Author Name

Cloud Computing: Methodology, System, and Applications

Foreward

Preface

Contributors

List of Figures

List of Tables

Contents

Symbol Description

α	To solve the generator maintenance scheduling, in the past, several mathematical techniques have been applied.

Part I

This is a Part

1

SOA and QoS Management for Cloud Computing

CONTENTS

1.1 Introduction

In the recent years, Cloud computing has become a key IT megatrend that will take root, although it is at 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 IT management to enable a more flexible approach for deploying and scaling applications at low cost [8].

Service provisioning in the Cloud is based on Service Level Agreements (SLA), which is a set of non-functional properties specified and negotiated between the customer and the service provider. 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 for both Cloud providers and consumers. On the one hand, the prevention of SLA violations avoids penalties that are costly to providers. On the other hand, based on flexible and timely reactions to possible SLA violation threats, user interaction with the system can be minimized enabling Cloud computing to take roots as a flexible and reliable form of on-demand computing.

In order to guarantee an agreed SLA, the Cloud provider must be capable of monitoring its infrastructure (host) resource metrics to enforce the agreed service level objectives. Traditional monitoring technologies for single machines or Clusters are restricted to locality and homogeneity of monitored objects and, therefore, cannot be applied in the Cloud in an appropriate manner. Moreover, in traditional systems there is a gap between monitored metrics, which are usually low-level entities, and SLA agreements, which are high-level user guarantee parameters.

In this book chapter we present a novel framework for the mapping of Low-level resource Metric to High-level SLA parameters named LoM2HiS framework, which is also capable of evaluating application SLA at runtime and detecting SLA violation situations in order to ensure the application QoS. Furthermore, we present a knowledge management technique based on Case-Based Reasoning (CBR) that is responsible for proposing reactive actions to prevent or correct detected violation situations.

The LoM2HiS framework is embedded into FoSII infrastructure aiming at developing an infrastructure for autonomic SLA management and enforcement. Thus, LoM2HiS represents the first building block of the FoSII [16] infrastructure. We present the conceptual design of the framework including the run-time and host monitors, and the SLA mapping database. We discuss our novel communication model based on queuing networks ensuring the scalability of the LoM2HiS framework. Moreover, we demonstrate sample mappings from the low-level resource metrics to the high-level SLA parameters. Thereafter, we discuss some details of the CBR knowledge management technique thereby showing how the cases are formulated and their similarities. Furthermore, we describe a utility function for calculating the quality of a proposed action.

The main contributions of this book chapter are: (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 low-level metrics to high-level SLA objectives; (iv) the evaluation of the SLA at run-time to detect violation threats or real violation situation; (v) the design of the knowledge management technique; (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 POV-Ray [19].

The rest of the book chapter is organized as follows. Section 1.2 presents the related work. In Section 1.3 we present the background and motivation for

this research work. The conceptual design and implementation issues of the LoM2HiS framework is presented in Section 1.4. In Section 1.5 we give the details of the case-based reasoning knowledge management technique. Section 1.6 deals with the framework evaluation based on a real Cloud testbed with POV-Ray applications and the discussion of the achieved results. Section 1.7 presents the conclusion of the book chapter and our future research work.

1.2 Related Work

We classify related work on SLA management and enforcement of Cloud based services into (i) Cloud resource monitoring [18, 20, 32] (ii) SLA management including QoS management [4, 10, 12, 17, 24, 31] and (iii) mapping techniques of monitored metrics to SLA parameters and attributes [6, 11, 29]. Since there is very little work on monitoring, SLA management, and metrics mapping in Cloud systems we look particularly into related areas such as Grid and 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 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.* [32] 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.* [4] 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 environments. Moreover, they do not detail how the low-level metric are monitored and mapped to high-level SLAs. Theilman et al. [31] discuss an approach for multi-level SLA management, where SLAs are consistently specified and managed within a service-oriented infrastructure (SOI). They present the runtime functional view of the conceptual architecture and discuss different case studies including Enterprise Resource Planning (ERP) or financial services. But they did not address low-level resource monitoring and SLA mappings.

Koller et al. [24] discuss 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 like scalability. Frutos et al. [17] discuss the main approach of the EU project BREIN [7] to develop a framework, which 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. [12] present a unified quality of service (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. 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 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. But they did not consider monitoring of low-level metrics and mapping them to high-level SLA parameters for ensuring the SLA objectives.

Brandic et al. [6] present an approach for adaptive generation of SLA templates. Thereby, SLA users can define mappings from their local SLA templates to the remote templates in order to facilitate communication with numerous Cloud service providers. However, they do not investigate mapping of monitored metrics to agreed SLAs. Rosenberg et al. [29] deal 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 resource metrics. Bocciarelli et al. [11] introduce a model-driven approach for integrating performance prediction into service composition processes carried out using BPEL. In their approach, they compose service SLA parameters from resource metrics using some mapping techniques. But they did neither consider 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 as those necessary in Cloud-like environments.

1.3 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 [8, 23]. 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 [5].

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* [13], which is a core component of the FoSII infrastructure 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.

