2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

A Language Support for Cloud Elasticity Management

Yousri Kouki[1], Frederico Alvares de Oliveira Jr.[1], Simon Dupont[1 2] and Thomas Ledoux[1]
[1] ASCOLA Research Group
Mines-Nantes, INRIA, LINA
Nantes, France
{Firstname.Lastname}@inria.fr
[2] SIGMA Informatique
Nantes, France
{sdupont}@sigma.fr

***Abstract*—Elasticity is the intrinsic element that differentiates Cloud computing from traditional computing paradigm, since it allows service providers to rapidly adjust their needs for 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 non-negligible resource initiation time, network fluctuations or unpredictable workload, it becomes hard to guarantee QoS levels and SLA violations may occur. This paper proposes a language support for Cloud elasticity management that relies on CSLA (Cloud Service Level Agreement). CSLA offers new features such as QoS/functionality degradation and an advanced penalty model that allow providers to finely express contracts so that services self-adaptation capabilities are improved and SLA violations minimized. The approach was evaluated with a real infrastructure and application testbed. Experimental results show that the use of CSLA makes Cloud services capable of absorbing more peaks and oscillations by trading-off the QoS levels and costs due to penalties.**

***Keywords*-Cloud computing, Elasticity, QoS, SLA, Language**

I. Introduction

With the popularization of the Internet, there has been an increasing demand for 24/7 available services. Services have to cope with varying demands while delivering the same Quality of Service (QoS) level. Cloud computing came with the hope to deliver computing as utility by enabling the on-demand and remote access to a set of shared resources [10], where resources can be any IT resource (hardware infrastructure, platforms, software, etc.) that is made available in the form of services consumed in a pay-as-you-go manner. As a consequence, service providers are able to rapidly adjust their needs for resources to absorb the demand and hence guarantee a minimum level of QoS that respects the Service Level Agreements (SLAs) previously defined with their clients.

Although this elasticity capability, which enables resource provisioning almost instantly, there are some technical and conceptual limitations that may turn the adaptation process in a cumbersome and hence difficult to deal with highly dynamic environment (e.g., workload variations within the same business day or hour). First, the non-negligible resource initiation time may prevent just-in-time provisioning to absorb workload peaks and thus avoid SLA violations. This can be even worse during oscillating scenarios as it may have a ”ping pong effects” in the request/release of resources. In addition, the granularity of the resources reservation time and the implied resource billing model (e.g., hourly, daily, etc.) may also lead service providers to either pay more than they actually consume (e.g., when it requests resource during a workload peak and releases it right after) or to take into consideration the reservation duration before deciding to request more resource. Therefore, it becomes really hard for service providers to manage resources and avoid SLA violations in this kind of unpredictable and dynamic environment.

This paper presents a language-based approach to deal with SLA-driven Cloud Elasticity Management. This proposition uses CSLA (*Cloud Service Level Agreement*) [13], a SLA language to finely express SLA and 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 features (QoS/functionality degradation and an advanced penalty model) introducing a fine language support for Cloud elasticity management. Our goal is to make contracts more flexible and consequently increase services’ self-adaptation capability and elasticity possibilities. In this context, service providers can develop a set of self-adaptation policies that are (automatically) derived from the SLA description. Those polices take into consideration the QoS/functionality degradation in order to absorb some violations and hence reduce costs implied by penalties. To this end, we rely on (i) software potential variability to allow SaaS services operating in different modes (e.g., 2D vs 3D display, a degree of security levels); (ii) existing rule-based auto-scaling techniques presented in CloudStack [2], one of the most widely adopted cloud management tool.

In order to evaluate the suitability of our language-based approach, we performed some experiments on a real infrastructure testbed. Results show that the use of

978-1-4799-2784-5/14 $31.00 © 2014 IEEE
DOI 10.1109/CCGrid.2014.17

CSLA makes services capable of absorbing more peaks and oscillations by trading-off the QoS levels and costs due to penalties.

The rest of this paper is organized as follows. Section 2 illustrates a motivating scenario, which is used to ease the understanding of our approach. Section 3 describes CSLA in more details. Section 4 shows how the CSLA features can be used along with elasticity tools. The results obtained from experimental evaluation are presented and discussed in Section 5. Section 6 provides a selection of relevant work related to this paper. Finally, Section 7 concludes this paper and provides some discussion on future work.

II. Example Scenario: Public Transport Itinerary Service

This section describes a motivating scenario of a Cloud application that is used all through the paper so as to ease the understanding of the proposed approach.

As it can be seen in Figure 1, it consists of a Transport Itinerary service in a SaaS fashion that depends on a third-party compute/storage resource provisioned in terms of Virtual Machines (VMs). The Itinerary SaaS provider is a IaaS client, whereas the clients of the SaaS provider are public transport companies, which in turn, offers itinerary services to their users.

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 a minimum level of availability. 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 a time window.

