Future Generation Computer Systems 54 (2016) 233–246

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

SLA guarantees for cloud services

Damián Serrano[a], Sara Bouchenak[a], Yousri Kouki[b], Frederico Alvares de Oliveira Jr.[b], Thomas Ledoux[b], Jonathan Lejeune[c], Julien Sopena[c], Luciana Arantes[c], Pierre Sens[c,*]

[a] University of Grenoble, France

[b] EMN – INRIA – LINA, Nantes, France

[c] Sorbonne Universités, UPMC, CNRS, Inria, Paris, France

HIGHLIGHTS

- We provide a domain specific language that allows to describe SLA in cloud services.
- We present a general control-theoretic approach for managing cloud service SLA.
- We apply our approach on MapReduce, locking, and e-commerce services.

ARTICLE INFO

Article history:
Received 30 September 2014
Received in revised form
20 March 2015
Accepted 23 March 2015
Available online 17 April 2015

Keywords:
SLA
QoS
Cloud Computing
Specific language
Online control

ABSTRACT

Quality-of-service and SLA guarantees are among the major challenges of cloud-based services. In this paper we first present a new cloud model called *SLAaaS — SLA aware* Service. *SLAaaS* considers QoS levels and SLA as first class citizens of cloud-based services. This model is orthogonal to other SaaS, PaaS, and IaaS cloud models, and may apply to any of them. More specifically we make three contributions: (i) we provide a novel domain specific language that allows to describe QoS-oriented SLA associated with cloud services; (ii) we present a general control-theoretic approach for managing cloud service SLA; (iii) we apply the proposed language and control approach to guarantee SLA in various case studies, ranging from cloud-based MapReduce service, to locking service, and higher-level e-commerce service; these case studies successfully illustrate SLA management with different QoS aspects of cloud services such as performance, dependability, financial energetic costs.

1. Introduction

Cloud Computing is nowadays a widely extended computation paradigm. It enables remote and on demand access to configurable computing resources providing hardware and software services in a way that it minimizes the human efforts needed by customers as well as providers to configure, use and maintain the services. A cloud service follows a pay-as-you-go approach, that means that customers are charged only for the time they use the service. Regarding the kind of services that are provided, a cloud may have the form of infrastructure services (IaaS), platform services (PaaS) and software services (SaaS). However, there is a lack of a solid foundation for quality of service in the clouds.

For instance, let us consider Dropbox, a widely-used cloud storage service where users can store files and get them from anywhere. In August 2012, users have frequently experienced unavailability of files and long synchronization delays between local files and data in some clouds. Such a bad performance has strongly affected the service reputation. Another example is the Amazon web service outage in April 2011 [1] which rendered inaccessible other services built on top of it, such as reddit, HootSuite or FourSquare. This last example also illustrates the strong relationship between the guaranties of the different layers of a cloud architecture: a lack of services at the IaaS level induces violations of the quality of service constraints at the SaaS level.

Formally, the definition of Quality of Service (QoS) is the ability of a service to meet certain requirements for different aspects of the service like performance, availability, reliability, or cost. In order to evaluate QoS in a qualitative and quantitative way, several metrics are considered like rejection rate, mean time

* Corresponding author.
E-mail addresses: Damian.Serran@imag.fr (D. Serrano), Sara.Bouchenak@imag.fr (S. Bouchenak), Yousri.Kouki@mines-nantes.fr (Y. Kouki), FredericoAlvares.deOliveiraJr@mines-nantes.fr (F.A. de Oliveira Jr.), Thomas.Ledoux@mines-nantes.fr (T. Ledoux), Jonathan.Lejeune@lip6.fr (J. Lejeune), Julien.Sopena@lip6.fr (J. Sopena), Luciana.Arantes@lip6.fr (L. Arantes), Pierre.Sens@lip6.fr (P. Sens).

http://dx.doi.org/10.1016/j.future.2015.03.018
0167-739X/© 2015 Elsevier B.V. All rights reserved.

between failures, response time, throughput, financial cost, or energy consumption. Then, service providers and customers have to negotiate a Service Level Agreement (SLA) that allows them to formally specify the QoS and agree on the requirements.

Any SLA mainly describes two things: the different Service Level Objectives (SLO) in terms of values for Quality of Service metrics and the penalties to be applied if the objectives have not been accomplished.

Existing public clouds providers offer very few guarantees in terms of performance and dependability [2]. This is the case of Amazon, Rackspace, or Microsoft for instance. Usually, they only commit to guarantee availability under the presence of hardware failures. For instance, we can get in Amazon's site the information that their service will be available 99.95% of time. However, other aspects like service response time, service network bandwidth, or even energy consumption are left to a best-effort' policy.

Contributions. We argue that the quality of service offered by cloud service providers and the respective SLA that they can commit are a differential key element among them. However, such a commitment raises the following challenges: (i) How to consider SLA in a general way for different cloud environments? (ii) How to describe the SLA terms between a cloud provider and a cloud customer, such as service levels objectives, or penalties in case of SLA violations? (iii) How to provide guarantees on cloud Quality of Service to produce better than best-effort behavior for clouds?

To address the above challenges, we propose a new method for providing SLA with quality of service in clouds. It is composed of the following components:

- a novel cloud model, *SLAaaS (SLA-aware Service)* which enables systematic and transparent integration of service levels and SLA into a cloud;
- a new language, called CSLA, to formally describe cloud-oriented Service level agreements;
- the definition of utility functions merging different *Service Level Objectives (SLO)*;
- online controlling algorithms to monitor and ensure the *SLOs*.

In order to illustrate the soundness and advantages of the proposed method, some SLA case studies (e.g. book store application, MapReduce service, locking service) at PaaS or SaaS levels are presented in the paper. A preliminary version of them can be found in [3]. Then, we propose in this paper a new scenario for energy shortage composed of SLAs at multi-levels which combines SLAs at both SaaS and IaaS layers.

The paper is organized as follows. Section 2 introduces some background and our general methodology. CSLA language is described in Section 3. Sections 4 and 5 detail our case studies. Section 6 describes our multi-SLA scenario. Section 7 reviews the related work, and Section 8 concludes the paper.

2. Design principles

A cloud provides a set of services where each of them exposes a functional interface with possible operations to be called in the context of the cloud. For instance, an IaaS cloud such as Amazon EC2 provides a functional interface that allows users to acquire compute instances, to run software on these instances, or to release instances while Amazon RDS PaaS cloud provides a relational database service that makes it easy to set up, operate, and scale a relational database. A third example is Google Apps SaaS cloud which provides a set of services with functional interfaces, such as Google Drive, that allows users to create, update, and share documents.

In addition to the above functional aspects of cloud services, there are also non-functional aspects related to the Quality of Service (QoS), such as *performance*, *availability*, *reliability*, and *cost*. For each QoS aspect, multiple QoS metrics may be considered. Some examples of such metrics are:

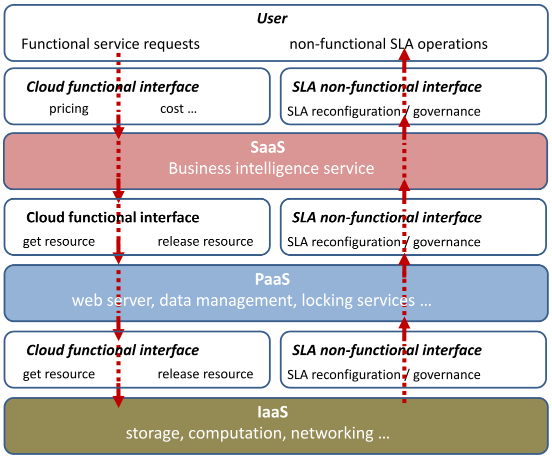

User
Functional service requests
non-functional SLA operations
Cloud functional interface
pricing
cost ...
SLA non-functional interface
SLA reconfiguration / governance
SaaS
Business intelligence service
Cloud functional interface
get resource
release resource
SLA non-functional interface
SLA reconfiguration / governance
PaaS
web server, data management, locking services ...
Cloud functional interface
get resource
release resource
SLA non-functional interface
SLA reconfiguration / governance
IaaS
storage, computation, networking ...

Fig. 1. SLAaaS cloud model.

- *performance metrics*: *response time*, which is the necessary time for a user request to get served, or *throughput* that reflects cloud service scalability, etc.
- *availability metrics*: *abandon rate*, which is the ratio of accepted service requests to the total number of requests, or *use rate* which is the ratio of time a cloud service is used to the total time.
- *reliability metrics*: *mean time between failures* which is the predicted elapsed time between inherent failures of the service, or *mean time to recover* which is the average time that a service takes to recover from a failure.
- *cost metrics* are the *energetic cost* that reflects the energy footprint of a service, or the *financial cost* of using a cloud service.

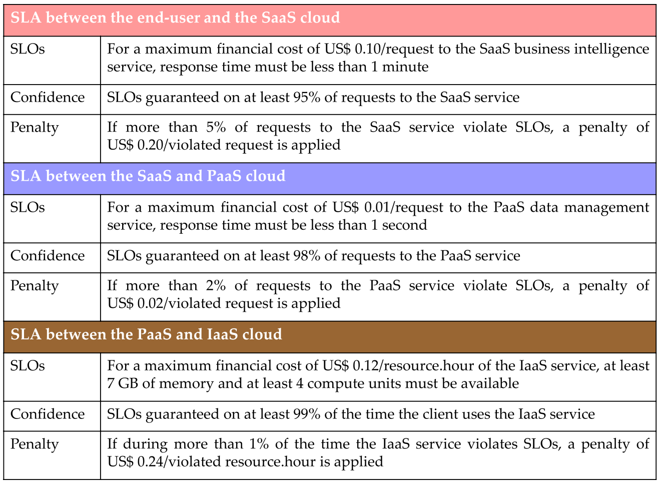
A QoS metric is, thus, a mean to quantify the service level with regard to a QoS aspect since the customer may require a service level to get a given objective, i.e., the *Service Level Objective (SLO)*. For instance, a SLO can define a QoS metric with a value higher/lower than a given threshold, maximize/minimize some QoS metrics, etc. Therefore, a *Service Level Agreement (SLA)* is a set of SLOs that should be satisfied and negotiated between the cloud service provider and the customer.

2.1. SLAaaS model

In order to allow the definition of non-functional interfaces which expose the SLA associated with cloud functional services, we have introduced a new cloud model denoted *SLA-aware-Service* (SLAaaS). Fig. 1 shows the SLAaaS model at three cloud levels: an IaaS cloud, a PaaS cloud and an example of a SaaS cloud that represents here a business intelligence system. We can observe in the figure four levels: an end-user is a client of the SaaS cloud, which is itself a client of the PaaS cloud, which is itself a client of the IaaS cloud.