1.3.1 FoSII Infrastructure Overview

Figure 1.1 presents an overview 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 1.1 is implemented by each Cloud service and specifies operations for sensing changes of the desired state and for reacting to those changes [5]. The host monitor sensors continuously monitor the infrastructure resource metrics (input sensor values arrow *a* in Figure 1.1) 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 1.1) 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 book chapter we focus on the LoM2HiS framework and give some details of the knowledge management technique since they are responsible for system monitoring, detection of SLA violations and proposing of reactive actions to prevent or correct the violation situation.

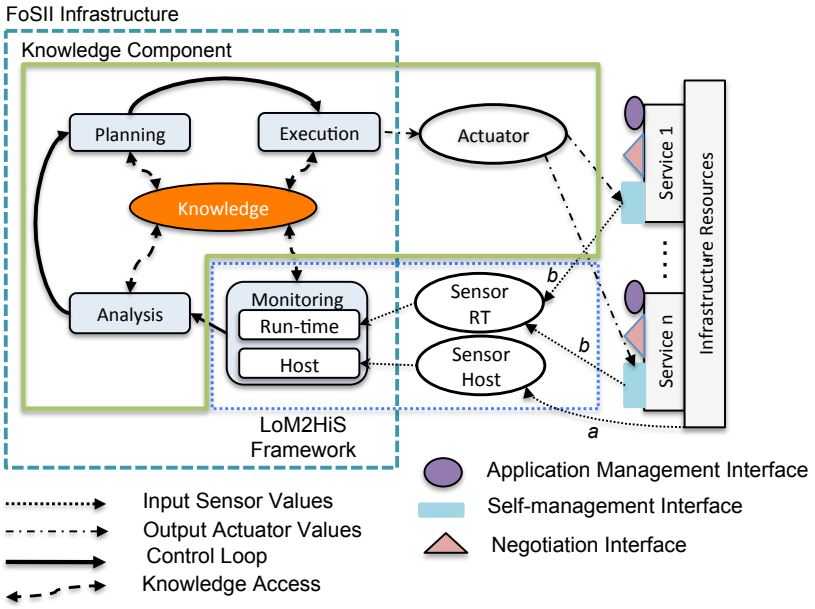


FIGURE 1.1
FoSII Infrastructure Overview

1.4 Design of the LoM2HiS Framework

The LoM2HiS framework is the first step towards achieving the goals of the FoSII infrastructure. In this section, we give the details of the LoM2HiS framework and in the sections below we describe the components and their implementations.

In this framework, we assumed that the SLA negotiation process is completed and the agreed SLAs are stored in the repository for service provisioning. Beside the SLAs, the predefined threat thresholds for guiding the SLA objectives are also stored in a repository. The concept of detecting future SLA violation threats is designed by defining more restrictive thresholds known as threat thresholds that are stricter than the normal SLA objective violation thresholds. In this book chapter 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 1.2 presents the architecture of our LoM2HiS framework. The service component including the run-time monitor represents the application layer where services are deployed using a Web Service container e.g., Apache Axis. The run-time monitor is designed to monitor the services based on the

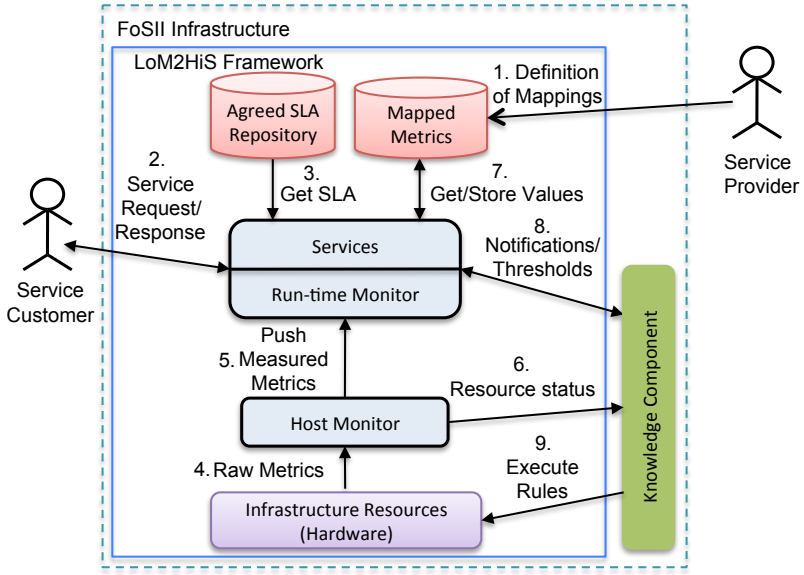


FIGURE 1.2
LoM2HiS Framework Architecture

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 1.2) using Domain Specific Languages (DSLs). DSLs 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 service SLA from the agreed SLA repository (step 3). Service provisioning is based on the infrastructure resources, which represent the physical-, virtual machines, and network resources in a data centre for hosting Cloud services. The resource metrics are measured by monitoring agents, and the measured raw metrics are accessed by the host monitor (step 4). The host monitor extracts metric-value pairs from the raw metrics and transmits them periodically to the run-time monitor (step 5) and to the knowledge component (step 6) using our designed communication mechanism.

