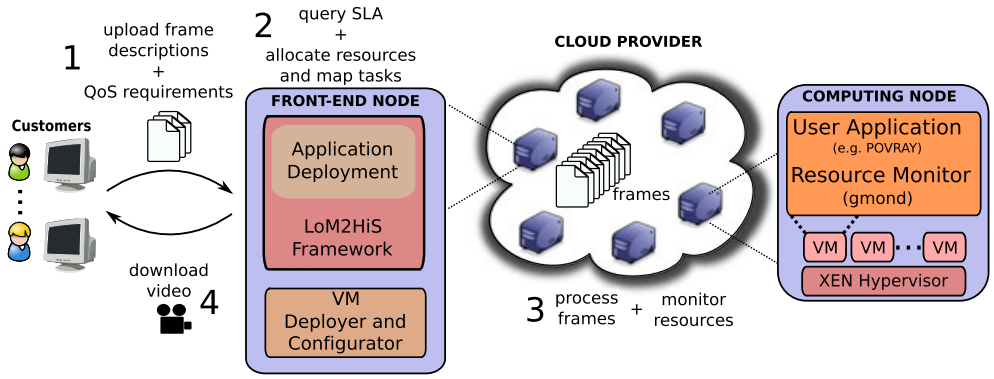


Figure 10. Evaluation testbed. LoM2HiS, low-level metrics to high-level service level agreement; QoS, quality of service; SLA, service level agreement; VM, virtual machine; POV-Ray, Persistence of Vision Raytracer.

with up to 10 computing nodes capable of provisioning resources to applications. One computing node, labeled front-end node, is responsible for the management activities.

Figure 10 presents the evaluation testbed. The computing node represents the VMs that execute POV-Ray frames submitted via application deployment. The VMs 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 VM 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 depicted on Figure 10, customers supply the QoS requirements in terms of SLOs (step 1 in Figure 10). At the same time, the images with the POV-Ray applications and input data (frames) can be uploaded to the front-end node. From the current system status, the SLA negotiator establishes an SLA with the customer. A description of the negotiation process and components is out of scope of this paper and is discussed by Brandic *et al.* [22]. Thereafter, the VM deployer starts configuration and allocation of the required VMs, whereas the application deployment maps the tasks to the appropriate VMs (step 3). In step 4, the application execution is triggered. The POV-Ray applications are executed in a distributed manner using all the computing node in our testbed in parallel.

As shown in Figure 10, the front-end node serves as the control entity. It runs the automated emulation framework, which configures VMs; the application deployment, which is responsible for managing the application execution locally on the VMs; and the LoM2HiS framework, which monitors and detects SLA violation situations. We use this cloud testbed to evaluate the use-case scenario presented in the previous section. We would like to mention that although the number of nodes in our testbed is small compared with that in public clouds such as Amazon EC2, the testbed is sufficient for supporting the execution of the POV-Ray applications and to test our proposed framework.

5.3. Achieved results and analysis

We defined and used six measurement intervals to monitor the POV-Ray applications during their executions. Table V shows the measurement intervals and the number of measurements made in each interval. The applications run for about 20 min for each measurement interval.

Table V. Measurement intervals.

	Intervals (s)					
	5	10	20	30	60	120
No. of measurements	240	120	60	40	20	10

DOI: 10.1002/cpe

The 5-s measurement interval is a reference interval, meaning the current interval used by the provider to monitor application executions on the cloud resources. Its results are assumed to capture all the SLA violation situations in the cloud environment.