The SaaS provider offers two services with the same functionality but not the same quality: S_1 may operate on only one mode (normal), whereas S_2 may operate on two (normal and degraded). In the normal mode, the service returns three itineraries, while in the degraded one, only one itinerary is returned to the end-user. The SaaS clients (public transport companies) are charged according to the service use in a timely manner (e.g., daily, weekly, monthly, etc.). A SLA is established between the SaaS provider and each one of its clients stating that the provider has to guarantee a minimum level of service with respect to usage mode (normal/degraded) and other quality of service criteria such as response time. Upon contract violations, penalties should be paid back to clients as compensation.

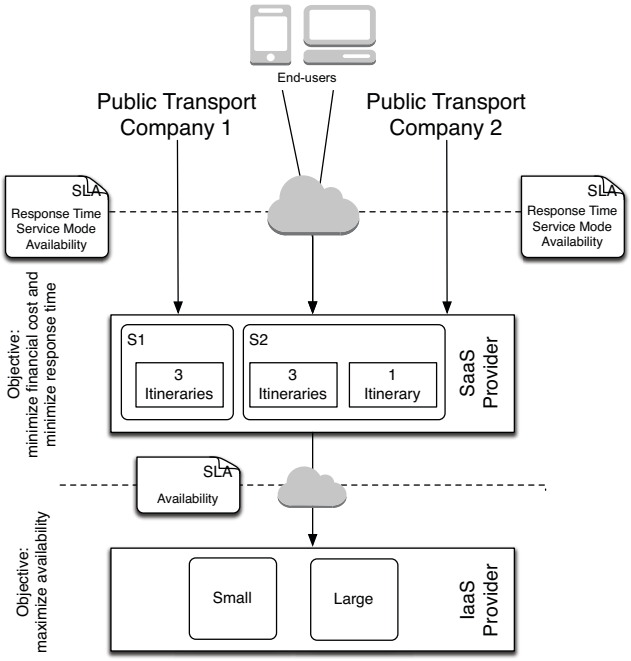


Figure 1. Actors of the multi-SLA scenario

III. CSLA Language

CSLA, the Cloud Service Level Agreement language, allows to define SLA in any language for any Cloud service (XaaS). CSLA addresses intrinsically (i) QoS uncertainty in unpredictable and dynamic environment; (ii) the cost model of Cloud computing. A preliminary version of CSLA has been introduced in our previous work [13]. This new version proposes more features such as functionality degradation to address SLA violations and is based on the Open Cloud Computing Interface (OCCI) [17] and the Cloud computing reference architecture of the National Institute of Standards and Technology (NIST) [10].

A. Conceptual Model

A SLA in the CSLA language contains three sections: a section describing the validity of the agreement, a section defining the parties involved in the agreement and a section referencing the template used to create the agreement.

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 to any XaaS service (SaaS, PaaS or

IaaS). In order to deal with unpredictable and dynamic environment, we propose the concept of functionality degradation (i.e., normal vs degraded *mode*) for each SaaS and PaaS services (e.g., 3D vs 2D display). We show the advantage of a such feature in the next section. We follow the OCCI standard for IaaS services definition.

Parameters provide a way to define variables in the context of the agreement that should be used in others sections. Variables refer to a distinct element such as *Metric*, *Monitoring* and *Schedule*.

Guarantees contain a set of guarantees. Each guarantee consists of four elements: *Scope*, *Requirements*, *Terms* and *Penalties*. A 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 terms 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 constraint and 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*). Similar to service definition, we propose a set of QoS degradation definitions to deal with unpredictable environments: (i) the *fuzziness* defines the acceptable margin degree around the threshold of an expression; (ii) the *confidence* defines the percentage of compliance of clauses.

In case of SLA violation, penalties are applied to the service provider in order to compensate the consumer for tolerating the service failure. The compensation can be applied either as a constant or variable rate. CSLA supports two types of billing: *Pay as You Go* and *All-in package*. Finally, the *Agreement* starts on *effectiveFrom* and ends on the earlier of the *effectiveUntil* or in accordance with *Terminations* section.

In conclusion, among its novelties, CSLA integrates features dealing with QoS uncertainty in unpredictable Cloud environment: functionality degradation (service *mode*), QoS degradation (*fuzziness*, *confidence*) and an advanced penalty model. In addition, CSLA allows defining SLA in any language (e.g., XML, Java) for any Cloud service (XaaS) and it supports open standards.

B. Concrete syntax

According to a Cloud service, a CSLA template is generated with pre-defined parameters to ensure that offered QoS guarantees are realistic and realizable[1]. In this paper, we use XML as a representation format. The following XML (see Listing 1) 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 S2 (see Section 2).