Upon receiving 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 mapping results are stored in the mapped metric repository (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 com-

ponent works out an appropriate preventive action to avert future SLA violation threats based on the resource status (step 6) and defined rules [28]. The knowledge components decisions (e.g., assign more CPU to a virtual host) are executed on the infrastructure resources (step 9).

1.4.1 Host Monitor

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

1.4.1.1 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 1.3 presents the host monitoring system.

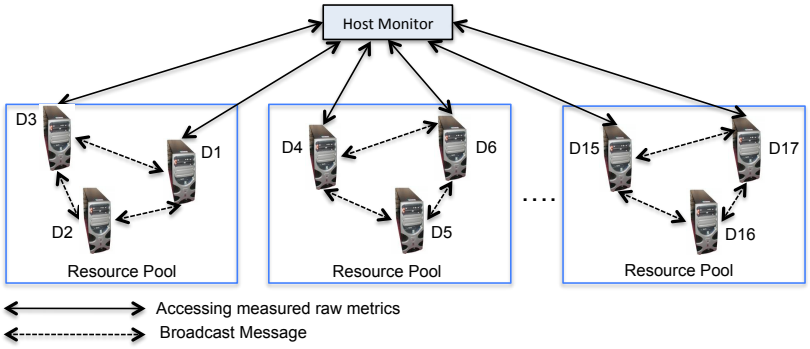


FIGURE 1.3
Host Monitoring System

As shown in Figure 1.3, the monitoring agent embedded in Device 1 (D1) measures its resource metrics and broadcasts them to D2 and D3. Equally, 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 other. This eradicates the problem of a bottleneck system and offers fault-tolerant capabilities. Note that a device can be a physical machine, a virtual machine, a storage device, or a network device. It should also be further noted that the above described broadcasting mechanism is configurable and can be deactivated in a Cloud environment where there are

lots of devices within resource pools to avoid communication overheads, which may consequently lead to degraded overall system performance.

1.4.1.2 Implementation of Host Monitor Component

The host monitor implementation uses the GMOND module from the GANGLIA open source project [27] as the monitoring agent. The GMOND module is a standalone component of the GANGLIA project. We use it to monitor the infrastructure resource metrics. The monitored results are presented in an XML file and written to a predefined network socket. We 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. Furthermore, we implemented an XML parser using the well-known open source SAX API [30] to parse the XML file in order 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 done once or repeated periodically depending on the monitoring strategy being used.

1.4.2 Communication Mechanism

The components of our FoSII infrastructure 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. Figure 1.4 presents an example scenario expressing the usage of the communication mechanism.

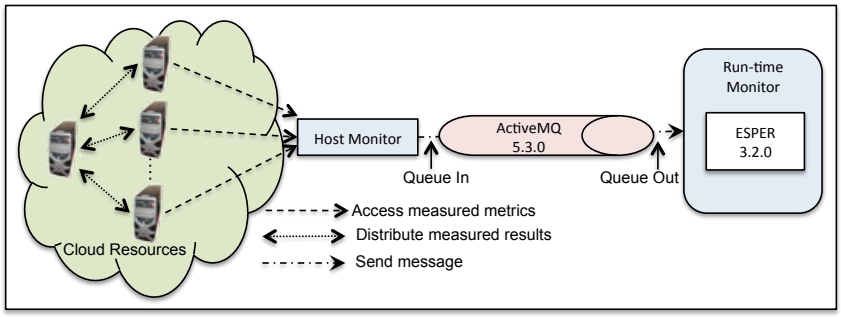


FIGURE 1.4
Communication Mechanism Scenario

The scenario of Figure 1.4 depicts the processes of extracting the low-level metrics from the monitoring agents embedded in the Cloud resources and passing them to the run-time monitor for mapping and further processing.

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 (MOM) API for sending messages between two or more clients [22]. In order for us to use JMS,

we need a JMS provider that manages the sessions and queues. Thus, we use the well-established open source Apache ActiveMQ [3] for this purpose.

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

1.4.3 Run-Time Monitor

The run-time monitor component, which is located at the application level in a Cloud environment is presented in this section. We first describe the design of the component and later explain its implementation details.

1.4.3.1 Run-Time Monitor Design

The run-time monitor performs the mappings and based on 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 repository; 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.

Mapping of low-level metric to high-level SLAs: As already discussed in Section 1.4, the run-time monitor chooses the mapping rules to apply based on the service being provisioned. That is for each service 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 low-level to high-level, as for example mapping low-level metric *disk space* to 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. Table 1.1 presents some complex mapping rules.

In the mapping rules presented in Table 1.1, the downtime variable represents the *mean time to repair (MTTR)*, 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 (MTBF)*, which denotes the time the system was operational between the last system failure to the next. R_{in} is the response time for a service request and is calculated as $\frac{packetsize}{availablebandwidthin-inbytes}$ in milliseconds. R_{out} is the response time for a service response and is calculated

TABLE 1.1
Complex Mapping Rules.