Notice that the traditional functional interface of a cloud exposes operations that allow a cloud customer to get new resources from the cloud, access/use resources in the cloud or release resources that he/she does not use anymore while SLAaaS allows the cloud to expose SLA non-functional interfaces. Furthermore, SLAaaS aims to provide SLA-oriented cloud reconfiguration and SLA governance. In this article, we focus on the former.

By using SLAaaS, the user firstly selects the QoS aspects in which he/she is interested (e.g. performance, cost), as well as the QoS metrics for these aspects (e.g. service response time, financial cost). The user can then choose the SLOs he/she wants to apply on the QoS metrics. For instance, the SLO for the response time and for



SLA between the end-user and the SaaS cloud	
SLOs	For a maximum financial cost of US$ 0.10/request to the SaaS business intelligence service, response time must be less than 1 minute
Confidence	SLOs guaranteed on at least 95% of requests to the SaaS service
Penalty	If more than 5% of requests to the SaaS service violate SLOs, a penalty of US$ 0.20/violated request is applied
SLA between the SaaS and PaaS cloud	
SLOs	For a maximum financial cost of US$ 0.01/request to the PaaS data management service, response time must be less than 1 second
Confidence	SLOs guaranteed on at least 98% of requests to the PaaS service
Penalty	If more than 2% of requests to the PaaS service violate SLOs, a penalty of US$ 0.02/violated request is applied
SLA between the PaaS and IaaS cloud	
SLOs	For a maximum financial cost of US$ 0.12/resource.hour of the IaaS service, at least 7 GB of memory and at least 4 compute units must be available
Confidence	SLOs guaranteed on at least 99% of the time the client uses the IaaS service
Penalty	If during more than 1% of the time the IaaS service violates SLOs, a penalty of US$ 0.24/violated resource.hour is applied

Fig. 2. Examples of SLAs at different cloud levels.

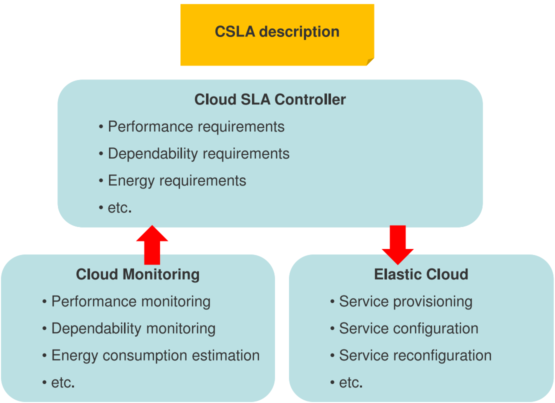


Fig. 3. Cloud autonomic reconfiguration.

the financial cost may be defined in order to respectively guarantee that the response time never exceeds a given threshold and the cost is minimized. Then, the SLA is defined as the combination of SLOs. Furthermore, the SLA between a cloud service and the customer may include additional information, such as the agreed confidence level (e.g. SLOs are guaranteed with a confidence of 95%), or the penalties applied in case of SLA violation. Fig. 2 presents three examples of SLAs that applied at three different cloud levels: between the end-user and the SaaS, between the SaaS and the PaaS, and between the PaaS and the IaaS.

2.2. Methodology overview

The SLAaaS model enriches the general paradigm of Cloud Computing, and enables systematic and transparent integration of service levels and SLA into the cloud. SLAaaS is orthogonal to IaaS, PaaS, and SaaS clouds and may apply to any of them. Furthermore, a specific language is introduced to describe QoS-oriented SLA associated with cloud services, the *CSLA* (*C*loud *S*ervice *L*evel *A*greement) language. CSLA is described in Section 3.

A control-theoretic approach is then described to provide performance, dependability and cost guarantees for online cloud services, with time-varying workloads. The online control of cloud services is based on a general feedback control loop as described in Fig. 3. To manage cloud SLA in a principled way, we follow a control-theoretic approach to design fully autonomic SLA-oriented cloud services. The general approach consists in three main steps.

First, a *utility/objective function* is defined to precisely describe the set of SLOs as specified in the cloud SLA, the weights assigned to these SLOs if any, and the possible trade-offs and priorities between the SLOs. The cloud service configuration (i.e., how many resources, what is their combination) with the highest utility is the best regarding SLA guarantees.

Then, *control theory techniques* are applied to model cloud service behavior, and propose control laws and algorithms for fully autonomic SLA-oriented cloud services. The challenges for modeling cloud services are to build accurate models that are able to capture the non-linear behavior of cloud services, and that are able to self-calibrate to render the variations of service workloads. The challenges for controlling cloud services are to propose accurate and efficient algorithms and control laws that calculate the best service configuration, and rapidly react to changes in cloud service usage.

3. CSLA language

CSLA, the Cloud Service Level Agreement language, allows to define SLA in any language for any cloud service (XaaS). CSLA addresses intrinsically the dynamic nature of the Cloud (e.g. elasticity) and its cost model. A preliminary version of CSLA has been introduced in our previous work [4]. This new version is more stable, addresses more features and is based on the Open Cloud Computing Interface (OCCI) [5] and the Cloud Computing Reference Architecture of the National Institute of Standards and Technology (NIST) [6].

3.1. Motivation and overview

Elasticity is the intrinsic element that differentiates Cloud computing from traditional computing paradigms, since it allows service providers to rapidly adjust resources to absorb the demand and hence guarantee a minimum level of Quality of Service (QoS) that respects the Service Level Agreements (SLAs) previously defined with their clients. However, due to technical and conceptual limitations (e.g., non-negligible resource initiation time, unpredictable workload), it becomes hard for service providers to guarantee QoS levels and SLA violations may occur. A Cloud SLA has to be suitable for heterogeneous, volatile resources in a highly unpredictable and dynamic environment. Existing SLA languages such as WSLA [7] and WS-Agreement [8] do not support the dynamic nature of the Cloud.

We propose CSLA (Cloud Service Level Agreement), a SLA language to finely express SLA contracts and to address SLA violations in the context of Cloud services. Besides the standard formal definition of contracts – comprising validity, parties, services definition and guarantees – CSLA is enriched with new properties (QoS/functionality degradation and an advanced penalty model) introducing a fine language support for Cloud elasticity management. Indeed, CSLA allows the expression of sophisticated Service Level Objectives (SLOs) with new features such as *confidence* and *fuzziness* to deal with QoS uncertainty: (i) the fuzziness defines the acceptable margin degree around the threshold of an expression; (ii) the confidence defines the percentage of compliance of clauses. Besides, the *functionality degradation* allows Cloud services to operate in different modes (e.g., 2D vs. 3D display, a degree of security levels), each one consuming more or less resources, consequently this property allows service providers more flexibility to raise additional resources. Finally, an advanced penalty model related to degradation is proposed. This model aligns penalties with functionality/QoS degradation in order to provide a good trade-off between price and quality which is both attractive for final clients and profitable for Cloud service providers.

Our goal is to make contracts more flexible and consequently increase Cloud services self-adaptation capability and elasticity possibilities. CSLA allows service providers to maintain its consumers satisfaction while minimizing the service costs due to resources fees.

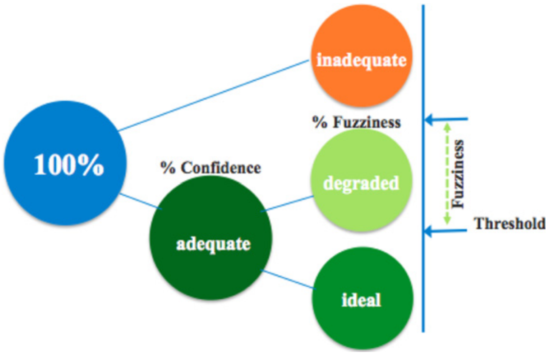


Fig. 4. SLO evaluation in CSLA.

3.2. How to evaluate service level objectives (SLO) in CSLA?

A SLO is a predicate which has usually one of the following form: a QoS metric (e.g., response time) with a value higher/lower than a given threshold (e.g., 3 ms). In CSLA, we enrich the SLO definition with the fuzziness and the confidence features (see 3.4 for examples). In order to evaluate an objective (SLO), an initial evaluation enables to classify the predicate as *ideal* (i.e., threshold is respected), *degraded* (i.e., threshold is respected using fuzziness margin) or *inadequate* (i.e., threshold is not respected even with fuzziness margin) (cf. Fig. 4). We distinguish two types of evaluation: (i) per-interval evaluation, in which the evaluation is performed at the end of each interval (e.g. time window of 30 min); (ii) per-request evaluation, in which the objective is evaluated for each request. At the end of the time window, a final evaluation allows one to verify an objective (SLO) by applying the fuzziness and confidence percentages to the initial evaluation. Moreover, the final evaluation enables the identification of non-accepted/accepted degradation and inadequate cases. In other words, the final evaluation absorbs or notifies the violations.

3.3. CSLA meta-model

A SLA in the CSLA language contains three sections: a section describing the validity, a section defining the parties involved and the section referencing the template used to create the agreement (cf. Fig. 5). The *Validity* defines how long an agreement is valid. CSLA distinguishes two types of *Parties*: *Signatory* parties, namely service provider and service customer, and *Supporting* parties (e.g., trusted third party).

A CSLA *Template* is like a pattern for SLA. It contains five elements: cloud services definition, Parameters, Guarantees, Billing and Terminations.

- A cloud service definition refers any XaaS service (SaaS, PaaS or IaaS). We use OCCI standard for IaaS services definition.
- Parameters provide a way to define variables in the context of the agreement which should be used in others sections. Variables refer to a distinct element such as *Metric*, *Monitoring* and *Schedule* (see 3.4).
- Guarantees contain four elements: *Scope*, *Requirements*, *Terms* and *Penalties*. The scope specifies which services in the agreement are covered by the guarantee. The requirements define the specifications that must be fulfilled for operating the scope services (e.g., Flash Player v10.1 or above). The terms aggregate guarantees term with *and* or *or* operators. A guarantee term contains one or more *Objectives* (SLO). Each objective defines an expression that must be met according to a precondition. An expression formulates a predicate. It is characterized by a *Metric*, a *Comparator* and a *Threshold*. We define a *Priority* for each objective to take into account the customer QoS preferences. The metric is evaluated according to predefined *Monitoring* in specific period (*Schedule*). Penalties compensate the consumer for accepting QoS or functionality degradation and tolerating the SLA violation. The compensation can be applied either as a constant or variable rate (see 3.4).
- CSLA supports two types of billing: *Pay as You Go* (i.e., price per request on the cloud service) and *All-in package* (i.e., fixed price per period).
- Finally, the agreement continues in force in accordance with the section *Validity* or in accordance with the *Terminations* section which can describe a specific cancellation clause.