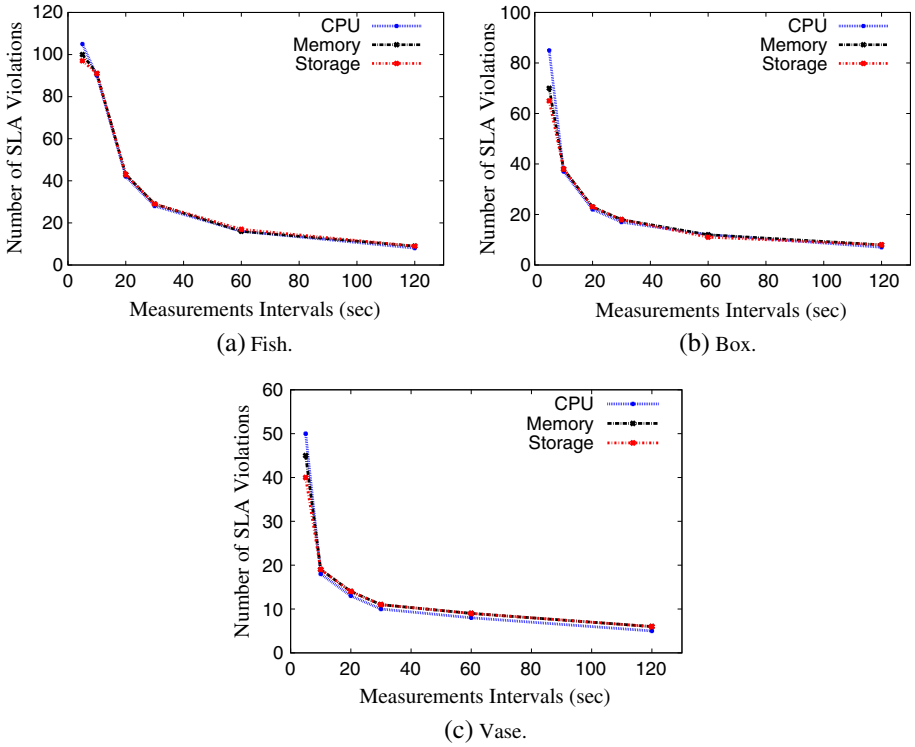
Figure 11 presents the achieved results of the three POV-Ray applications with varying characteristics in terms of frame rendering as explained in Section 5.1. We use the 10 VMs in our testbed to simultaneously execute the POV-Ray frames. There is a load balancer integrated in the application scheduling, which ensures that the frame executions are balanced among the VMs.

The LoM2HiS framework monitors the resource usage of each VM to determine if the SLA objectives are ensured and reports violations otherwise. Because the load balancer balances the execution of frames among the VMs, we plot in Figure 11 the average number of violations encountered in the testbed for each application with each measurement interval.

Figure 11 shows the number of SLA violations encountered with each measurement interval. The diagram indicates that as the measurement interval increases, many SLA violation detections are being missed in between the measurement frequency, which results to a low number of detected SLA violations in larger intervals. For example, in Figure 11(b), the 5-s measurement interval reports about 85 SLA violations for CPU resource in executing the box POV-Ray application, whereas the 2-min measurement interval reports about 10 for the same resource. This example clearly shows that many SLA violations went undetected in between the measurement with the 2-min measurement interval. Note that Figure 11 does not mean that the CPU, memory, and storage resources are violated at the same time but rather that the curves are close to each other to indicate similarities in resource consumption behaviors.

Therefore, to find the optimal measurement interval for detecting applications' SLA objective violations at run time, we discuss the following two determining factors: (i) the 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 or frequency.

Using these two factors and other parameters, we define a cost function (C) based on which we derive the optimal measurement interval (frequency). The ideas of defining this cost



(a) Fish. (b) Box.

(c) Vase.

Figure 11. Measurement intervals with number of service level agreement (SLA) violations.

DOI: 10.1002/cpe

function are derived from utility functions discussed by Lee *et al.* [39]. Equation (1) presents the cost function.

$$C = \mu * C_{\mathrm{m}} + \sum_{\psi \epsilon \{\mathrm{cpu,\ memory,\ storage}\}} \alpha(\psi) * C_{\mathrm{v}}, \tag{1}$$

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 from the results of the 5-s reference measurement interval, which is assumed to be an interval capturing all the possible application SLA objective violations. This cost function now forms the basis for analyzing the achieved results presented in Figure 11.

The cost of making measurement in our testbed is defined by considering the intrusiveness of the measurements on the overall performance of the system. On the basis of our testbed architecture and the intrusiveness test performed, we observed that measurements have minimal effects on the computing nodes. This is because measurements and their processing take place in the front-end node whereas the applications are provisioned using the computing nodes. The monitoring agents running on the computing nodes have minimal impact on resource consumption. This means a low cost of making measurements in the cloud environment.

The cost of missing an SLA violation detection is an economic factor, which depends on the SLA penalty cost agreed for the specific application and the effects of the violations on the cloud provider, for example in terms of reputation or trust issues.

Applying the cost function on the achieved results of Figure 11, with a measurement cost of \$0.5 and an aggregated missing violation cost of \$1.5, we achieve the monitoring costs presented in Table VI. These cost values are example values for our experimental setup. It neither represents nor suggests any standard values. The approach used here is derived from the cost function approaches presented in the literature [40, 41].

The costs presented in Table VI represent the total cost of measurements for each interval and the cost of missing to detect an SLA violation situation. The cost is calculated for each cloud resource. The different amounts of cost calculated for each resource type reflect the number of violations

Table VI. Monitoring cost.