In this example, three SLOs are composed using the "*And*" operator: a performance SLO, a dependability SLO and a mode usage SLO. In fact, we offer a measurable metric (Mode Usage - *Mu*) to evaluate the use of the functionality degradation. The performance SLO (lines 7-9), for instance, specifies that for each interval of 3mn in window of 10mn (expressed in the variable *Mon-1*, not detailed here), the maximum request response time (*Rt*) must be below 3s. This objective should be achieved every day between 7am and 22pm (expressed in the variable *Sch-Morning*, not detailed here) during the validity of the contract. It is guaranteed on at least 99% of requests to the Cloud service (*confidence*) among which 10% can be degraded (*fuzziness*), i.e., a margin of 0.2 second is acceptable as a QoS degradation.

Lines 10-12 show the dependability SLO whereas lines 13-15 specify the mode usage SLO. The functionality degradation must be used in 30% of requests for the Cloud service S2. Notice that the functionality degradation is managed like any other SLOs since it defines an objective of usage.

The second part presents the penalties (lines 17-38). 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 [12].

A violation of the dependability SLO (lines 26-31) or the mode usage SLO (lines 32-37) implies a penalty equal to 0.1 euro/request whereas the penalty of the performance SLO (lines 18-25) depends on the exceeding delay. The request price is modeled as: $P = \alpha - \beta \cdot dt$; where α is the price with no violation ($\alpha > 0$), 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 (lines 22-24/28-29/34-36) 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).

Listing 1. CSLA example.

```
1  <csla:terms>
2  <csla:term id="T1" operator="and">
3  <csla:item id="availabilityTerm"/>
4  <csla:item id="responseTimeTerm"/>
5  <csla:item id="modeTerm"/>
6  </csla:term>
7  <csla:objective id="performanceSLO" priority="1" actor="provider">
8  <csla:expression metric="Rt" comparator="lt" threshold="3" unit="
       second" monitoring="Mon-1" schedule="Sch-Morning"
       Confidence="99" fuzziness-value="0,2" fuzziness-percentage="
       10"/>
```

[1]This pre-calibration process is out of scope of this paper.

```
</csla:objective>
<csla:objective id="dependabilitySLO" priority="2" actor="provider">
<csla:expression metric="Av" comparator="gt" threshold="98" unit="
    \%" monitoring="Mon-2" Confidence="99" fuzziness-value="1
    " fuzziness-percentage="5"/>
</csla:objective>
<csla:objective id="modeSLO" priority="3" actor="provider">
<csla:expression metric="Mu(S1-M2)" comparator="lt" threshold="30"
     unit="\%" monitoring="Mon-1" Confidence="99" fuzziness-
    value="5" fuzziness-percentage="5"/>
</csla:objective>
</csla:terms>
<csla:penalties>
<csla:Penalty id="p-Rt" objective="responseTimeTerm" condition="
    violation" obligation="provider">
<csla:Function ratio="0,5" variable="delais" unit="second">
<csla:Description> ... </csla:Description>
</csla:Function>
<csla:Procedure actor="provider" notificationMethod="e-mail"
    notificationPeriod="7 days">
<csla:violationDescription/>
</csla:Procedure>
</csla:Penalty>
<csla:Penalty id="p-Av" objective="availabilityTerm" condition="
    violation" actor="provider">
<csla:Constant value="0,1" unit="euro/request"/>
<csla:Procedure actor="provider" notificationMethod="e-mail"
    notificationPeriod="7 days">
<csla:violationDescription/>
</csla:Procedure>
</csla:Penalty>
<csla:Penalty id="p-Mu" objective="modeTerm" condition="violation
    " obligation="provider">
<csla:Constant value="0,1" unit="euro/request"/>
<csla:Procedure actor="provider" notificationMethod="e-mail"
    notificationPeriod="7 days">
<csla:violationDescription/>
</csla:Procedure>
</csla:Penalty>
</csla:penalties>
```

C. Economic Model

1) Pricing Model: The major advantage of the functionality degradation is that it can be used in the design of autoscaling policies in order to cope with peak loads of short duration or to absorb significant instances initialization time. Hence, providers profit remains strongly linked to the management of service costs based on the *confidence* and *fuzziness* properties as well as the functionality degradation (*service mode*). Service providers can then better plan their resources, but without disturbing their consumers, since the latter will be charged according to an advanced economic model. This model aligns penalties with functionality/QoS degradation in order to provide good trade-off price-quality which is both attractive for final clients and profitable for SaaS vendors. Hence, the service price will be adjusted according to the confidence, fuzziness and modes in a win-win strategy.

In CSLA, we propose to clearly contractualize the percentage of use of each one of those properties by a respective metric. In our case study, the price has been adjusted following statistics on simulations. We analyze these statistics to define a value both attractive for final clients and profitable for SaaS vendors. The most successful business models such as the ones based on yield management [9] would make it possible to refine these rates.

2) Penalty Model: CSLA offers a sophisticated penalty model based the QoS degradation feature.

Ideal/degraded/inadequate: We distinguish three states of a request (or a time interval): ideal, degraded and inadequate (see Figure 2). Ideal means that the objective threshold is respected. The QoS degradation consists of utilizing an error margin (*fuzziness value*). Beyond this margin, it is an inadequate state.