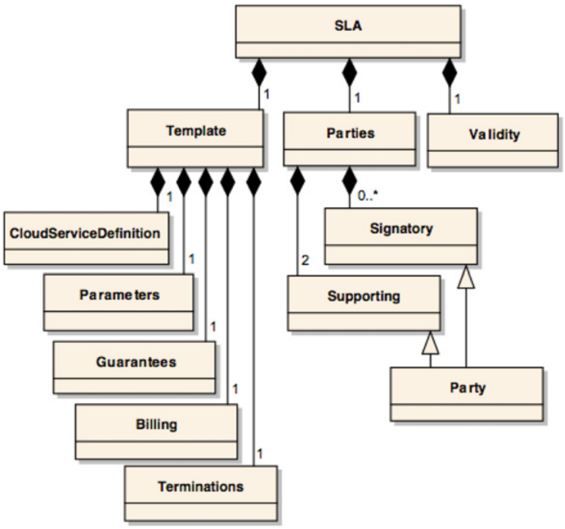


Fig. 5. CSLA meta model.

3.4. CSLA example

The CSLA syntax is defined according to the grammar generated from the CSLA meta-model. In this paper, we use XML as a representation format. The following XML presents an example of a CSLA file describing the guarantee *Terms* and *Penalties* for SLA between a SaaS provider and its customer concerning the service S1 (for more details see Section 6).

In this example, two SLOs are composed using the "*and*" operator: a performance SLO – contractualizing the QoS of the response time – and a mode SLO—contractualizing the use of the functionality degradation. The performance SLO (lines 6–11) specifies that for each interval of 3 min in window of 30 min (expressed in the variable *Mon-1*, not detailed here), the maximum request response time (*Rt*) must be below 3 s if the data size is less than 1 TB. This objective should be achieved every day between 8 a.m. and 10 a.m. (expressed in the variable *Sch-Morning*, not detailed here) during the validity of the contract. It guarantees that, on at least 99% of requests for the service S1 (*Confidence*) among which 10% can be degraded (*Fuzziness*), i.e., a margin of 0.2 s is acceptable as a QoS degradation. Lines 12–14 specify the mode SLO. The functionality degradation mode must be used in 10% of requests for the service S1. It is noticeable that the functionality degradation is managed like any other SLOs since it defines an objective of usage.

The second part presents the penalties (lines 16–31). They are applied in case of SLA violations to compensate cloud service customers, i.e., penalties reduce the service price. The reduction can be applied either as a constant or variable rate. In the latter case, the request price is modeled as linear function [9]. A violation of the mode SLO (lines 25–30) implies a penalty equal to 0.1 euro/request whereas the penalty of the performance SLO (lines

17–24) depends on delay. In the request, price is modeled as: $P = \alpha - \beta \cdot dt$; where α is the price with no violations ($\alpha > 0$), β is the penalty rate ($\beta > 0$) and dt is the absolute difference between the actual value and the SLO threshold. For each penalty, a procedure indicates the actor in charge of the violation notification (e.g., provider), the notification method (e.g., email) and the notification period (e.g., 7 days).

```
1    <csla:terms>
2            <csla:term id="T1" operator="and">
3            <csla:item id="responseTimeTerm"/>
4            <csla:item id="modeTerm"/>
5    </csla:term>
6    <csla:objective id="performanceSLO" priority="1" actor="provider">
7            <csla:precondition policy="Required">
8                    <csla:description> Data size less than 1 TB</csla:description>
9            </csla:precondition>
10           <csla:expression metric="Rt" comparator="lt" threshold="3"
11              unit="second" monitoring="Mon−1" schedule="Sch−Morning"
12              Confidence="99" fuzziness−value="0.2" fuzziness−percentage="10"/>
13   </csla:objective>
14   <csla:objective id="modeSLO" priority="2" actor="provider">
15           <csla:expression metric="Mu(S1−M2)" comparator="lt" threshold="10"
                    unit="\%" monitoring="Mon−1" Confidence="99"
16                  fuzziness−value="2" fuzziness−percentage="5"/>
17   </csla:objective>
18   </csla:terms>
19   <csla:penalties>
20   <csla:Penalty id="p−Rt" objective="responseTimeTerm" condition="violation"
             obligation="provider">
21           <csla:Function ratio="0.5" variable="delais" unit="second">
22                   <csla:Description> ... </csla:Description>
23           </csla:Function>
24           <csla:Procedure actor="provider" notificationMethod="e−mail"
                    notificationPeriod="7 days">
25                   <csla:violationDescription/>
26           </csla:Procedure>
27   </csla:Penalty>
28   <csla:Penalty id="p−Mu" objective="modeTerm" condition="violation" obligation
             ="provider">
29           <csla:Constant value="0.1" unit="euro/request"/>
30           <csla:Procedure actor="provider" notificationMethod="e−mail"
                    notificationPeriod="7 days">
31                   <csla:violationDescription/>
32           </csla:Procedure>
33   </csla:Penalty>
34   </csla:penalties>
```

4. SLA for a SaaS service: bookstore application

To illustrate the design principles, we describe in the following how we applied the proposed SLAaaS model to the TPC-W [10] online bookstore Software-as-a-Service.

4.1. SLA actors

TPC-W [10] is a well-known benchmark that emulates a bookstore which can be offered as SaaS. TPC-W is organized in two tiers, namely the front-end web tier and the back-end database tier; each tier may have one or more servers. Multi-tier architectures are intended to improve scalability, since the larger the set of servers in each tier is, the better the performance and availability. However, the number of servers involved in a cloud service determines its cost. There is a trade-off between performance, availability and cost, which is not straightforward to handle.

This kind of trade-offs can be easily controlled with a SLAaaS service. Table 1 presents an example of SLA for such a service. Here, the SLA is established between the SaaS bookstore provider and any of its customers. The table shows three SLOs: response time that should not exceed 500 ms with 150 ms of fuzziness, availability with at least 95% of requests have to be successfully processed, and cost, i.e., number of servers, that should be kept at a minimum given that performance and availability objectives are guaranteed.

Table 1
SLA for multi-tier bookstore SaaS in CSLA language.

Service	Metric	Oper.	Value (ms)	Fuzz. (ms)	Conf. (%) ($)
Multi-tier Bookstore	Response time	$\leq$	500	150	100
	Availability	$\geq$	95	0	100
	Cost (#nodes)	min	–	–	–

4.2. Objective function

The first step to build our SLA-aware service is to translate the desired SLA into an objective function (recall Section 2.2). To do that, we first draw a function to capture that performance and availability objectives (PAO) are guaranteed at a given time.

$$PAO(t) = PO(t) \cdot AO(t) \tag{1}$$

$$PO(t) = \begin{cases} 1 & \text{if } \ell(t) \leq \ell_{\max} + f_{perf}^{\Delta} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$AO(t) = \begin{cases} 1 & \text{if } \alpha(t) \geq \alpha_{\min} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $\ell(t)$ is the average response time at time t, $\ell_{\max}$ is the maximum response time defined in the SLA, f_{perf}^{Δ} is the fuzziness value described in the SLA, $\alpha(t)$ the availability (ratio of successfully processed requests) at time t, and $\alpha_{\min}$ is the minimum availability defined in the SLA. Thus, $PAO(t) = 0$ means objectives are not met at time t, and $PAO(t) = 1$ when they are.

We have to relate now the third objective with the other two. The third objective is to minimize the cost of the service, which, in a cloud context, is directly related with the usage of the cloud infrastructure. For simplification purposes, we consider only homogeneous nodes/virtual machines. Other cost parameters like network usage or storage could be added similarly. We can, thus, draw the following objective function:

$$\theta(t) = \frac{T \cdot PAO(t)}{\omega(t)} \tag{4}$$

where $\omega(t)$ gives us the number of nodes at time t hosting both tiers of the SaaS, T being the number of tiers ($T = 2$ in this use case) which is used for normalization purposes. Note that $\forall t,\ \theta(t) \in [0, 1]$, since $\omega(t) \geq T$, that is, we consider a minimum of one node in each tier at any moment, and, as said, $PAO(t) \in \{0, 1\}$.

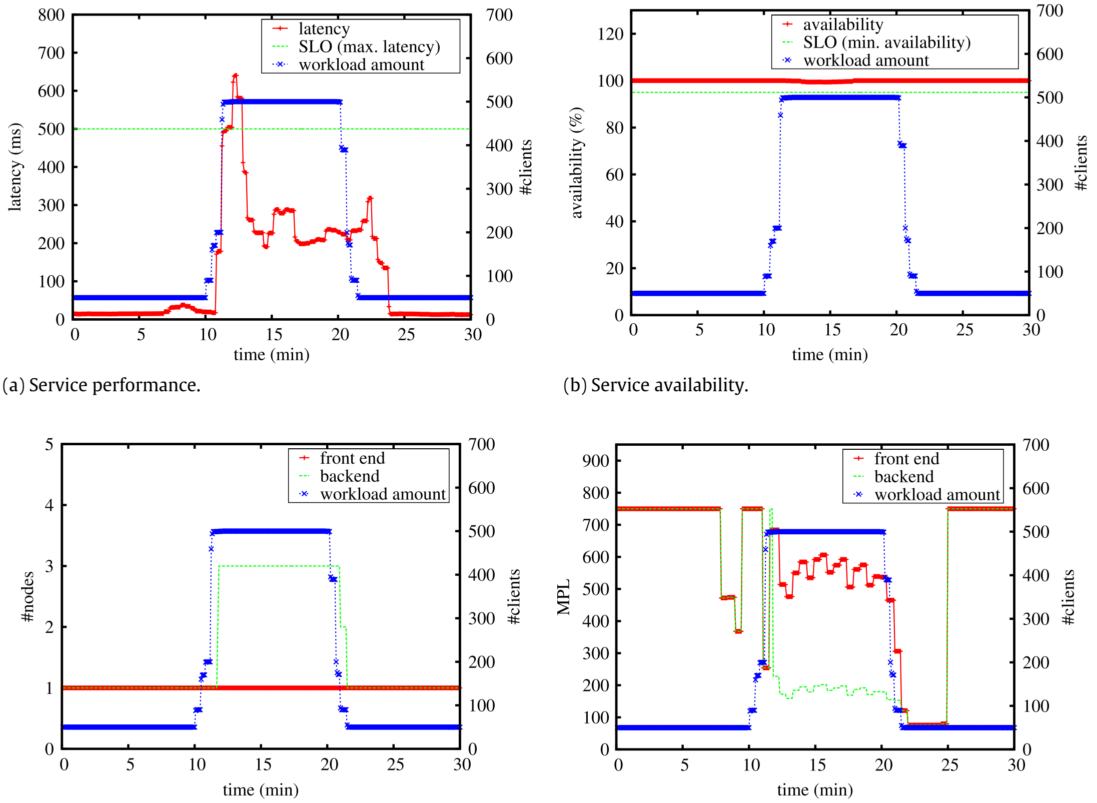
Eq. (4) constitutes the objective function that integrates the requirements described in the SLA. This objective function will be maximized by the control algorithm.