		Intervals				
SLA parameter	Reference	10 s	20 s	30 s	1 min	2 min
		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

POV-Ray, Persistence of Vision Raytracer; SLA, service level agreement.

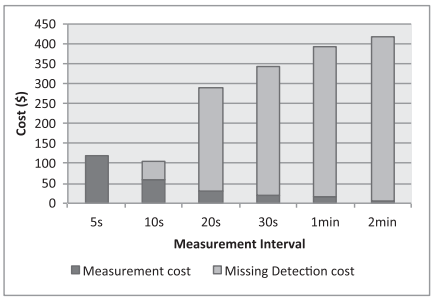
DOI: 10.1002/cpe

detected or missed for this specific resource type. This shows that the number and occurrence of SLA violations for the three resource types are not the same as could be misinterpreted from the curves in Figure 11.

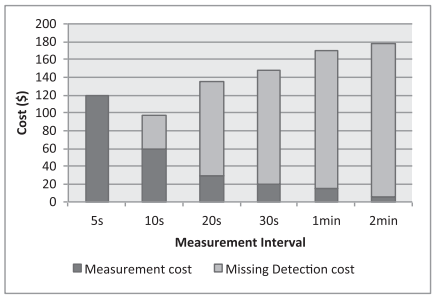
The reference measurement captures all SLA violations for each application; thus, it only incurs measurement cost. With a closer look at Table VI, it is clear that the cost values of the shorter measurement interval are closer to the reference measurement than those of the longer measurement interval. This is attributed to our novel architecture design, which separates management activities from computing activities in the cloud testbed.

The relations between the measurement cost and the cost of missing SLA violations for each interval are graphically depicted in Figure 12 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 is the measurement cost because of less frequency of measurements, and in terms of cost for missing SLA violations, the longer the measurement interval, the higher is the cost for missing SLA violations. This is due to the increasing number of missed SLA violation detection in between the longer measurement intervals. Thus, the sum of these two costs gives the cost of monitoring the application execution to guarantee the agreed SLA objectives.

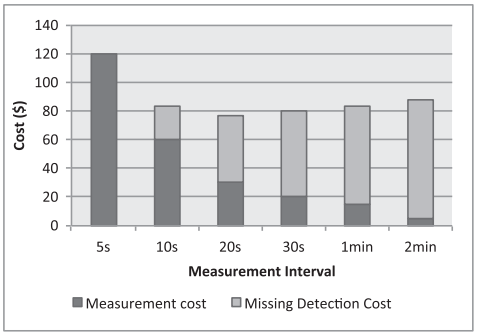
Considering the cost of monitoring the fish POV-Ray application in Table VI and Figure 12(a), it can be seen that the reference measurement is not the cheapest although it does not incur any cost of missing an SLA violation detection. In this case, the 10-s interval is the cheapest and in our opinion the most suited measurement interval for this application. In the case of the box POV-Ray application, the total cost of monitoring, as shown in Table VI and depicted graphically in Figure 12(b), indicates that the lowest cost is incurred with the 10-s measurement interval. Thus, we conclude that this interval is best suited for this application type. Also, from Table VI and Figure 12(c), it is clear that the reference measurement interval by far does not represent the optimal measurement interval for the vase POV-Ray application. From the application behavior, longer measurement intervals are better fitted than shorter ones. Therefore, in this case, the 20-s measurement interval is best suited for the considered scenario.



(a) Fish.



(b) Box.



(c) Vase.

Figure 12. Measurement cost relations.

DOI: 10.1002/cpe

From our experiments, it is observed that the optimal best-suited measurement interval depends on the application type. Therefore, there is no particular appropriate measurement interval for all application types. 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 customers and providers.

It should be noted that our evaluations in this paper do not show the scalability of our approach. This will be demonstrated in our future work where we will run further experiments in a larger cloud environment testbed.

6. CONCLUSION AND FUTURE WORK

Flexible and reliable management of SLA agreements represent an open research issue in cloud computing infrastructures. 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, thereby maximizing the revenue of the providers. Moreover, on the basis of flexible and timely reactions to possible SLA violations, interactions with admins can be minimized. In this paper, we presented the LoM2HiS framework—a novel framework for monitoring low-level cloud resource metrics, mapping them to high-level SLA parameters, and using the mapped values and predefined thresholds to monitor the SLA objectives at run time to detect and report SLA violation threats or real violation situations. We also presented an application deployment technique for scheduling and provisioning applications in clouds.