Resource Metrics	SLA Parameter	Mapping Rule
downtime, uptime	Availability (A)	$A = 1 - \frac{downtime}{uptime}$
inbyte, outbytes, packetsize, avail.bandwidthin, avail.bandwidthout	Response Time (R_{total})	$R_{total} = R_{in} + R_{out}$ (ms)

as $\frac{packetsize}{availablebandwidthout - outbytes}$ in milliseconds. The mapped SLAs are stored in the mapped metric repository for usage during the monitoring phase.

Monitoring SLA objectives and notifying the knowledge component: In this phase the run-time monitor accesses the mapped metrics' repository to get 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 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 probably occur if the system is unable to resolve the cause of a violation threat notification.

1.4.3.2 Implementation of Run-Time Monitor Component

The run-time monitor receives the measured metric-value pairs and passes them into the Esper engine [15] for further processing. Esper is a component for CEP and ESP applications, available for Java as Esper, and for .NET as NEsper. Complex Event Processing (CEP) is a technology to process events and discover complex patterns among multiple streams of event data. Event Stream Processing (ESP) 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 drastically reduces 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.

1.5 Knowledge Management

In this section we give some details about Case-Based Reasoning (CBR), which is the knowledge management technique we are currently investigating for proposing reactive actions to SLA violation threats or real violation situations. CBR was first built on top of FreeCBR [1], but is now a completely independent Java framework taking into account, however, basic ideas of FreeCBR. We first explain the ideas behind CBR, describe how to infer the similarity of two cases, and finally derive a utility function to estimate the "goodness" of a reactive action in a specific situation.

1.5.1 Case-Based Reasoning Overview

Case-Based Reasoning is the process of solving problems based on past experiences [2]. 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.

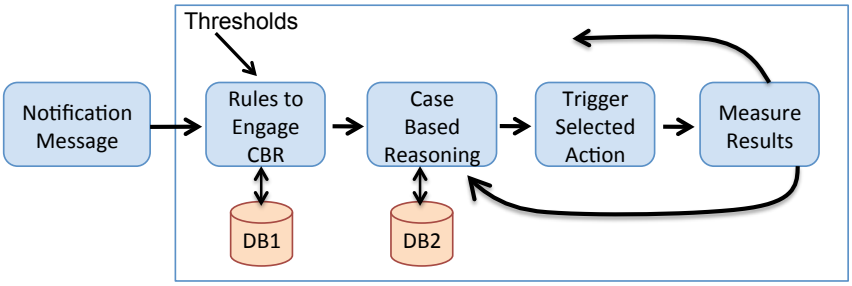


FIGURE 1.5
Case-Based Reasoning Process Overview

As shown in Figure 1.5 the ideas of using CBR in SLA management is to have rules stored in a database that engage the CBR system once a threshold value has been reach for a specific SLA parameter. The notification information are fed into the CBR system as new cases by the monitoring component. Then, CBR prepared with some initial meaningful cases stored in database 2 (Figure 1.5), chooses the set of cases, which are most similar to the new case by various means as described in Section 1.5.2. 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 to 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 new case and corresponding solution into knowledge database.)

In order to relate the cases to SLA agreement, we formalize the language elements used in the remaining of the book chapter. Each SLA has a unique identifier id and a collection of Service Level Objectives (SLOs), which are predicates of the form

$$SLO_{id}(x_i, comp, \pi_i) \text{ with } comp \in \{<, \leq, >, \geq, =\}, \tag{1.1}$$

where $x_i \in P$ represents the parameter name for $i = 1, \ldots, n_{id}$, π_i the parameter goal, and $comp$ the appropriate comparison operator. Additionally, action guarantees that state the amount of penalty that has to be paid in case of a violation can be added to SLOs, which is out of scope in this book chapter. Furthermore, a case c is defined as

$$c = (id, m_1, p_1, m_2, p_2, \ldots, m_{n_{id}}, p_{n_{id}}), \tag{1.2}$$

where id represents the SLA id, and m_i and p_i the measured (m) and provided (p) value of the SLA parameter x_i, respectively. The measured value (m) indicates the current amount of this specific Cloud resource used by the running application and the provided value (p) shows the amount of this specific Cloud resource allocated to this application. These two parameters are paramount for efficient Cloud resource management in proposing reactive actions to prevent or correct SLA violation situation.

A typical use case for the evaluation might be: SLA id = 1 with SLO_1("Storage", $\geq$, 1000, ag_1) and SLO_1 ("Bandwidth", $\geq$, 50.0, ag_1), where ag_1 stands for the appropriate preventive action to execute after an SLO violation. A simple case that can be notified by the measurement component would therefore look like $c = (1, 500, 700, 20.0, 30.0)$. A result case $rc = (c^-, ac, c^+, utility)$ includes the initial case c^-, the executed action ac, the resulting case c^+ measured after some time interval later and the calculated $utility$ as described in Section 1.5.3.