4.3. Control algorithm

The environment where an SaaS is used is not static. Different amounts of customers can access the service at different moments, competing for computing resources. Finding the configuration of the service that provides the highest utility in a constantly changing cloud environment constitutes the main challenge when building an SLAaaS. That configuration is maintained and adapted by the control algorithm.

To control our SaaS, first we need to model its behavior and then to find the configuration of the service that uses the model to maximize the utility function.

The considered SaaS service follows a multi-tier architecture. More specifically, there is a queue of received requests waiting to be executed in the first tier and the execution of the requests in the first tier generates more requests that are placed in a queue for the second tier. This execution behavior accommodates to a queuing network approach where the execution queue of each tier is modeled as a M/M/c/K queue [11,12]. M/M/c/K means that



(a) Service performance.

(b) Service availability.

(c) Online control of cloud instances.

(d) Online control of MPL.

Fig. 6. SLAaaS multi-tier bookstore service.

request arrival time is independent of other requests (first M), execution time is independent of other requests (second M), c is the number of servers (called $\omega(t)$ in the utility function), and K is the number of requests accepted for execution (MPL). Both c and K correspond to the configuration of the service. Besides c and K, each tier queue needs the average response time and the average arrival time to estimate the execution time. The estimate for the availability is calculated as the ratio of received requests in each tier and the MPL.

In the M/M/c/K model, the parameters c and K can be varied to find the configuration that maximizes the utility function (Eq. (4)). This is the task of the capacity planning algorithm. For that, the capacity planning algorithm uses the M/M/c/K network with the monitored values for average response time and average arrival time and implements a dichotomic search on the other two parameters (c and K). The search has maximum values fixed for each parameter. Once the search finds the values of c and K with the highest utility, if they differ from the current configuration, actuators are triggered to add or remove server nodes and to modify the MPL.

We assume that between two consecutive executions of the capacity planning algorithm the service is able to stabilize, specially new resources are warming up. This assumption is suitable as the stabilization time is short compared to the duration of the experiment, as will be shown in graphs in Fig. 6. [13] describes a model and capacity planning algorithm deeper in detail for a multi-tier service. The most interesting particularity is that the capacity planning algorithm is able to configure the service at once for a given SLA, without the need to follow a step-by-step approach. More complex capacity planning algorithms could be implemented, for instance to cope with the added resources while they are not yet stabilized. However, that is out of the scope of the paper.

The suitability of a queuing system for online services has been already studied in [13]. Finding other models for the same service dealing with other situations is out of the scope of this paper. More enriched models could be proposed instead of using queuing theory, they might take into account the impact of VM migrations, for instance.

4.4. Experimental results

We have implemented our control algorithm in an existent implementation of the benchmark [10]. The results of our experiments can be found in Fig. 6. These experiments consider the SLA depicted in Table 1.

To emulate dynamicity in the usage of the cloud service, we have varied the number of clients during the execution of the experiment, starting at 50, then sharply increased until 500 and finally coming back to 50 again. Clients submit read-only requests belonging to the browsing-mix, which is a workload provided in the specification of TPC-W and integrated in the implementation we used.

To cope with the execution of the capacity planning algorithm, the parameters needed by the model are monitored every 5 s. These parameters include the request arrival rate and the average request response time. The capacity planning algorithm is executed every minute and uses the history of the monitor data collected in the 2 previous minutes. Maximum values for the number of nodes and MPL are fixed to 25 and 900 respectively. Row G5K I in Table 2 describes the hardware configuration. As initial configuration, each tier is composed of only one server, plus one extra server that runs the capacity planned separately.

Figs. 6(a) and (b) show the values overtime for the performance and availability objectives defined in the SLA, as well as the monitored number of concurrent clients accessing the service. Below, Figs. 6(c) and (d) show the controlled configuration, the number of nodes and the maximum number of requests executed

Table 2
Hardware configurations.

Cluster	CPU	Memory	Storage	Network
Amazon EC2	Large instances, 4 EC2 Compute Units in 2 virtual cores	7.5 GB	850 MB	10 Gbit Ethernet
G5K I	4-core 2-CPU 2.5 GHz Intel Xeon E5420 QC	8 GB	136 GB SATA	1 Gbit Ethernet
G5K II	4-core 1-CPU 2.53 GHz Intel Xeon X3440	16 GB	278 GB SATA II	Infiniband 20G

Table 3
SLA for MapReduce PaaS in CSLA language.

Service	Metric	Oper.	Value (s)	Fuzz. (s)	Conf. (%)
MapReduce	Response time	≤	90	5	100
	Cost	min	–	–	–

concurrently (MPL), respectively. Results shown in the graphs correspond to measurements after 15 min of warmup with 50 clients.

At the beginning, each tier is composed of only one server, offering satisfying values for performance and availability regarding the considered SLA. Then, our SLAaaS bookstore reacts adding 2 new servers to the database tier (Fig. 6(c)) and setting new values for the MPL (Fig. 6(d)) to cope with the sharp increment on the number of concurrent clients at minute 12. That decision avoids the violation of the response time SLO (Fig. 6(a)), which although we can see that the response time is higher than the maximum value of the response time SLO, that values fall into the acceptable margins due to the fuzziness property of CSLA. Moreover, we can see that impact on availability is almost unnoticeable (Fig. 6(b)).

5. SLA for PaaS services

As stated earlier in Section 2.1, our SLAaaS model can be applied to any service in the three XaaS layers. We successfully applied it to an SaaS service in the previous section. We illustrate here how to use our proposed approach to build SLAaaS services at the platform layer (PaaS).

The experiments presented in this section were conducted a private cloud, Grid'5000 [14], and a public cloud, Amazon EC2. Table 2 shows the hardware configurations.

5.1. Map-Reduce service

The first PaaS service built using our SLAaaS model is a MapReduce service. Following directions in Section 2.2 we describe how to build this service and we validate it experimentally afterwards. Some previous works [15,16] have taken similar approaches, however, they do not provide a MapReduce framework that guarantees a SLA regarding the intrinsic dynamicity of cloud services in terms of variation in the workload amount.

5.1.1. SLA actors

MapReduce [17] has become a widely extended programming model and execution environment for Big Data processing (i.e., large amounts of unstructured data). It can be run on clusters of commodity computers attaining high availability with an acceptable high performance. To that end, MapReduce provides automatic mechanisms to parallelize execution and to partition and replicate data across the cluster. Several cloud providers (for instance, Amazon or Azure) offer MapReduce as Platform-as-a-Service through a functional interface with operations to start/stop a MapReduce cluster or submit a job for execution.

Following our SLAaaS model, a SLA could be established between a MapReduce PaaS provider and its customers. We take in this section the SLA presented in Table 3. In that SLA, response time should be below 90 s (performance objective) with the minimum cost in terms of the size of the MapReduce cluster (cost objective).

5.1.2. Objective function

To apply SLAaaS to a MapReduce service, first a utility function drawn ad hoc from the SLA is defined. Similarly to the SaaS case, we combine an expression for performance objective with another integrating the cost. Eq. (5) tests if the service guarantees the performance SLO at time t:

$$PO(t) = \begin{cases} 1 & \text{if } \ell(t) \le \ell_{\max} + f_{perf}^{\Delta} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where $\ell(t)$ is the average response time at time t, and $\ell_{\max}$ is the threshold in the performance SLO and f_{perf}^{Δ} its fuzziness. Thus, $\forall t,\ PO(t) \in \{0, 1\}$ whether performance objective is guaranteed or not. Then, we combine $PO(t)$ with cost:

$$\theta(t) = \frac{PO(t)}{\omega(t)} \tag{6}$$

where $\omega(t)$ is the size of the MapReduce cluster at time t in terms of the number of servers.

Eq. (6) is the utility function that we need to maximize in this case. Obviously, the minimum number of servers that guarantees response time provides the highest utility. Note that, $\forall t,\ \theta(t) \in [0, 1]$.

5.1.3. Control algorithm

The control of the considered MapReduce service takes place into two steps. First, a model for the service is defined to estimate $\ell(t)$, the average response time at a given moment. Then, a capacity planning algorithm to find the configuration of the service that guarantees the SLA is provided. That algorithm uses the model to maximize the previously defined utility function ($\theta(t)$).

MapReduce divides each job into smaller work units called tasks and place those tasks into a queue to be executed by any of the server nodes. Similarly to the SLA-aware SaaS service presented in Section 4, the queue can be modeled as a M/M/c queue [11,12]. That is, where the arrival of jobs is independent of other jobs (first M), the execution time of jobs is independent of other jobs (second M) and c is the number of server nodes (called $\omega(t)$ in the utility function). Although there are two different types of tasks, we use only one queue assuming there will be always enough tasks to feed the server nodes. Finding a finer grain model could be possible but we consider it to be out of the scope of the paper.

The model uses as inputs the average job arrival rate, the average job response time and the current size of the MapReduce cluster, the model is able to predict the average response time for future jobs given a different configuration (i.e., changing the number of nodes).

The capacity planning algorithm uses the previous defined model to search for the minimum cluster size that maximizes the utility function $\theta(t)$. For that it uses monitored values for the average job arrival rate and the average job response time and implements a binary search on the parameter c. If the result varies from the current configuration, the number of server nodes is adjusted triggering actuators to add/remove servers. As for the SaaS control algorithm, here again the number of servers guaranteeing the SLA is calculated at once without the need of following a step-by-step approach. Again, we assume that the system stabilizes between two consecutive executions of the

Table 4
SLA for locking PaaS.