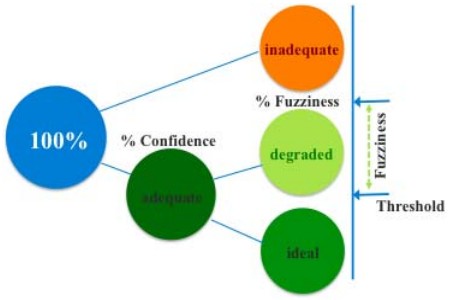


Figure 2. Ideal, degraded and inadequate

Evaluation Process: In order to evaluate an objective (SLO), an initial evaluation enables to classify the request/interval as ideal, degraded or inadequate. We distinguish two concepts (see Figure 3): (i) *per-interval evaluation*, in which the evaluation is performed at the end of each interval; (ii) *per-request evaluation*, in which the objective is evaluated for each request. A final evaluation, at the end of the time window, allows one to verify an objective (SLO) by applying the *fuzziness* and *confidence* percentages to the initial evaluation. The final evaluation enables the identification of non-accepted/accepted degraded and inadequate cases, that is, that will/will not result penalties. In other words, the final evaluation absorbs or notifies the violations.

Figure 3. Per-request/Per-interval Evaluation

IV. Cloud Elasticity Management

This section presents a set of self-adaptation policies for optimizing the Cloud elasticity management. We consider a cross-layer elasticity, i.e., from infrastructure

to software resource elasticity (according to the CSLA functionality degradation feature).

A. IaaS Elasticity

Static threshold-based rules method is the most popular for Capacity Planning and it has been used by Cloud providers like Amazon EC2 [1] and Microsoft Azure [4] or Cloud Manager like CloudStack [2].

The main idea behind this method is that the capacity might vary according to a set of rules, where each rule is based on one or more metrics (e.g., response time, availability or CPU usage). A rule may be composed of an upper threshold *upper*, a lower threshold *lower*, two time values ($time_{upper}$, $time_{lower}$), during which the metric is greater/lower than the corresponding threshold, and two calm durations: $calm_{add}$ and $calm_{remove}$ during which no scaling decisions can be committed in order to prevent system oscillations. Based on those parameters, the rules can be defined as follows:

Scale Out:
If $m > upper$ *for* $time_{upper}$ *then* $capacity = capacity + k_{add}$
Do nothing for $calm_{add}$

Scale In:
If $m < lower$ *for* $time_{lower}$ *then* $capacity = capacity - k_{remove}$
Do nothing for $calm_{remove}$

where m is the metric, capacity is the current capacity, k_{add} and k_{remove} are respectively the capacity to add and the capacity to remove.

B. SaaS Elasticity

We propose a SLA-driven variability approach in which we consider Cloud applications as white boxes (i.e., we can handle SaaS components lifecycle and connections between them). We define rules to manage the architectural elasticity that are (automatically) derived from the SLA description and more particularly the CSLA functionality degradation feature:

Scale App:
If $m > upper$ *for* $time_{mode}$ *then* $Mode = degraded$

where m is a high-level metric for end-user, such as response time.

The main idea is to fix the value of the parameter $time_{mode}$ lower than the value of the parameter $time_{upper}$. Indeed, after $time_{mode}$, the application is encouraged to switch to degraded mode – according to the CSLA mode usage – to avoid to trigger the *Scale Out* rule at the IaaS level. The return to normal mode will be controlled by the SaaS adaptation manager that specifies the degradation period, the frequency of use and the window.

Figure 4 summarizes the rules both at the application and infrastructure levels. Each *Scale Out* action will be followed by a modification at the SaaS level. The idea here is to absorb the increasing load during the initiation time of the resources using the degraded mode. Once instances are activated, the SaaS application switches to normal mode.

In conclusion, those rules – automatically derived from the CSLA description – help service providers to manage finely resources and avoid SLA violations.

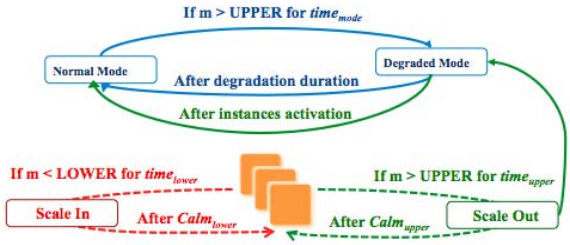


Figure 4. Elasticity management rules overview

V. Experiments

This section presents the results obtained from some experiments. In order to evaluate the proposed approach, we have explored the execution of a real use case scenario deployed on a real physical infrastructure.

A. Experimental Testbed

In this section, we describe the testbed used in the experiments, as it is depicted in Figure 5.