1.5.2 Inferring Similarity of two Cases

To retrieve similar cases already stored in the database in order to propose an action for a new case, the similarity of the two cases has to be calculated. However, there are many metrics that can be considered in this process.

We approach this problem using a strategy similar to Euclidean distance. However, the problem with Euclidean distance, for instance, is due to its symmetric nature and therefore cannot correctly fetch whether a case is in a state of over- or under-provisioning. Additionally, the metric has to treat parameters in a normalized way so that parameters that have a larger distance range are not over-proportionally taken into account than parameters with a smaller difference range. For example, if the difference between measured and provided values of parameter A always lie between 0 and 100 and of parameter B between 0 and 1000, the difference between an old and a new case can only be within the same ranges, respectively. Thus, just adding the differences of the parameters would yield an unproportional impact on parameter B.

This leads to the following equation whose summation part follows the principle of semantic similarity [21]:

$$d(c^-, c^+) = \min(w_{id}, \left|id^- - id^+\right|) + \sum_{x \in P} w_x \left| \frac{(p_x^- - m_x^-) - (p_x^+ - m_x^+)}{max_x - min_x} \right|, \quad (1.3)$$

where $w = (w_{id}, w_{x_1}, \ldots, w_{x_n})$ is the weight vector; w_{id} is the weight for non-identical SLAs; w_x is the weight, and max_x and min_x the maximum and minimum values of the provided and measured resource differences $p_x - m_x$ for parameter x. As it can be easily checked, this indeed is a metric also in the mathematical sense.

Furthermore, the match percentage mp of two cases c^- and c^+ is then calculated as

$$mp(c^-, c^+) = \left(1 - \frac{d(c^-, c^+)}{w_{id} + \sum_x w_x}\right) \cdot 100. \quad (1.4)$$

This is done because the algorithm does not only consider the case with the highest match, but also cases in a certain percentage neighborhood (initially set to 3%) of the case with the highest match. From these cases the algorithm then chooses the one with the highest utility. By calculating the match percentage, the cases are distributed on a fixed line between 0 and 100, where 100 is an identical match, whereas 0 is the complete opposite.

1.5.3 Utility Function and Resource Utilization

To calculate the utility of an action, we have to compare the initial case c^- vs. the resulting final case c^+. The *utility function* presented in Equation 1.5 is composed by a SLA violation and a resource utilization term weighed by the factor $0 \leq \alpha \leq 1$:

$$utility = \sum_{x \in P} violation(x) + \alpha \cdot utilization(x) \tag{1.5}$$

Higher values for α indicate desire to achieve high resource utilization, whereas lower values implies the wish of non-violation of SLA parameters. Thus, to achieve a balance, a trade-off must be found. We further note that $c(x)$ describes a case only with respect to parameter x. E.g., we say that a violation has occurred in $c(x)$, when in case c the parameter x was violated.

The function *violation* for every parameter x is defined as follows:

$$violation(x) = \begin{cases} 1, & \text{No violation occurred in } c^+(x), \text{ but in } c^-(x) \\ 1/2, & \text{No violation occurred in } c^+(x) \text{ and } c^-(x) \\ -1/2 & \text{Violation occurred in } c^+(x) \text{ and } c^-(x) \\ -1 & \text{Violation occurred in } c^+(x), \text{ but not in } c^-(x) \end{cases} \tag{1.6}$$

For the *utilization* function we calculate the utility from the used resources in comparison to the provided ones. We define the distance $\delta(x, y) = |x - y|$, and utilization for every parameter as

$$utilization(x) = \begin{cases} 1, & \delta(p_x^-, m_x^-) > \delta(p_x^+, u_x^+) \\ -1, & \delta(p_x^-, m_x^-) < \delta(p_x^+, u_x^+) \\ 0, & \text{otherwise.} \end{cases} \tag{1.7}$$

We get a utilization utility of 1 if we experience less over-provisioning of resources in the final case than in the initial one, and a utilization utility of -1 if we experience more over-provisioning of resources in the final case than in the initial one.

We map utilization, u, and the number of SLA violations, v, into a scalar called *Resource Allocation Efficiency* (RAE) using the following equaltion

$$\text{RAE} = \begin{cases} \frac{u}{v}, & v \neq 0 \\ u, & v = 0, \end{cases} \tag{1.8}$$

which represents an important goals for the knowledge management. High utilization leads to high RAE, whereas a high number of SLA violations leads to a low RAE, even if utilization is in normal range. This can be explained by the fact that having utilization at a maximum - thus being very resource efficient in the first place - does not pay if the SLA is not fulfilled at all.

1.6 Evaluations

In this section we present an image rendering use-case scenario to evaluate our approach in this book chapter. The knowledge management technique is developed as an independent work and has not yet been integrated with the LoM2HiS framework. Thus our evaluations here covers only the monitoring framework.

The goal of our evaluations is to determine the optimal measurement interval for monitoring agreed SLA objectives for applications at runtime. We first present our real Cloud experimental environment after which we discuss in details the use-case scenario.