Service	Metric	Unit	Oper.	Value	Fuzz.
Locking	Response time	ms	$\leq$	400	0
	Resource usage	% of time	max	–	–

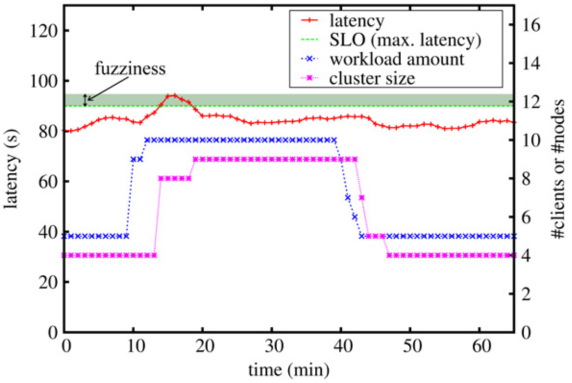


Fig. 7. Self-elastic MapReduce service.

capacity planning algorithm as the time to stabilize is short compared to the experiment duration (see Fig. 7).

Our M/M/c queue is suitable since MapReduce is proposed as an online service where users submit requests concurrently [13]. Like the SLA-aware SaaS, building more complex capacity planning algorithms, for instance being able to cope with resources that are not yet warmed up, or models that can predict VM migrations are out of the scope of this paper. Note that, we have considered only homogeneous servers in our use case. The SLA-aware MapReduce service could be built taking into consideration different node types. For that, we need to change the model and the utility function. Instead of a M/M/c queue model, we need to build the queuing system with a different service time distribution associated to each node. On the other hand, the utility function should combine the cost of the different nodes instead of only counting them. The controller could be enriched with heuristics to deal with nodes that are not working properly. The controller could blacklist and report those nodes or replace those nodes by "fresh" instances in a virtualized scenario.

5.1.4. Experimental results

We have implemented our control algorithm in Hadoop [18], a very popular implementation of MapReduce and we set up a MapReduce PaaS service in a cluster of Amazon EC2 instances (see Table 2).

We have generated a dynamic workload to test the SLAaaS MapReduce using MRBS [19] emulating MapReduce clients in another instance. One additional instance is used to host the SLA controller. Among the five different benchmarks provided, we used Recommendation System which is proposed to evaluate online services that use MapReduce. Recommendation Systems emulates an online movie recommendation site with real data from MovieLens [20] (the experiments use the dataset consisting of 1700 movies, 1000 users, and 100,000 ratings given by users for movies indicating how much users like or dislike them). As other recommendation sites, the request that can be executed are requesting the top ten recommendations for a user, listing all the ratings for a specific movie, list all the ratings given by a certain user or to propose how much a user would like or dislike a certain movie, among others. Clients take a random request and submit the jobs for execution to the SLAaaS MapReduce clustering a closed loop.

Fig. 7 shows the results of our experiment. The graph depicts the measurements after warming-up the service with 5 clients during 10 min and the number of concurrent clients (workload amount) changes over time from 5 to 10 and back to 5. The initial size of MapReduce cluster is set to 4 servers, plus 1 node that runs MRBS, plus 1 node that runs the SLA controller. We have added also monitoring for the parameters need by model: job arrival rate and average response time, monitored with a 1-min time window. In this experiment, the capacity planning algorithm is executed every 3 min and uses the data collected in the previous 5 min.

Interestingly, we observe that the capacity planning algorithm reacts when the workload amount increases until 10. At that moment we can see that client request response time also increases above the SLO for response time at time 13 min. Nevertheless, this behavior does not impose any penalty. The response time is kept under acceptable margins as indicated by the fuzziness property in the SLA (see Table 3). Nevertheless, the capacity planning algorithm detects that this configuration for MapReduce would lead to violations of the SLA. That is why the number of servers is increased. When the number of clients goes back to 5 again, the capacity planning algorithm releases the servers that are no more needed to guarantee the performance SLO. This decision is taken to keep the service cost at a minimum.

Through this use case, we have successfully applied our proposed SLAaaS model to a MapReduce PaaS service enabling it to accomplish a SLA established between provider and its clients which also involves performance and cost objectives.

5.2. Locking service

Locking services ensure exclusive access to shared resources by concurrent processes, and is usually provided as a Platform-as-a-Service in a cloud. For instance, Google provides the Chubby distributed locking mechanism that is used by other cloud services such as Google File System service and BigTable data storage service [21]. Basically, such a mechanism provides a functional interface with operations to acquire or release locks. However, locking procedures remain costly. According to [22], locking is considered as an important and poorly resolved problem in cloud. Its protocols have to be scalable and take into account QoS objectives.

5.2.1. SLA actors

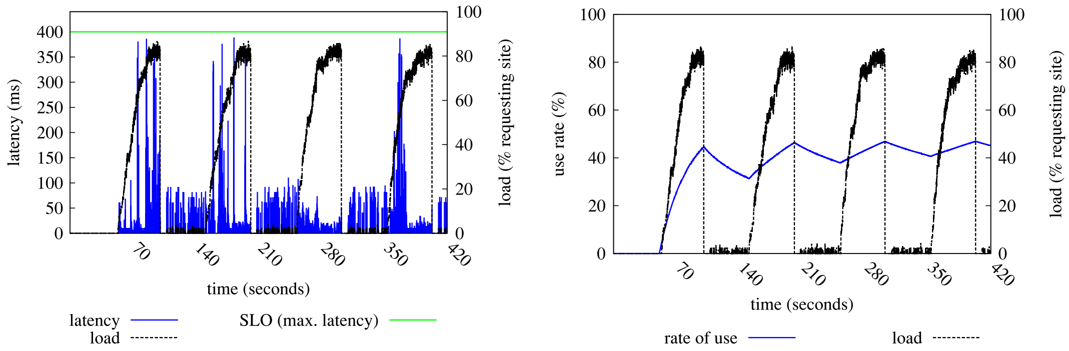
The SLAaaS model can be applied to a locking service at PaaS level. In this case, the SLA is engaged between the locking service and the respective customers. Table 4 gives an example of SLA that combines performance and availability objectives. The SLA specifies that the response time of a request to the lock service should not exceed 400 ms.

5.2.2. Objective function

In order to minimize the response time of a lock service, the use rate of the locked shared resource should be held as high as possible. This is expressed ad hoc into a utility function:

$$\theta(t) = \frac{PO(t)}{\rho(t)} \tag{7}$$

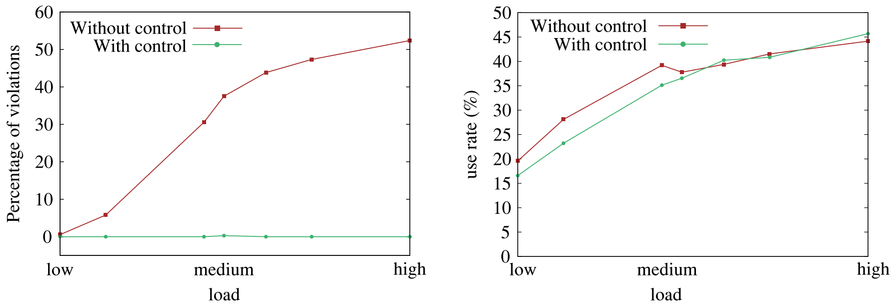
where $PO(t)$ is given in Eq. (5), and $\rho(t)$ is the use rate of the locked resource. Intuitively, the locking service with the highest utility value is the one that guarantees the SLO (if possible) with a high resource use rate, and therefore, the SLA is guaranteed.



(a) Service performance.

(b) Service availability.

Fig. 8. Self-adaptive locking service.



(a) Service performance.

(b) Service availability.

Fig. 9. Impact of control admission in locking service.

5.2.3. *Control algorithm*

For providing the above SLA guarantees, we propose a locking service that combines admission control techniques with a distributed locking algorithm [23]. Thus, before accepting a request, the locking service controller first verifies that, taking into account the current system state, the performance SLO can be satisfied. If it is the case, the request for lock acquisition is accepted and will be satisfied; otherwise, the request is rejected. The complete algorithm can be found in [23].

5.2.4. *Experimental results*

We conducted experiments with the proposed SLAaaS-oriented locking service approach, on top of a 40 node cluster in the G5K II infrastructure (see Table 2). To emulate long distance, we injected network latency between nodes. Each node runs a process that may request the locking service related to a shared resource. The load of requests of the system varies over the time. It is characterized by the ratio of processes requesting lock acquisition to the total number of processes, as shown in Fig. 8.

Fig. 8(a) shows the mean response time of lock requests (latency) over the execution of the experiment when the load varies. When the load is low, the response time remains low compared to the SLO. When the load increases, there is more contention on the shared resource, with an increase of lock request latency. However, the locking service automatically adapts itself in order to keep request latency below the threshold, as specified by the SLA. Such an adaptation is possible thanks to the admission control.

Fig. 8(b) shows the use rate of the shared resource, i.e., how often the resource is actually locked and used by one of the processes. It is expressed by the ratio of time during which the resource is used by processes to the total time. In the network configuration testbed, such a ratio cannot exceed 50% since half of the total time is spent in message transmission. Interestingly, when the load increases the locking service adapts to the load, with an increasing use rate until a maximum value, which corresponds to the availability objective of the underlying SLA. In summary, SLAaaS successfully applies to associate SLA with a locking service at PaaS level.

We have evaluated the impact of our admission control mechanism and the results are summarized in Fig. 9. To this end, we considered two versions of our algorithm, named *Without control* and *With control*, which respectively disables and enables the control mechanism. Contrarily to the experiment of Fig. 8, each point of the figures corresponds to an experiment computed with a static given load. Fig. 9(a) shows the number of violated requests, i.e., the number of requests which have been satisfied after their required deadline while Fig. 9(b) shows the use rate of the shared resource.

On the one hand, we note that, in terms of use rate, both versions have the same behavior: in low load, the resource is slightly less available in the version with the control mechanism, whereas they behave more or less the same way with medium and high load. On the other hand, Fig. 9(a) shows that there is no violation with the control mechanism whatever the load. Consequently, we can deduce that the admission control mechanism avoids violation of requests without degrading service availability.

6. Cross-layer SLAs: energy shortage scenario

Energy consumption of datacenters has been seen as a big issue [24]. The way energy management is performed at the IaaS level may drastically impact the other services that depends on it. For instance, IaaS may be lead to shutdown part of its physical infrastructure in order to cope with periods of energy shortage. As a result, the SLAs established with IaaS consumers (e.g. PaaS/SaaS providers) hosted on it may no longer be guaranteed. That has a domino effect since it may also impact the upper levels SLAs established between the PaaS/SaaS provider and the its consumer and hence force them to seamlessly adapt itself to avoid violations.