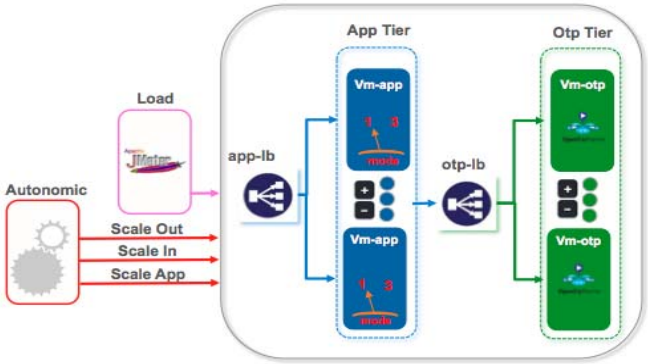


Figure 5. Testbed architecture

Infrastructure: We consider two kinds of VMs: small and large. The small one is configured with a 2 GHz CPU core and 1 GB of RAM. The large one is configured with two 2 GHz CPU core and 4 GB of RAM. Both small and large VMs are shipped with CentOS 6.0 (64-bit) operating system.

Two TCP-level load balancers (LB) with least connection algorithms and without sticky session have been deployed for distributing workloads across multiple VMs. Both VMs and LBs rely on a widely adopted Cloud management tool: Apache CloudStack [2]. CloudStack is an open source Cloud computing software for creating, managing, and deploying infrastructure Cloud services. It uses existing hypervisors for virtualization. The VMware's hypervisor called vSphere [2], is used in our experiments.

Two other small VM instances are added to the infrastructure testbed without being connected to the LB. The first one, called *vm-autonomic*, hosts the autonomic managers of the system providing interfaces to monitor and act on the system. The second one, named *vm-load*, is used for load injection purposes. It contains an Apache JMeter [3] application that is used to simulate a load variation on the business application and to analyze overall performance under different types of workload.

SaaS application: We developed a public transport web application, based on OpenTripPlanner (OTP [4]), an open source platform for multi-modal trip itinerary planning and network analysis. Our application offers adaptation abilities since the itinerary calculation component provides two response levels. The first response level, called *highResponseLevel*, returns three different itineraries in response to a request whereas the second one, named *lowResponseLevel*, returns only one itinerary. The basic idea is to trade off the service modes (number of itineraries) and the costs due to compute resources. Indeed, the first level offers the best quality of functionality to end-users but is the most resource consuming (CPU, bandwidth), by requiring the calculation of three itineraries, whereas the latter provides a minimum service while consuming fewer resources.

The application is architecturally organized in 2-tiers. The first tier corresponds to the business tier, which consists of a Spring-based business application deployed on small VM presented earlier (*vm-app*). An Apache server [5] responding to the incoming HTTP requests is installed for each *vm-app*, whereas the second tier corresponds to an OTP component. It is deployed on a large VM type (*vm-otp*). All *vm-app* are assigned to the *app-lb* load balancer whereas *vm-otp* are assigned to the *otp-lb* load balancer. That way, it is possible to scale both the business application and the route calculation component.

[2]http://www.vmware.com/products/vsphere/
[3]http://jmeter.apache.org/
[4]http://opentripplanner.com/
[5]http://httpd.apache.org/

B. Prototype

This paragraph aims at presenting the prototype including autonomic managers of the system providing interfaces to monitor metrics about the system state, analyze them and act on the system at runtime.

Monitoring: The monitoring rests on a log-based solution. When a *vm-app* or a *vm-otp* is deployed, they log different low-level metrics (e.g., CPU load, memory) corresponding to their health. In addition to these metrics, the *vm-app* logs high-level metrics (e.g., response time, HTTP status codes) corresponding to the Apache Tomcat server access log. Each VM runs an agent through the upstart system daemon which is responsible for regularly sending metrics to an asynchronous message broker.

Analyze: VM metrics collected and sent by agents into the message queue are centralized in the *vm-autonomic*. From these received messages, we identify some specific events using WildCAT [8], a Java framework for context-aware applications. WildCAT allows to easily organize and access sensors through a hierarchical organization. It becomes easy to trigger actions upon particular conditions which can be convenient in the case of autoscaling rules.

Execution: We have implemented actions interface to act on both the IaaS and the SaaS layers. At the infrastructure level, we are able to deploy/destroy VMs and more generally manage our Cloud infrastructure using the CloudStack API. At the software level, we can order a configuration change to the application asking it to degrade/upgrade some functionalities.

C. SLAs and Auto-scaling Rules

A SLA is established between the IaaS provider and its client (in this case the SaaS provider) stating that the IaaS provider must guarantee at least 99.95% of availability, as it is shown in Table I. 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. In case of violation, a financial compensation should be given to the client, that is,

10% or 30% of the service price back depending on the degree of violation (cf. Table I).