1.6.1 Experimental Environment

The resource capacities of our Cloud experimental testbed is shown in Table 1.2. The table shows the resource compositions of the physical - and the virtual machines being used in our experimental setups. We use Xen virtualization technology in our testbed, precisely we run Xen 3.4.0 on top of Oracle Virtual Machine (OVM) server.

TABLE 1.2
Cloud Environment Resource Setup Composed of 10 Virtual Machines.

Machine Type = Physical Machine

OS	CPU	Cores	Memory	Storage
OVM Server	Pentium 4 2.8 GHz	2	2.5 GB	100 GB

Machine Type = Virtual Machine

OS	CPU	Cores	Memory	Storage
Linux/Ubuntu	Pentium 4 2.8 GHz	1	1 GB	5 GB

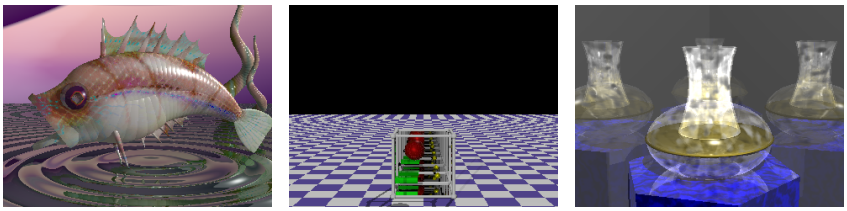
We have in total five physical machines and, based on their resource capacities as presented in Table 1.2, we host two VMs on each physical machine. We use an Automated Emulation Framework (AEF) [9] to deploy the VMs onto the physical hosts, thus creating a virtualized Cloud environment with up to ten computing nodes capable of provisioning resources to applications and one front-end node responsible for management activities.

The front-end node serves as the control entity. It runs the automated emulation framework, the application deployer [14] responsible for moving the application data to the virtual machines, and the LoM2HiS framework to monitor and detect SLA violation situations. We use this Cloud environment to evaluate the use case scenario presented in the next section.

1.6.2 Image Rendering Application Use-Case Scenario

We developed an image rendering application based on the Persistence of Vision Raytracer (POV-Ray)[1], which is a ray tracing program available for several computing platforms [19]. In order to achieve heterogeneous load in this use-case scenario, we experiment with three POV-Ray workloads, each one with a different characteristic of time for rendering frames, as described below and illustrated in Figures 1.6 and 1.7:

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



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

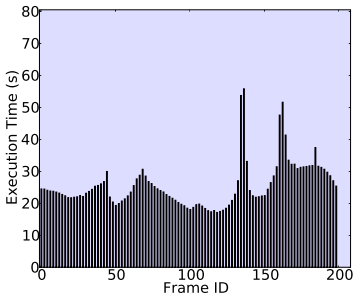
FIGURE 1.6
Example of images for each of the three animations.

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 1.3 presents the SLA objectives for each of the applications. These SLA objective thresholds are defined based on 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 application requires. Thus, the provider can make better resource provisioning plan for the applications.

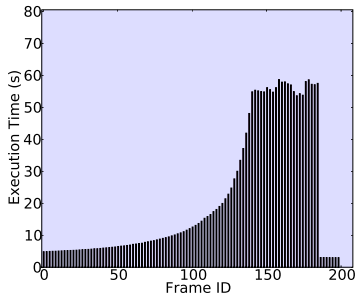
Based on these SLA objectives, the applications are monitored to detect SLA violations. These violations may happen because SLAs are negotiated per application and not per allocated VM considering the fact that the service provider may provision different application requests on the same VM.

Figure 1.8 presents the evaluation configurations for the POV-Ray applications. We instantiate 10 virtual machines that execute POV-Ray frames

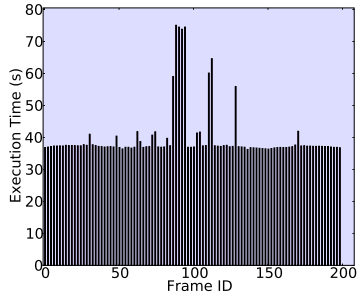
[1] www.povray.org



(a) Fish.



(b) Box.



(c) Vase.

FIGURE 1.7
Behavior of execution time for each POV-Ray application.

submitted via Application Deployer. The virtual machines are continuously monitored by Gmond. Thus, LoM2HiS has access to resource utilization during execution of the applications. Similarly, information about the time taken to render each frame in each virtual machine is also available to LoM2HiS. This information is generated by the application itself and is sent to a location where LoM2HiS can read it. As described in Figure 1.8, users supply the QoS requirements in terms of SLOs (step 1 in Figure 1.8). 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 book chapter and is discussed by Brandic *et al.* [5]. Thereafter, VM deployer starts configuration and allocation of the required VMs whereas application deployer maps the tasks to the appropriate VMs (step 3). In step 4 the application execution is triggered.