Table 5
SLA between the IaaS and SaaS providers.

Service	Metric	Oper.	Value (%)	Price ($)	Fuzz. (%)	% of Fuzz.	Conf. (%)	Penalty ($)
Small/Large	Availability	$\geq$	98%	0.06/0.12	18	10	100	0.05/CPU core

Table 6
SLAs between the SaaS provider and Ad Clients 1 and 2.

Service	Price ($- CPM)	Usage mode	Funct. Deg. Penalty ($)	Metric	Oper.	Value (ms)	Fuzz. (ms)	% of Fuzz.	Conf. (%)	Penalty ($)
Ad 1	0.30	Video	–	Resp. time	$\leq$	500	300	20	90	0.10
Ad 2		Video (80%) Image (20%)	0.05 per exceeding %							

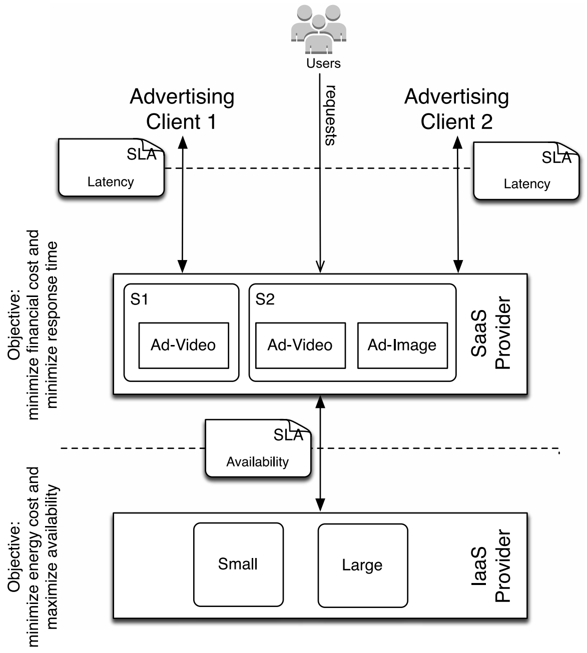


Fig. 10. Actors of the multi-SLA scenario.

The objective of this section is to show how SLAaaS can be employed to establish SLAs at different levels and how those multi-level SLAs can guide autonomous behaviors in several layers of the cloud stack.

6.1. SLA actors

Fig. 10 illustrates the architecture of this case study. We consider a two-level cloud system, in which, at the lower level an IaaS provides physical infrastructure as a service by means of virtual machines, whereas at the upper level, a SaaS provides an advertisement software as a service. So, the SaaS provider is a IaaS client and companies willing to deploy advertising campaigns are the clients of the SaaS provider. Finally, end-users are regular website visitors who implicitly requests advertisements through a web browser.

Like other commercial cloud infrastructure providers (e.g. Amazon EC2 or Microsoft Azure), IaaS consumers are given two service options with respect to the amount of compute virtual resources (CPU and RAM), namely large and small. A SLA is established between the IaaS provider and its client (in this case the SaaS provider) stating that the provider must guarantee at least 98%, as it is shown in Table 5.

The availability can be defined as the proportion of service uptime, that is, the measure of likelihood to successfully access resources. It is calculated by the proportion of time the allocated resources (VMs) can be accessed within an observation period.

According to the SLA there is also a fuzziness of 18% and fuzziness percentage of 10%, which means that the availability may be between 98% and 80% in 10% of the observation periods. In case of violation, a financial compensation of 0.05$ per CPU core allocated should be given to the client.

The SaaS provides only one service (advertisement), which may operate in two modes: normal (video) and degraded (static image). Clients are charged in a cost-per-mille (CPM) manner, that is, according to the number of times per thousand (mille) each advertisement is viewed. As shown in Table 6, a SLA is established between the SaaS provider and each one of its clients (companies interested in internet advertisement). More precisely, for Ad Client 1, only video views is accepted, whereas for Ad Client 2, up to 20% of image views may be accepted. Beyond that threshold, a functional degradation penalty of 0.05$ per exceeded percentage point must be payed back to the client as compensation.

For both SLAs, the SaaS provider should guarantee an average response time less than or equal to 500 ms, with confidence, fuzziness and percentage fuzziness of 90%, 300 ms and 20%, respectively. It means that the average response time measured within an observation period may exceed 500 ms in at most 10% of the observation periods within a predefined time window and may be between 500 and 800 ms in at most 18% (90% of 20%) of the measured values within a predefined time window.

6.2. Objective functions

Based on the SLA, we define a utility function (cf. Eq. (8)) that takes into account the number of physical nodes and the availability.

$$(\text{IaaS})AO(t) = \begin{cases} 1 & \text{if } \alpha(t) \geq \alpha_{\min} \\ f_{av} & \text{if } (\alpha_{\min} - f^{\Delta}_{av}) \leq \alpha(t) < \alpha_{\min} \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

where $\alpha(t)$ is the service availability, $\alpha_{\min}$ is the minimum service availability to be guaranteed and f^{Δ}_{av} is the fuzziness. $f_{av} \in \{0, 1\}$ is the fuzziness function, which takes into consideration the percentage of fuzziness and the confidence to return either 0, if the IaaS violates the SLA, or 1 otherwise.

The boolean expression in Eq. (9) states that the current energy consumption should not exceed a given threshold due to energy shortage reasons.

$$(\text{IaaS})EO(t) = (\epsilon(t) \leq \epsilon_{\max}) \quad (9)$$

where $\epsilon(t)$ corresponds to the power consumption at time t, and $\epsilon_{\max}$ to the maximum power consumption the data center must have due to energy shortage reasons.

The decision module of the IaaS is modeled with Constraint Programming (CP), in which the problem is stated by means of decision variables, variable domains and constraints on these variables. Given an energy threshold to be used to cope with the energy shortage and the shortage duration, the model determines which nodes should be shutdown so that the SLA violations are minimized, that is, the solution with highest utility (cf. Eq. (10)). In other words it tries to choose the set of nodes in way the impact on the already allocated resources (to the SaaS) is minimized.

$$(\text{IaaS})\theta(t) = (AO(t) \cdot EO(t)). \tag{10}$$

In order to avoid SLA violation, the objective of SaaS provider is to have the minimum amount of resources necessary to maintain the response time.

As formalized in Eq. (12) the SaaS utility function takes into consideration the expression associated to the response time (cf. Eq. (11)), the service usage mode (e.g. image or video advertisement) and the number of instances allocated to the service.

$$(\text{SaaS})PO(t) = \begin{cases} 1 & \text{if } \ell(t) \leq \ell_{\max} \\ f_{perf} & \text{if } \ell_{\max} < \ell(t) \leq (\ell_{\max} + f^{\Delta}_{perf}) \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

where $\ell(l)$ corresponds to the average latency, $\ell_{\max}$ to the maximum latency the SaaS provider has to guarantee and f^{Δ}_{perf} to the fuzziness interval. f_{perf} corresponds to the fuzziness function and takes into account the percentage of fuzziness and the confidence to return 0, if the SaaS violates the SLA, or 1 otherwise.

$$(\text{SaaS})\theta(t) = \frac{PO(t) \cdot \mu(t)}{\omega(t)} \tag{12}$$

where $\omega(t)$ corresponds to the number of instances allocated to the service at time t and $\mu(t) \in \{0, 1\}$ to the utility function associated to the service mode (video/image) at time t. More precisely, it takes into consideration the percentage of service mode and return 0, if the SaaS violates the SLA, or 1 otherwise.

Similar to the IaaS decision module, the SaaS decision module is also modeled in CP. Its objective is to find a minimum resource allocation that keeps the latency lower than $\ell_{\max}$, or in the last case lower than $\ell_{\max} + f^{\Delta}_{perf}$ for a given workload (number of client requests).

6.3. Control algorithm

IaaS. First, we define the decision variable z_i, whose domain $D(z_i) \in \{0, 1\}.\forall i \in [1, p]$. This variable indicates whether or not a node $pm_i \in P$ should be shutdown. The second part consists of a set of constraints over the decision variables z_i.

Eq. (13) states that the total power consumption of nodes should not exceed $\epsilon_{\max}$.

$$\sum_{i=1}^{p} z_i * pw_i \leq \epsilon_{\max} \tag{13}$$

where pw_i corresponds to the power consumption of a node pm_i.

Finally, Eq. (14) corresponds to the objective function that tries to maximize the sum of all availabilities.

$$maximize \left(\sum_{i=1}^{v} \alpha_i \right) \tag{14}$$

where v corresponds to the number of virtual machines and α_i, the service availability.

SaaS. The first part of the CP model is a set of decision variables: x, whose domain $D(x) \in [1, m]$, is a variable indicating the usage mode configuration; y_{ij}, where $D(y_{ij}) \in [1, \mathbb{N}].\forall i \in [1, n]$, $\forall j \in [1, q]$, is a variable indicating the number of virtual machines of class m_j allocated to component c_i; u_i, where $D(u_i) \in \{0, 1\}.\forall i \in [1, n]$, is a variable indicating whether the component is used by the mode x. Eq. (15) is the objective function that maximizes the performance (for a given usage mode x and allocation matrix y) and minimizes the total number of virtual machines allocated.

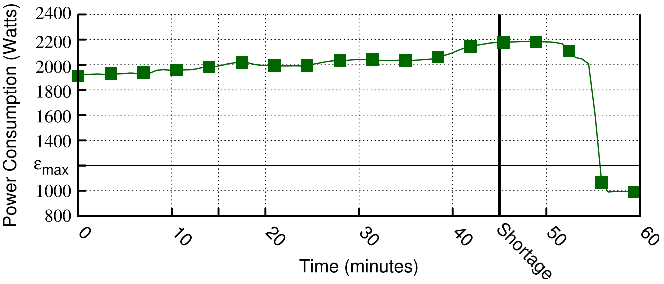


Fig. 11. Power consumption before and after an *Energy Shortage*.

$$maximize \left(\frac{(\ell^{obj}(x, y) + \mu_x)}{\sum_{i=1}^{n} \sum_{j=1}^{q} y_{ij}} \right). \tag{15}$$

6.4. Experimental results

Setup. For this use case, we relied on a sub-set of 13 nodes from Grid'5000 [14], a French grid for experimental testbed. Two nodes were used to host the controllers, i.e., one per service instance (two for the SaaS provider and one for the IaaS provider); one node was used to host load injectors (one for each Ad Client); and ten nodes were used to host the VMs containing the SaaS services themselves. The duration of the experiments was fixed at one hour, during which the workload of both SaaS instances increases from 0 to 140 request per second. It means that both instances require the same amount of resources until reaching the peak load.