We evaluated our system using a use-case scenario consisting of image-rendering applications based on POV-Ray with heterogeneous workloads. The evaluation is focused on the goal of finding an optimal measurement interval for monitoring application SLA objectives at run time. From our experiments, we observed that there is no particular suited measurement interval for all applications. It is easier to identify optimal measurement intervals for applications with steady resource consumption behaviors, such as the ‘vase’ POV-Ray animation. However, applications with variable resource consumption behaviors require dynamic measurement intervals. Our framework can be extended to tackle such applications, but this will be in the scope of our future work.

Currently, on the FoSII project, we are working toward integrating the knowledge management with the LoM2HiS framework to achieve a complete solution pack for the SLA management. Furthermore, we will design and implement an actuator component for applying the proposed actions from the knowledge database on the cloud resources. We are also adding new physical machines to our evaluation testbed to test the scalability of our approach. Thus, in future, besides our investigation on dynamic measurement intervals, we will also evaluate the influence of such intervals on the quality of the reactive actions proposed by the knowledge database. If the effects of measurement intervals are known, the best reactive actions can be taken, contributing to our vision of flexible and reliable on-demand computing via fully autonomic cloud infrastructures.

ACKNOWLEDGEMENTS

This work is supported by the Vienna Science and Technology Fund (WWTF) under grant agreement ICT08-018 Foundations of Self-governing ICT Infrastructures (FoSII). This paper is a substantially extended version of an HPCS 2010 paper [23]. The experiments were performed in the High Performance Computing Lab at the Catholic University of Rio Grande do Sul (LAD-PUCRS), Brazil. We would like to thank Marco A. S. Netto and Rodrigo N. Calheiros for their support in realizing the experiments.

REFERENCES

1. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 2009; **25**(6):599–616.
2. Chung W-C, Chang R-S. A new mechanism for resource monitoring in grid computing. *Future Generation Computer Systems* 2009; **25**(1):1–7.