TABLE 1.3
POV-Ray Applications SLA Objective Thresholds

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

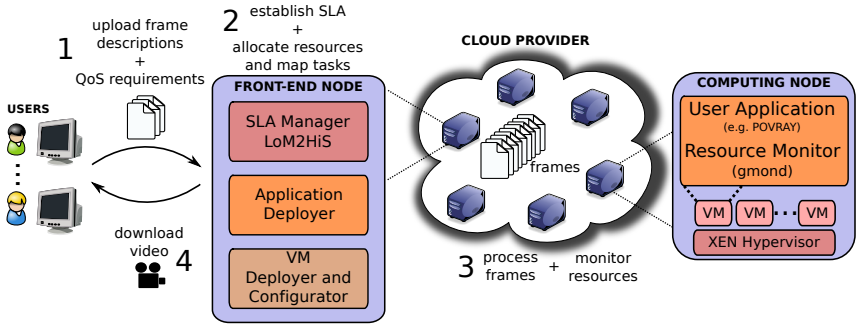


FIGURE 1.8
Pov-Ray Evaluation Configuration.

1.6.3 Achieved Results and Analysis

We defined and used six measurement intervals to monitor the POV-Ray applications during their executions. Table 1.4 shows the measurement intervals and the number of measurements made in each interval. The applications run for about 20 minutes for each measurement interval.

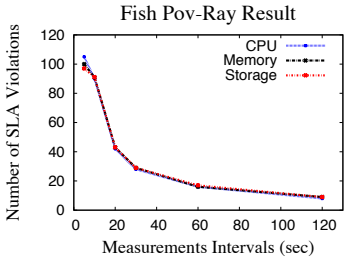
TABLE 1.4
Measurement Intervals.

Intervals	5s	10s	20s	30s	60s	120s
Nr. of Measurements	240	120	60	40	20	10

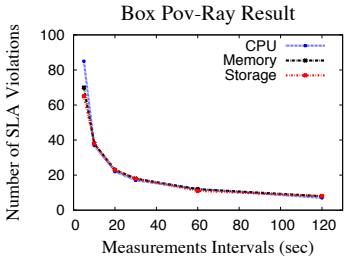
The 5 seconds measurement interval is a reference interval meaning the current interval used by the provider to monitor application executions on the Cloud resources. Its results show the present situation of the Cloud provider.

Figure 1.9 presents the achieved results of the three POV-Ray applications with varying characteristics in terms of frame rendering as explained in Section 1.6.2. We use the 10 virtual machines in our testbed to simultaneously execute the POV-Ray frames. There is a load-balancer integrated in the application deployer, which ensures that the frame executions are balanced among the virtual machines.

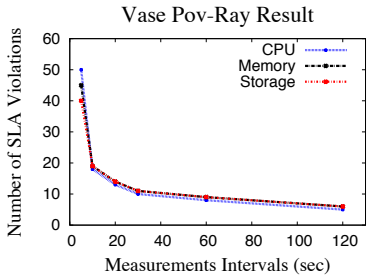
The *LoM2HiS* framework monitors the resource usage of each virtual machine to determine if the SLA objectives are ensured and reports violations otherwise. Since the load-balancer balances the execution of frames among the virtual machines, we plot in Figure 1.9 the average numbers of violations encountered in the testbed for each application with each measurement interval.



(a) Fish.



(b) Box.



(c) Vase.

FIGURE 1.9
POV-Ray experimental results

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

Using these two factors and other parameters we define a cost function (C) based on which we can derive an optimal measurement interval. The ideas of

defining this cost functions are derived from utility functions discussed by Lee *et al.* [26]. Equation 1.9 presents the cost function.

$$C = \mu * C_m + \sum_{\psi \epsilon \{cpu, memory, storage\}} \alpha(\psi) * C_v \tag{1.9}$$

where μ is the number of measurements, C_m is the cost of measurement, $\alpha(\psi)$ is the number of undetected SLA violations, and C_v is the cost of missing an SLA violation. The number of undetected SLA violations are determined based on the results of the 5 seconds reference measurement interval, which is assumed to be an interval capturing all the violations of applications' SLA objectives. This cost function now forms the basis for analyzing the achieved results of our use-case scenario.

The cost of making measurement in our testbed is defined by considering the intrusiveness of the measurements on the overall performance of the system. Based on our testbed architecture and 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 while the services are hosted in the computing node. The monitoring agents running on computing nodes have minimal impact on resource consumption. This means a low cost of making measurements in the Cloud environment.

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

Applying the cost function on the achieved results of Figure 1.9, with a measurement cost of 0.5 dollar and an aggregated missing violation cost of 1.5 dollar, we achieve the monitoring costs presented in Table 1.5. 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 literature [25, 33].

The monitoring cost presented in Table 1.5 represents the cost of measurement for each interval 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 1.5, 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 the Cloud testbed.

The relations of the measurement cost and the cost of missing SLA violation detection is graphically depicted in Figure 1.10 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

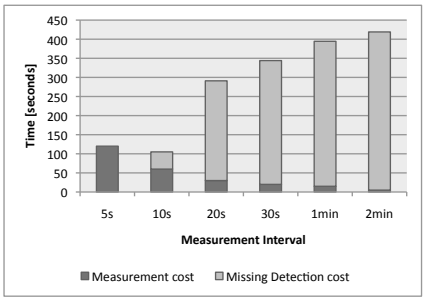
TABLE 1.5
Monitoring Cost.