An *Energy Shortage* is scheduled to be triggered at 45 min of execution. The objective is to observe how IaaS and SaaS providers face with this situation while taking into consideration the SLAs defined in the previous section. For the IaaS provider, the observation interval was fixed in six minutes, whereas for the SaaS provider it was fixed in one minute. It means that the availability of the IaaS at a given observation interval is calculated by the percentage of time the resources are accessible within the six minutes of the concerned interval. Regarding the SaaS, the response time is calculated by the average of all the requests within one minute interval. These intervals are inspired by real world cloud providers such as Amazon EC2 and Microsoft Azure.

Results. Fig. 11 shows the total power consumption of the infrastructure, before and after the *Energy Shortage* event. This event contains the number of nodes that should be shutdown in order to maintain a certain level of power consumption ($\epsilon_{\max}$). The interval between the event detection (vertical line) and the decrease in the power consumption is due to the timeout between the *Scale Down* notification (from the IaaS provider to the SaaS provider) and the actual shutdown of nodes. In this scenario, we specified an $\epsilon_{\max}$ of 2400 W before and 1200 W after the shortage event. Based on the total power consumption of each node, it is possible to have the number of nodes needed to be shutdown in order to keep the power consumption under the specified $\epsilon_{\max}$. In that case, 1200 W corresponds to five nodes, i.e., 50% of the total infrastructure.

It is straightforward that level of shortage may lead to an impact on the service availability. Fig. 12 depicts the IaaS availability all through the experiment. We have established the observation interval every 6 min so that we could have 10 intervals within one hour. As it can be seen, until the shortage event, all the intervals have 100% of availability and thus respect the SLA established in

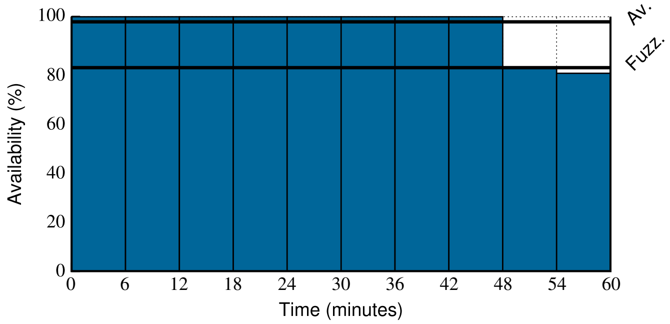


Fig. 12. IaaS availability.

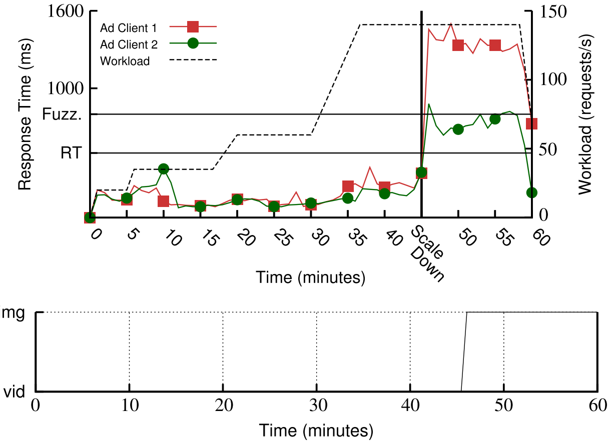


Fig. 13. The average response time before and after the *downscaling* and the usage mode change.

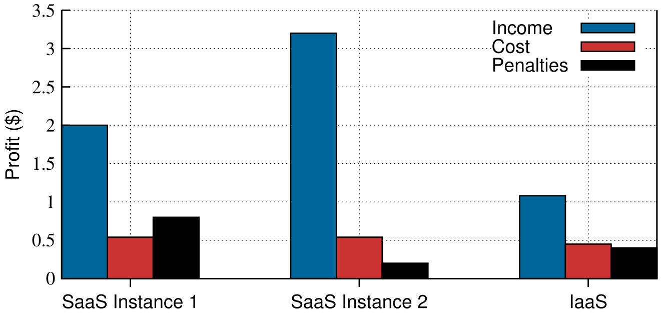


Fig. 14. The incomes, cost and penalties for SaaS and IaaS providers.

Table 5. After the shortage event, the availability decreases to 80% for the first observation interval and to 78% for the last one. This decrease is explained by the VMs made unavailable due to the shortage.

According to the SLA expressed in Table 5, in each observation interval the availability must not be lower than 98%, with confidence 100%. Since there is a fuzziness of 18% for 10% of the 10 intervals, the first interval after the shortage will not be considered as a violation. The last interval, instead, violates the specified SLA, because it exceeds the 98% of availability and does not meet the fuzziness conditions.

The *Energy Shortage* at the IaaS layer forces SaaS providers to *Scale Down*, that is, to work with less resources than allocated to them. As a consequence, the *downscaling* may also impact on the SaaS QoS and thus lead to SLA violations. Fig. 13 shows the QoS in terms of Average Response Time of the SaaS for Ad Clients 1 and 2 according to a workload variation. According to the SLA expressed in Table 6, the average response time must not exceed 500 ms. Hence, there is no penalty before the *downscaling*, since none of the observation intervals has an average response time greater than 500 ms.

It should be reminded that the SLA specifies a confidence of 90%, and fuzziness of 300 ms for 20% of requests. Even though, after the *downscaling*, the SaaS instance for the Ad Client 1 gets penalties for each observation intervals exceeding 500 ms of average response time beyond the 10% of intervals accepted thanks to the confidence margin. Contrary to the Ad Client 1, the SLA established with Ad Client 2 accepts a functional degradation for 20% of requests within a time window. That way, the SaaS instance for Ad Client 2 is turned into a functional degraded mode (image) so as to absorb the same workload while avoiding SLA violations related the QoS. It should be noticed that those violations are avoided thanks to the fuzziness. Indeed, with the functional degradation the SaaS provider manages to keep the average response time between 500 and 800 ms. However, a few violations are not avoided at the end of the time window, because both the fuzziness and usage mode works only for 20% of requests.

Finally, Fig. 14 depicts the incomes, costs and penalties of both SaaS instances as well as the IaaS. Not surprisingly, both instances have the same cost since the same amount of resources is allocated to them. The SaaS instance 1 has a lower income with respect to instance 2, since it is able to process less requests within the same amount of time (lower throughput). It is also important to observe that the penalties are higher for the instance 1 then for instance 2. This is because of the numerous SLA violations after the forced downscaling, which occurs at the end of the experiments for requests that do not meet neither the fuzziness nor the functional degradation conditions. With respect to the IaaS, the income corresponds to the costs of instance 1 and 2 together. For simplicity reasons, the IaaS costs are due to the energy consumption, that is, the amount of energy multiplied by the current energy fees. Finally, the IaaS penalties are payed by the IaaS to the SaaS provider due to violations in the availability caused by the energy shortage.

To sum up, this use case is important to show how the SLA can be used to guide decisions in several layers of the cloud stack. Moreover, it shows how new properties of CSLA (QoS/functionality degradation, advanced penalty model) introduce a fine language support for Cloud elasticity management. Indeed, interesting features such as the usage mode, fuzziness and confidence can be used so as to turn SaaS and IaaS providers more flexible and able to seamlessly cope with extreme situations such as the energy shortage.

7. Related work

7.1. SLA specification

Historically, SLA has been used since the 1980s in a variety of areas such as Networking and Web Services. The Web services community has performed significant level of research in SLAs languages. Several languages, such as SLAng [25], WSLA [7] and WS-Agreement [8], have been proposed for SLA specification using a XML-based language. All these works have contributed significantly to the standardization of SLA. However, none meets the needs for cloud computing environment and particularly the elasticity concept. In Cloud computing, a SLA has to be suitable for multiple layers (XaaS) with heterogeneous and volatile resources in a highly dynamic environment. Moreover, performance of cloud services may fluctuate due to the dynamic Internet environment, which makes the QoS inherently uncertain.

More recently, initiatives such as SLA@SOI [26] or Optimis [27] have addressed SLA specification for Clouds. The SLA@SOI language (SLA*) is based on the WS-Agreement while the Optimis language (WSAG4J) is a full Java-based implementation of WS-Agreement and WS-Agreement Negotiation. Their solutions covers SLA lifecycle. However, the violations management does not reflect cloud characteristics. In addition, in SLA*, the description of SLA is just limited to guarantee terms while external file (e.g., OVF) is needed to describe IaaS services. The CSLA language shares motivations with the SLA@SOI project and goes further by taking into account the cross-layer nature of Cloud and QoS instability: CSLA allows defining SLA in any language for any cloud service (XaaS) in the same file and allow services providers to address violations. Finally, CSLA supports open standards to address the need for interoperability in the field of cloud computing (OCCI [5], NIST [6]).

7.2. SLA control

Existing public clouds provide very few guarantees in terms of performance and dependability [28]. Amazon EC2 compute service offers a service availability of at least 99.95% [29], and Amazon S3 storage service guarantees a service reliability of 99.9% [29]. However, in case of an outage, Amazon requires the customer to send them a claim within thirty business days for Amazon EC2 and ten days for Amazon S3. Amazon cloud services do not provide performance guarantees or other QoS guarantees. Rackspace and Azure cloud services provide similar behaviors [30,31].

Several recent research works consider SLA in cloud environments [32–35]. Chhetri et al. propose the automation of SLA establishment based on a classification of cloud resources in different categories with different costs, e.g. on-demand instances, reserved instances and spot instances in Amazon EC2 cloud [32]. However, this approach does not provide guarantees in terms of performance, nor dependability. Macias and Guitart follow a similar approach for SLA enforcement, based on classes of clients with different priorities, e.g. Gold, Silver, and Bronze clients [33]. Here again, a relative best-effort behavior is provided for clients with different priorities, but neither performance nor dependability SLOs are guaranteed. Other works consider to better tune MapReduce systems for performance improvement [36,37], target other specific environments such SaaS [35], or propose heuristics for SLA management [34]. However, these works provide best-effort behavior without strict guarantees on SLA, and do not tackle the many types of clouds.