Similarly, the SaaS provider establishes a SLA with its clients (cf. Table II) in which it should guarantee an average response time less than or equal to 750*ms*, with confidence, fuzziness and percentage fuzziness of 90%, 50*ms* and 16.66%, respectively. It means that the average response time measured within an observation period may exceed 800*ms* in at most 10% of the observation periods within a predefined time window and may be between 750*ms* and 800*ms* in at most 15% (90% of 16.66%) of the measured values within a predefined time window. Regarding the availability, the SLA states that at least 99.5% of availability should be guaranteed, with 0.5% of fuzziness, meaning that the availability can be lower then 99% in 10% of the observation periods and between 99% and 99.5% in 15% (90% of 16.66%) of the observation periods. Regarding the functionality degradation, for S1, only normal mode is accepted (i. e., 3 itineraries returned), whereas for S2, up to 30% of the observation periods may be accepted on degraded mode (i.e., only 1 itinerary returned). A violation of the response time SLO (resp. the availability and mode usage SLOs) implies a penalty equal to 0.003$/*request* (resp. 0.001$/*request* and 0.0005$/*request*).

In order to respect the SLA defined in Table II, four autoscaling rules are defined at the IaaS level.

app-tier

Rule r1: *If (avg(cpuLoad of vm – app linked to the load balancer app – lb)) > 75% for 60s then Scale Out: deploy new vm – app and link it to the app – lb.*

Rule r2: *If (avg(cpuLoad of vm – app linked to the load balancer app – lb)) < 40% for 60s then Scale In: remove vm – app.*

otp-tier

Rule r3: *If (avg(cpuLoad of vm – otp linked to the load balancer otp – lb)) > 80% for 60s then Scale Out: deploy new vm – otp and link it to the otp – lb.*

Rule r4: *If (avg(cpuLoad of vm – otp linked to the load balancer otp – lb)) < 30 for 60s then Scale In: remove vm – otp.*

Only one autoscaling rule is defined at the SaaS level so as to guarantee the SLA defined in Table II.

Rule r5: *If (avg(responseTime of vm-app linked to app-lb)) > 750ms for 30s then Scale App: degrade service QoS for all instances (3 itineraries to 1 itinerary) during 3 minutes.*

In the case of SaaS elasticity implementation, *r1* and *r3 Scale Out* actions are followed by *r5 Scale App* action: degrade service QoS for all instances (3 itineraries to 1 itinerary).

D. Evaluation Scenario

We compare the business application behavior on two architectures: i) a traditional Cloud hosting, providing IaaS elasticity, in which machines are switched on/off on demand meeting certain auto-scaling rules; and ii) a fully elastic Cloud hosting providing, multi-layer elasticity, able to scale both at the IaaS level, by allocating or removing resources following auto-scaling rules, and at the SaaS level by degrading the functionality delivered to end-users to save resources according to specific rules.

Comparing *with/without SaaS Elasticity* implementations could be seen as the same as comparing CSLA with any SLA language. All SLA languages have the same general purpose, although each one has its own particularities. A WS-Agreement/SLA* contract, for instance, could be formalized with CSLA by fixing the confidence to 100% and fuzziness to 0. However, this is not reciprocal because CSLA offers more properties that is not considered in existing SLA languages. Regarding the functional degradation (mode usage), while the SLOs would be easily described as it would be for any metric in any SLA language, the service description with different modes would not be possible in the existing languages.

The initial architecture of our experiments for these two scenarios is composed of one *vm-app* and two *vm-otp* for the business application. We have studied a scenario that consists of a single peak of workload: 5 threads are started and maintained for 20 min, whereupon we simulate a peak of workload passing from 5 to 15 threads (clients). We can imagine that a strike manifestation affecting the public transportation service has led to this workload peak.

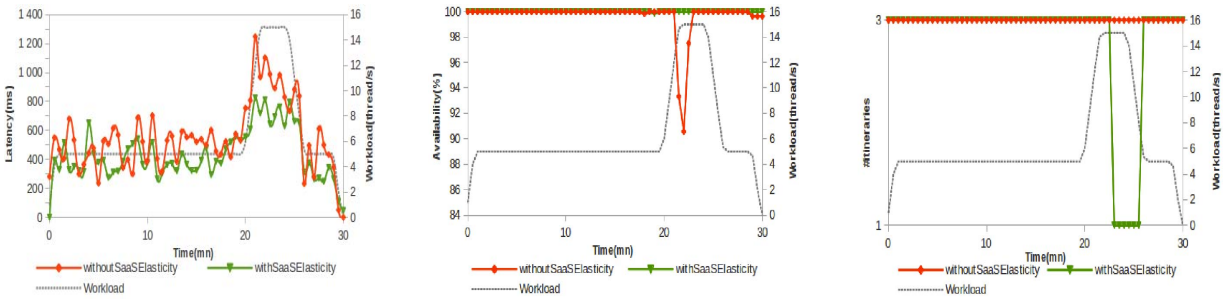
For the IaaS provider, the observation interval was fixed in five minutes, whereas the evaluation window was fixed in on month, just like real world Cloud providers such as Amazon EC2 and Microsoft Azure. It means that the availability of the IaaS at a given observation interval is calculated by the percentage of time the resources are accessible within the five minutes of the concerned interval. For the SaaS provider, the observation interval and evaluation window were fixed in 30 seconds and 10 minutes, respectively. The availability is calculated similarly to the IaaS provider,