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

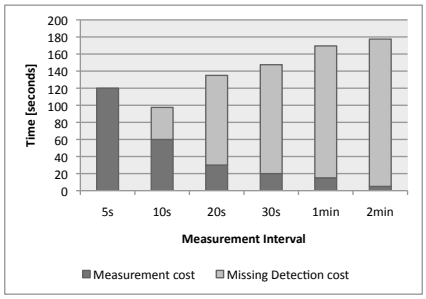
detected, the higher the detection cost rises. This implies that to keep the detection cost low, the number of missed SLA violation must be low.

Considering the total cost of monitoring the fish POV-Ray application in Table 1.5 and Figure 1.10(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 1.5 and depicted graphically in Figure 1.10(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 1.5 and Figure 1.10(c), it is clear that the reference measurement by far does not represent 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.

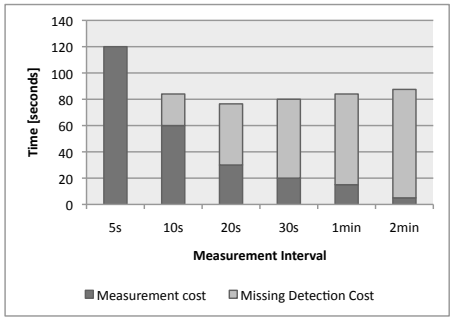
Based on our experiments, it is observed that there is no best suited measurement interval for all applications. Depending on how steady the resource consumption is, the monitoring infrastructure requires different measurement intervals. Note that the architecture can be configured to work with different intervals. In this case, specification of the measurement frequencies depends on policies agreed by users and providers.



(a) Fish.



(b) Box.



(c) Vase.

FIGURE 1.10
POV-Ray application cost relations.

1.7 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 thereby maximizing the profit of the provider. Moreover, based on flexible and timely reactions to possible SLA violations, interactions with users can be minimized. In this book chapter we presented *LoM2HiS framework*— a novel framework for monitoring low-level Cloud resource metrics, mapping them to high-level SLA parameters, using the mapped values and predefined thresholds to monitor the SLA objectives at runtime, detecting and reporting SLA violation threats or real violations situations. We also presented the knowl-

edge management technique based on Case-Based Reasoning for managing the SLA violation situation and proposing preventive or corrective actions.

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 runtime. From our experiments we observed that there is no particular suited measurement interval for all applications. It is easier to identify optimal 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 framework can be extended to tackle such applications but this will be in the scope of our future work.

Currently on FoSII project, we are working toward integrating the knowledge management with the LoM2HiS framework in order 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.

Thus, in the future besides our investigation on dynamic measurement intervals, we will 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, best reactive actions may be taken, contributing to our vision of flexible and reliable on-demand computing via fully autonomic Cloud infrastructures.

Acknowledgments

This work is supported by the Vienna Science and Technology Fund (WWTF) under grant agreement ICT08-018 Foundations of Self-governing ICT Infrastructures (FoSII). We would like to thank Marco A. S. Netto for his support in carrying out the evaluations. The experiments were performed in the High Performance Computing Lab at Catholic University of Rio Grande do Sul (LAD-PUCRS)Brazil.

Bibliography

[1] FreeCBR, http://freecbr.sourceforge.net/.

[2] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7:39–59, 1994.

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

[4] 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.

[5] Ivona. Brandic. Towards self-manageable cloud services. In *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, 2009.

[6] Ivona Brandic, Dejan Music, Philipp Leitner, and Schahram Dustdar. Vieslaf framework: Enabling adaptive and versatile sla-management. In *Proceedings of the 6th International Workshop on Grid Economics and Business Models*, GECON '09, pages 60–73, 2009.

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

[8] 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.

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

[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] Vincent. 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.

[14] Vincent C. Emeakaroha, Rodrigo N. Calheiros, Marco A. S. Netto, Ivona Brandic, and César A. F. De Rose. DeSVi: An architecture for detecting SLA violations in cloud computing infrastructures. In *Proceedings of the 2nd International ICST Conference on Cloud Computing (CloudComp'10)*, 2010.

[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] Mark Hefke. A framework for the successful introduction of KM using CBR and semantic web technologies. *Journal of Universal Computer Science*, 10(6), 2004.

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

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

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

[25] Cynthia Bailey Lee and Allan 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] Kevin Lee, Norman W. Paton, Rizos Sakellariou, and A. A. Fernandes 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] 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.

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

[31] Wolfgang Theilmann, Ramin Yahyapour, and Joe Butler. Multi-level sla management for service-oriented infrastructures. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet*, ServiceWave '08, pages 324–335, 2008.

[32] 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.

[33] Chee Shin Yeo and Rajkumar Buyya. Pricing for utility-driven resource management and allocation in clusters. *Int. J. High Perform. Comput. Appl.*, 21(4):405–418, 2007.