Recent works have also addressed SLA guaranties for cloud services. [38] defines an architecture focused on detecting anomalies rather than modeling the cloud behavior, as it is done in this paper. [39] proposes an SLA-aware PaaS MapReduce, however, its solution reacts at scheduling time without capturing cloud dynamicity during the execution of jobs. Our work proposes the CSLA language and an online control architecture to cope with cloud dynamicity. [40] takes into account maximization of provider profit in addition to SLA constraints. They model the PaaS as an optimization problem. [41] has developed an SLA-aware PaaS Database service defining SLOs on database specific metrics like data freshness. They provision virtualized database replicas based on a previous experimental characterization of the service instead of modeling its behavior. Concerning locking control, in the majority of work found in the literature that provide locking services with time constraints, accesses to the shared resource are usually ordered based on the priorities assigned to each request rather than the deadline when the locking request should be satisfied. Therefore, several priority-based algorithms have been proposed to cope with time requirements [42–44]. They usually exploit a token-based distributed algorithm and a priority level, that dynamically changes, is associated to every process's locking request.

Few locking services explicitly address real-time constraints [45,46] whose algorithms directly take into account deadlines of requests and processes agree on the same order of locking request satisfactions.

With respect to SLAs and energy management, [47] proposes an autonomic framework that deals with several energy-related subproblems (e.g. load balancing, frequency scaling) across multiple layers of the IT infrastructure so as to guarantee SLAs established between applications and end-users. [48] proposes an approach for the coordination of multiple control loops in order to manage power in infrastructures and performance SLAs of applications. Although those work provide a nice contribution for cross-layered power management and SLA guarantees, they fall short to explore the variety of features that can be formalized within SLAs (e.g. QoS degradation) such as in CSLA. As a consequence, contracts are more constrained and services less adaptable for certain circumstances.

8. Conclusion

We have presented in this paper a new method that combines Quality of Service (QoS) with Service Level Agreement (SLA) in clouds aiming at facing challenges such as better performance, dependability, or cost reduction of online cloud services. To this end, we introduce the SLAaaS model which enables the integration of service levels and SLA into clouds. We also propose the CSLA language for finely expressing SLA definition and addressing SLA violations in the context of Cloud services. Furthermore, objective functions can be defined to ensure the best SLA guarantees since it allows the specification of the SLOs of the SLA in question as well as the trade-off, priority, and weight among them. Finally, the behavior of the cloud is characterized, and online control algorithms are applied in order to achieve the SLOs, and, therefore, the respective SLA guarantees.

Three use cases with evaluation performance results (one for SaaS layer and two for PaaS layer) confirm the solidity and usefulness of the proposed method. In the near future, we intend to extend them or propose new ones applying other metrics, such as service throughput or energetic cost. We have also presented a fourth use case concerning energy shortage that shows that our method can be used for multi-layer SLAs, avoiding then, SLA violation domino effect.

Although in this paper we have considered only the number of instance as a metric for cost in the SLA, as a perspective for this work, we will take into consideration a cloud billing model for cost which relates resource cost and the time units (e.g. hours) resources are used.

This work opens also interesting perspectives in terms of SLA governance. We plan to allow cloud customers to be part of the loop and to be automatically notified about the state of the cloud, such as SLA violation and cloud energy consumption. A second future work could involve other QoS aspects of cloud services such as privacy and security guarantees.

Acknowledgments

This work was supported by the ANR agency, under the MyCloud project (ANR-10-SEGI-0009, http://mycloud.inrialpes.fr/). Part of the experiments were conducted on the Grid'5000 experimental testbed (http://www.grid5000.fr/).

References

[1] Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region, 2011. http://aws.amazon.com/fr/message/65648/.
[2] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, A. Shraer, Verifying cloud services: Present and future, Oper. Syst. Rev. 47 (2) (2013).
[3] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, P. Sens, Towards QoS-oriented SLA guarantees for online cloud services, in: 13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid, 2013.

[4] Y. Kouki, T. Ledoux, CSLA: a language for improving cloud SLA management, in: 2nd Int. Conf. on Cloud Computing and Services Science, CLOSER, 2012.

[5] T. Metsch, A. Edmonds, Open cloud computing interface—infrastructure. Open Grid Forum, 2011.

[6] L. Fang, T. Jin, M. Jian, B. Robert, L.B. John Messina, D. Leaf, NIST cloud computing reference architecture, 2011.

[7] H. Ludwig, A. Keller, A. Dan, R.P. King, R. Franck, Web service level agreement (WSLA) language specification. Tech. Rep., IBM, 2003.

[8] A. Andrieux, et al. Web services agreement specification (ws-agreement). OGF, 2007.

[9] D. Irwin, L. Grit, J. Chase, Balancing risk and reward in a market-based task service, in: 13th IEEE Int. Symp. on High Performance Distributed Computing, HPDC, 2004.

[10] Transaction Processing Performance Council, 2014. TPC-W. www.tpc.org/tpcw.

[11] N. Gautam, Analysis of Queues: Methods and Applications, CRC Press, 2012.

[12] P. Harrison, N.M. Patel, Performance Modelling of Communication Networks and Computer Architectures, Addison-Wesley, 1992.

[13] J. Arnaud, S. Bouchenak, Performance and Dependability in Service Computing, IGI Global, 2011.

[14] R. Bolze, et al., Grid'5000: A large scale and highly reconfigurable experimental grid testbed, Int. J. High Perform. Comput. Appl. (IJHPCA) 20 (4) (2006).

[15] Z. Fadika, M. Govindaraju, DELMA: Dynamically ELastic MapReduce framework for CPU-intensive applications, in: 11th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid, 2011.

[16] A.W. Gordon, P. Lu, Elastic phoenix: Malleable MapReduce for shared-memory systems, in: 8th IFIP Int. Conf. on Network and Parallel Computing, NPC, 2011.

[17] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: 6th USENIX Symp. on Operating Systems Design and Implementation, OSDI, 2004.

[18] T. White, Hadoop: The Definitive Guide, first ed., O'Reilly Media, Inc., 2009.

[19] A. Sangroya, D. Serrano, S. Bouchenak, Benchmarking dependability of MapReduce systems, in: 31st IEEE Int. Symp. on Reliable Distributed Systems, SRDS, 2012.

[20] MovieLens web site, 2014. http://movielens.umn.edu/.

[21] M. Burrows, The chubby lock service for loosely-coupled distributed systems, in: 7th USENIX Symp. on Operating Systems Design and Implementation, OSDI, 2006.

[22] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the clouds: A berkeley view of cloud computing. Tech. Rep., University of California, Berkeley, 2009.

[23] J. Lejeune, L. Arantes, J. Sopena, P. Sens, Service level agreement for distributed mutual exclusion in cloud computing, in: 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid, 2012.

[24] J.G. Koomey, Growth in data center electricity use 2005 to 2010. Tech. Rep., Analytics Press, 2011.

[25] D. Lamanna, J. Skene, W. Emmerich, SLAng: A language for defining service level agreements, in: 9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS) Rico, Proceedings, 2003, p. 100.

[26] SLA@SOI project, 2012. http://sla-at-soi.eu/.

[27] W. Ziegler, M. Jiang, OPTIMIS SLA framework and term languages for SLAs in cloud environment. Deliverable D2.2.2.1, OPTIMIS European project, 2011.

[28] S.A. Baset, Cloud SLAs: present and future, ACM SIGOPS Oper. Syst. Rev. 46 (2) (2012).

[29] Amazon Web Services, 2012. http://aws.amazon.com/.

[30] Rackspace SLA, 2012. http://www.rackspace.com/cloud/legal/sla/.

[31] Windows Azure, 2012. http://www.microsoft.com/windowsazure.

[32] M.B. Chhetri, Q.B. Vo, R. Kowalczyk, Policy-based automation of SLA establishment for cloud computing services, in: 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid, 2012.

[33] M. Macias, J. Guitart, Client classification policies for SLA enforcement in shared cloud datacenters, in: 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid, 2012.

[34] H. Goudarzi, M. Ghasemazar, M. Pedram, SLA-based optimization of power and migration cost in cloud computing, in: 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid, 2012.

[35] L. Wu, S.K. Garg, R. Buyya, SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments, in: 11th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid, 2011.

[36] H. Herodotou, S. Babu, Profiling, what-if analysis, and cost-based optimization of mapreduce programs, in: 37th International Conference on Very Large DataBases, VLDB, 2011.

[37] A. Verma, L. Cherkasova, R.H. Campbell, Aria: Automatic resource inference and allocation for mapreduce environments, in: The 8th ACM International Conference on Autonomic Computing, ICAC, 2011.

[38] A.C. Oliveira, H. Chagas, M. Spohn, R. Gomes, B.J. Duarte, Efficient network service level agreement monitoring for cloud computing systems, in: 2014 IEEE Symposium on Computers and Communication (ISCC), IEEE, 2014, pp. 1–6.

[39] F. Teng, F. Magoulès, L. Yu, T. Li, A novel real-time scheduling algorithm and performance analysis of a mapreduce-based cloud, J. Supercomput. (2014) 1–27.

[40] D. Dib, N. Parlavantzas, C. Morin, SLA-based profit optimization in cloud bursting PaaS, in: 14th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid. Chicago, United States, May 2014.

[41] S. Sakr, L. Zhao, A. Liu, Clouddb autoadmin: A consumer-centric framework for SLA management of virtualized database servers, in: Large Scale and Big Data—Processing and Management, 2014, pp. 357–388.

[42] A.M. Goscinski, Two algorithms for mutual exclusion in real-time distributed computer systems, J. Parallel Distrib. Comput. 9 (1) (1990) 77–82.

[43] S. Kanrar, N. Chaki, FAPP: A new fairness algorithm for priority process mutual exclusion in distributed systems, J. Netw. 5 (1) (2010) 11–18.

[44] Y.-I. Chang, Design of mutual exclusion algorithms for real-time distributed systems, J. Inf. Sci. Eng. 11 (4) (1994) 527–548.

[45] K. Han, Scheduling distributed real-time tasks in unreliable and untrustworthy systems (Ph.D. thesis), Faculty of the Virginia Polytechnic Institute and State University, USA, 2010.

[46] R. Rajkuman, Synchronization in Real-time Systems; A Priority Inheritance Approach, Kluwer Academic Publishers, Boston, 1991.

[47] D. Ardagna, B. Panicucci, M. Trubian, L. Zhang, Energy-aware autonomic resource allocation in multitier virtualized environments, IEEE Trans. Serv. Comput. 5 (1) (2012) 2–19.

[48] X. Wang, Y. Wang, Coordinating power control and performance management for virtualized server clusters, IEEE Trans. Parallel Distrib. Syst. 22 (2) (2011) 245–259.