Table I
SLA between the IaaS and SaaS providers

service	metric	oper.	threshold	fuzz.	% of fuzz.	conf.	price	penalty
Small/large	Av	≥	99.95%	0	0	100	0.046$ /h	10% if 99.95 ≤ Av ≤ 99% 30% if Av ≤ 99%

Table II
SLA between the SaaS provider and customers

service	metric	oper.	threshold	fuzz.	% of fuzz.	conf.	penalty
S1/S2	Rt	≤	750ms	50ms	16.66%	90%	0.003$/rqt
	Av	≥	99.5%	0.5%			0.001$/rqt
S2	Mu(degraded)	≤	30%	5%			0.0005$/rqt



(a) Response Time (b) Availability (c) Mode Usage

(d) Resources cost (e) Penalties cost

Figure 6. *withoutSaaSElasticity* vs *withSaaSElasticity*

whereas the response time is calculated by the average of all the requests within one 30-seconds interval.

E. Results and Discussion

This use case shows how interesting CSLA features such as the functionality degradation can be used so as to turn SaaS elasticity more flexible and able to seamlessly cope with some technical and conceptual limitations (e.g., the non-negligible resource initiation time).

Figure 6 depicts the results for the experiments with two implementations (*withSaaSElasticity* and *withoutSaaSElasticity*) considering the SLA given in Table II. As shown in the graphs, a variable number of clients was used for these experiments (initially 5, then 15 and finally 5 again). Every 30 seconds, the number of concurrent clients, the average request response time and the availability are monitored.

At the beginning, the *app-tier* is composed of only one VM whereas the *otp-tier* consists of two VMs. When the number of concurrent clients begins to rise at $t = 20\ mn$, the *withSaaSElasticity* implementation has prevented the addition of resources absorbing the workload increase by using the functionality degradation whereas the *withoutSaaSElasticity* implementation has required to start and add a new *vm-app* to the *app-tier* (see Figure 6(d)). Moreover, we can see that SaaS elasticity is more reactive than the resource one to cope with rapid workload fluctuation.

Figures 6(a), 6(b) and 6(c) show respectively the QoS

in terms of response time, availability and mode usage of the SaaS application for the two implementations according to a workload variation. When workload increases to 15 threads/s, the request response time (see Figure 6(a)) also increases above the SLO (750 ms) for the *withoutSaaSElasticity* implementation. This implementation detects that the capacity (number of VMs) would lead to violations of the SLA and triggers an action at IaaS level to avoid them. IaaS elasticity reaches its limits when SaaS applications need to keep a certain level of QoS, which is to say that it could be more interesting to answer all the requests even with a degraded functionality. In this case, the SaaS elasticity is imperative to achieve such a level of flexibility. The *withSaaSElasticity* implementation increases the self-adaptation capability and allows the just-in-time adjustment. As we can see in Figure 6(a), the response time for the *withSaaSElasticity* implementation does not exceed 800 ms during peak. Figure 6(b) shows the service availability over the execution of the experiment when the load varies. When the load increases, there is more rejected requests with the *withoutSaaSElasticity* implementation. However, the *withSaaSElasticity* implementation switches to degraded mode (1 itinerary) in order to keep availability as specified by the SLA. Figure 6(c) shows the mode usage objective. The *withSaaSElasticity* implementation switches between 3 itineraries (normal mode) and 1 itinerary (degraded mode) to absorb workload peaks.

In order to ease the results comparison, for Figure 6(e), we have evaluated only the last window of the experiment, i.e., the last 10 minutes during which the workload varies. We can see that the *withSaaSElasticity* implementation performs best to reduce the number of SLA violations. Besides, the customer perception is better since the ideal requests/answers are more numerous with the *withSaaSElasticity* implementation than with the *withoutSaaSElasticity* implementation.

Finally, without functionality degradation, the SaaS provider has a lower income since it is able to process less requests within the same amount of time (lower throughput).

To sum up, the results show that the functionality degradation is very effective to make SaaS application more flexible and therefore more able to adapt to absorb peaks. Moreover, it succeeds in keeping a better ratio of the QoS to the service cost. The SaaS elasticity is complementary alternative of IaaS elasticity to meet SLA requirements.

VI. Related Work

A. SLA Languages 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 WSLA [14] and WS-Agreement [5] have been proposed for SLA specification using a XML-based language. All these works have contributed significantly to the emergence of SLA. However, none meets the needs for Cloud computing environment and particularly they do not address the SLA violations in this kind of unpredictable and dynamic environment.

More recently, initiatives such as SLA@SOI [3] or Optimis [19] have addressed SLA specification for Clouds. The SLA@SOI [3] 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 cover 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 files are needed (e.g., Open Virtualization Format[6]) 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 allows service providers to address violations.