DOI: 10.1002/cpe

3. Reyes S, Muoz-Caro C, Nio A, Sirvent R, Badia RM. Monitoring and steering grid applications with grid superscalar. *Future Generation Computer Systems* 2010; **26**(4):645–653.
4. D'Ambrogio A, Bocciarelli P. A model-driven approach to describe and predict the performance of composite services. In *6th International Workshop on Software and Performance (WOSP'07)*, Buenes Aires, Argentina, 2007; 78–89.
5. Gunter D, Tierney B, Crowley B, Holding M, Lee J. Netlogger: a toolkit for distributed system performance analysis. In *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'00)*, San Francisco, California, USA, 2000; 267–273.
6. Kondo D, Fedak G, Cappello F, Chien AA, Casanova H. Characterizing resource availability in enterprise desktop grids. *Future Generation Computer Systems* 2007; **23**(7):888–903.
7. Li C, Li L. Competitive proportional resource allocation policy for computational grid. *Future Generation Computer Systems* 2004; **20**(6):1041–1054.
8. FoSII. Foundations of self-governing infrastructures. (Available from: http://www.infosys.tuwien.ac.at/linksites/FOSII/index.html) [Accessed date:13 June 2012].
9. Glassner AS. *An Introduction to Ray Tracing*. Academic Press: London, 1989.
10. Fu W, Huang Q. GridEye: a service-oriented grid monitoring system with improved forecasting algorithm. In *International Conference on Grid and Cooperative Computing Workshops*, Changsha, Hunan, China, 2006; 5–12.
11. Wood T, Shenoy PJ, Venkataramani A, Yousif MS. Sandpiper: black-box and gray-box resource management for virtual machines. *Computer Networks* 2009; **53**(17):2923–2938.
12. Koller B, Schubert L. Towards autonomous SLA management using a proxy-like approach. *Multiagent Grid Systems* 2007; **3**(3):313–325.
13. Frutos HM, Kotsiopoulos I. BREIN: business objective driven reliable and intelligent grids for real business. *International Journal of Interoperability in Business Information Systems* 2009; **3**(1):39–42.
14. Theilmann W, Yahyapour R, Butler J. Multi-level SLA management for service-oriented infrastructures. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet, ServiceWave '08*, Madrid, Spain, 2008; 324–335.
15. Dobson G, Sanchez-Macian A. Towards unified QoS/SLA ontologies. In *IEEE Services Computing Workshops (SCW'06)*, Chicago, Illinois, USA, 2006; 169–174.
16. Comuzzi M, Kotsokalis C, Spanoudkis G, Yahyapour R. Establishing and monitoring SLAs in complex service based systems. In *IEEE International Conference on Web Services 2009*, Los Angeles, CA, USA, 1009.
17. Boniface M, Phillips SC, Sanchez-Macian A, Surridge M. Dynamic service provisioning using GRIA SLAs. In *International Workshops on Service-Oriented Computing (ICSOC'07)*, Vienna, Austria, 2007; 56–67.
18. Rosenberg F, Platzer C, Dustdar S. Bootstrapping performance and dependability attributes of web services. In *IEEE International Conference on Web Services (ICWS'06)*, Chicago, USA, 2006; 205–212.
19. Brandic I, Music D, Leitner P, Dustdar S. Vieslaf framework: enabling adaptive and versatile SLA-management. In *Proceedings of the 6th International Workshop on Grid Economics and Business Models, GECON'09*, Delft, The Netherlands, 2009; 60–73.
20. BREIN. Business objective driven reliable and intelligent grids for real business. (Available from: http://www.eu-brein.com/) [Accessed date:13 June 2012].
21. Kephart JO, Chess DM. The vision of autonomic computing. *IEEE Computer* 2003; **36**(1):41–50.
22. Brandic I. Towards self-manageable cloud services. In *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, Seattle, Washington, USA, 2009; 128–133.
23. Emeakaroha VC, Brandic I, Maurer M, Dustdar S. 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)*, Caen France, 2010; 48–54.
24. Maurer M, Brandic I, Sakellariou R. Simulating autonomic SLA enactment in clouds using case based reasoning. In *ServiceWave 2010: Proceedings of the 2010 ServiceWave Conference*, Ghent, Belgium, 2010; 25–36.
25. Maurer M, Brandic I, Sakellariou R. Enacting SLAs in clouds using rules. In *Proceedings of Euro-par 2011*, Bordeaux, France, 2011; 455–466.
26. FreeCBR. FreeCBR. (Available from: http://freecbr.sourceforge.net/) [Accessed date:13 June 2012].
27. Aamodt A, Plaza E. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications* 1994; **7**:39–59.
28. Maurer M, Brandic I, Emeakaroha VC, Dustdar S. Towards knowledge management in self-adaptable clouds. In *4th International Workshop of Software Engineering for Adaptive Service-Oriented Systems (SEASS'10)*, Miami Florida, USA, 2010; 527–534.
29. Elmroth E, Tordsson J. A grid resource broker supporting advance reservations and benchmark-based resource selection. *Proceedings of the 7th international conference on Applied Parallel Computing: state of the Art in Scientific Computing PARA 2004*, Lyngby, Denmark, 2006; 1061–1070.
30. Abramson D, Buyya R, Giddy J. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems* 2002; **18**(8):1061–1074.
31. Krauter K, Buyya R, Maheswaran M. A taxonomy and survey of grid resource management systems for distributed computing. *Software—Practice and Experience* 2002; **32**(2):135–164.
32. Massie ML, Chun BN, Culler DE. The GANGLIA distributed monitoring system: design, implementation and experience. *Parallel Computing* 2004; **30**(7):817–840.

DOI: 10.1002/cpe

33. SAX. Simple API for XML. (Available from: http://sax.sourceforge.net/) [Accessed date:13 June 2012].
34. JMS. Java Messaging Service. (Available from: http://java.sun.com/products/jms/) [Accessed date:13 June 2012].
35. ActiveMQ. Messaging and integration pattern provider. (Available from: http://activemq.apache.org/) [Accessed date:13 June 2012].
36. ESPER. Event stream processing. (Available from: http://esper.codehaus.org/) [Accessed date:13 June 2012].
37. Hibernate. Relational persistence for Java and .NET. (Available from: http://www.hibernate.org/) [Accessed date:13 June 2012].
38. Calheiros RN, Buyya R, De Rose CAF. Building an automated and self-configurable emulation testbed for grid applications. *Software: Practice and Experience* 2010; **40**(5):405–429.
39. Lee K, Paton NW, Sakellariou R, Alvaro AAF. Utility driven adaptive workflow execution. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society: Washington, DC, USA, 2009; 220–227.
40. Lee CB, Snavely A. On the user-scheduler dialogue: studies of user-provided runtime estimates and utility functions. *International Journal of High Performance Computing Applications* 2006; **20**(4):495–506.
41. Yeo CS, Buyya R. Pricing for utility-driven resource management and allocation in clusters. *International Journal of High Performance Computing Applications* 2007; **21**(4):405–418.

DOI: 10.1002/cpe