B. SLA Control

Existing public Clouds provide very few guarantees in terms of performance and dependability [6]. For example, Amazon EC2 compute service offers a service availability of at least 99.95% [1] but Amazon Cloud services do not provide performance guarantee or other QoS guarantees.

Several recent research works consider SLA in Cloud environments [16], [7], [15], [11], [18]. [16] proposes a SLA-driven resource allocation scheme but does not provide cross-layer elasticity management. [7] proposes 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. However, this approach does not provide guarantees in terms of performance, nor dependability. [15] follows a similar approach for SLA enforcement, based on classes of clients with different priorities, e.g., Gold, Silver, and Bronze clients. Here again, a relative best-effort behavior is provided for clients with different priorities, but neither performance nor dependability SLOs are guaranteed. Another body of work target other specific issue such as efficient resource allocation

[6]http://www.dmtf.org/standards/ovf

for SaaS providers [18] or propose heuristics for SLA management [11]. However, these works provide best-effort behavior without strict guarantees on SLA.

In conclusion, existing SLA languages or solutions are not enough sufficient to express finely SLA which prevent service providers to develop (i) elasticity policies that are (automatically) derived from the SLA description; (ii) just-in-time provisioning to absorb workload peaks and thus avoid SLA violations.

VII. Conclusion

Cloud computing promises to completely revolutionize the way to manage resources. Thanks to the elasticity capability, resources can be provisioned within minutes to satisfy a required level of QoS formalized by SLAs between different Cloud actors. However, due to non-negligible resource initiation time, network fluctuations or unpredictable workload, QoS uncertainty remains a reality thus SLA violations are possible. In order to address this issue, we propose a language support for Cloud elasticity management: CSLA. In comparison with others SLA languages, CSLA proposes new features related to services functionality/QoS degradation and an advanced penalty model that allow service providers to maintain its consumers satisfaction while minimizing the service costs due to resources fees. Our experiments indicate that SaaS providers – by relying on CSLA features – manage to reduce the number of violations and hence increase their profit.

In the next future, we plan to experiment our approach in a large scale environment (hundreds of nodes and thousands of VMs). Besides, we will develop a framework to automatize the generation of scaling rules from the SLAs contracts. Finally, we plan to allow Cloud customers to be aware of the SLA governance: they will be automatically notified about the state of the Cloud, such as SLA violations and Cloud energy consumption.

Acknowledgement

This work is supported by the MyCloud project (ANR-10-SEGI-010, *http://mycloud.inrialpes.fr*) and SIGMA Informatique (*http://www.sigma.fr*).

References

[1] Amazon EC2. http://aws.amazon.com/ec2/, 2013.

[2] Apache CloudStack. http://cloudstack.apache.org/, 2013.

[3] Sla@soi. sla-at-soi.eu/, 2013.

[4] Windows Azure. http://www.windowsazure.com/, 2013.

[5] A. Andrieux and al. *Web services agreement specification (ws-agreement)*. OGF, 2007.

[6] S. A. Baset. Cloud SLAs: Present and Future. *ACM SIGOPS Operating Systems Review*, pages 57–66, 2012.

[7] M. B. Chhetri, Q. B. Vo, and 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)*, pages 164–171, 2012.

[8] P.-C. David and T. Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7. ACM, 2005.

[9] P. Dube, Y. Hayel, and L. Wynter. Yield management for it resources on demand: Analysis and validation of a new paradigm for managing computing centres. *J. Revenue Pricing Manag.*, pages 24–38, 2005.

[10] L. Fang, T. Jin, M. Jian, B. Robert, L. B. John Messina, and D. Leaf. NIST Cloud Computing Reference Architecture. 2011.

[11] H. Goudarzi, M. Ghasemazar, and 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)*, pages 172–179, 2012.

[12] D. Irwin, L. Grit, and J. Chase. Balancing Risk and Reward in a Market-based Task Service. In *13th IEEE Int. Symp. on High Performance Distributed Computing (HPDC)*, pages 160–169, 2004.

[13] Y. Kouki and T. Ledoux. CSLA: a Language for Improving Cloud SLA Management. In *2nd Int. Conf. on Cloud Computing and Services Science (CLOSER)*, pages 481–591, 2012.

[14] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. Web Service Level Agreement (WSLA) Language Specification. Technical report, IBM, 2003.

[15] M. Macias and 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)*, pages 156–163, 2012.

[16] S. Son and S. C. Jun. Negotiation-based flexible sla establishment with sla-driven resource allocation in cloud computing. In *13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 168–171, 2013.

[17] M. Thijs and E. Andy. Open Cloud Computing Interface - Infrastructure. 2011.

[18] L. Wu, S. K. Garg, and 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)*, pages 195–204, 2011.

[19] W. Ziegler and M. Jiang. OPTIMIS SLA Framework and Term Languages for SLAs in Cloud Environment. Technical report, 2011.