2012 IEEE Fifth International Conference on Cloud Computing

An Integrated Approach for Specifying and Enforcing SLAs for Cloud Services

André Lage Freitas, Nikos Parlavantzas, Jean-Louis Pazat
Université Européenne de Bretagne
INSA, INRIA, IRISA, UMR 6074
F-35708 Rennes, France
Emails: {Andre.Lage,Nikos.Parlavantzas,Jean-Louis.Pazat}@irisa.fr

***Abstract*—Cloud computing has evolved from the provisioning of virtual machines to the provisioning of complex services, delivered to customers under the terms of Service-Level Agreements (SLAs). SLAs specify the Quality of Service (QoS) that should be provided to customers as well as the billing model. A main concern for cloud service providers is to maintain the agreed SLA terms in order to avoid losses and penalties. Maintaining the SLA in turn requires translating the QoS to configurations of low-level mechanisms, able to enforce the agreed terms. Current systems provide no integrated support for SLA specification, translation, and enforcement. In this paper, we propose an approach for specifying and enforcing SLAs for cloud service providers. The approach covers the creation of SLA templates under a billing model, the design of performance and fault-tolerance QoS assurance mechanisms as well as the translation of QoS to appropriate configurations of those mechanisms. We demonstrate the feasibility of our approach by using the Qu4DS framework for PaaS cloud providers. Moreover, we evaluate the impact of failures on the provider profit. The experiments were carried out on the Grid5000 testbed and demonstrate the effectiveness of ensuring fault tolerance in different scenarios.**

I. Introduction

The cloud computing paradigm has evolved from the provisioning of virtual machines to the provisioning of complex services. Higher-level services are built on top of distributed infrastructures in a layered fashion as proposed by the common cloud architecture [29], [6], [26]. The cloud architecture decomposes resources, platforms, and the final software in three layers: IaaS (Infrastructure as a Service), PaaS (Platform as a Service), and SaaS (Software as a Service). Example services at each layer include Amazon EC2 [1], SalesForce [28], and Google Maps [11] respectively.

The interactions between customers and cloud service providers are typically specified in Service-Level Agreements (SLAs). SLAs cover customer and provider obligations and further details about the service, such as functional requirements as well as the Quality of Service (QoS) that should be delivered along with the service. Moreover, SLAs should include billing details which describe the pricing model and how penalties are computed in case of SLA violations. For instance, Zencoder [34] offers video and audio encoding services and charges customers based on each minute of encoded video output. Regarding SLA violations, if Zencoder services are not available at least 99.9% in a month, Zencoder fully refunds customers by means of service credits. Similarly, Amazon EC2 [1] also pays service credits to customers if virtual machines are available less than 99.95% in a month.

Although SLAs are useful for describing the service along with its QoS and billing aspects, SLAs are not enough to ensure QoS. SLAs serve as guidelines to which the service execution should comply in order to prevent losses. Therefore, QoS assurance mechanisms should be integrated with the service execution in order to ensure that the service will be delivered as agreed. Moreover, the employment of QoS assurance mechanisms requires translating high-level SLA objectives to system-level configurations.

Current work fails to integrate SLA description, translation and enforcement. Indeed, some approaches describe SLAs and their pricing model but do not address QoS assurance [1], [34], [25], [9]. These approaches also fail to include fines into their pricing model. Other approaches deal with QoS assurance but describe neither details about SLA nor billing concerns [15], [13], [16], [23]. Finally, other approaches focus on translating SLA to low-level configurations [12], [30], [22] but they neither specify how QoS can be ensured nor apply QoS under a pricing model.

In this paper, we propose an integrated approach for SLA management. Firstly, we introduce the context of our approach in Section II. Section III discusses the creation and translation of SLA templates based on the customization of performance and fault-tolerance QoS. We also include into the SLA description a pricing model which considers operational expenses and currency fines in a pay-per-use fashion. In Section IV, we present two QoS assurance mechanisms which ensure performance and fault-tolerance QoS along with the translation of QoS to low-level mechanism configurations. Section V explains how the Qu4DS framework [8] is used as proof of concept for our approach. Qu4DS provides automatic SLA management functionalities which include service negotiation, resource booking, instantiation, billing, SLA translation, and QoS assurance in a transparent fashion. Furthermore, experiments were performed on Grid'5000 in order to analyze the impact on the service profit for different scenarios. The results of these experiments are presented in Section VI.

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.135

II. The Problem

The Figure 1 introduces the context of this work by describing a fictitious example based on real cloud services. A music group wants to publish its album in an audio sharing service such as Jamendo [14]. The group uploads their songs to Jamendo in a lossless audio format, e.g., FLAC [31], as required by Jamendo. In order to save storage space, Jamendo asks the Zencoder [34] audio encoder service to compress the FLAC songs to the lossy OGG [32] format. In turn, Zencoder books virtual machines from a cloud provider such as Amazon EC2 [1] in order to encode the songs. After encoding the songs, Zencoder forwards them to Jamendo which publishes the album along with the OGG songs for streaming and downloading.

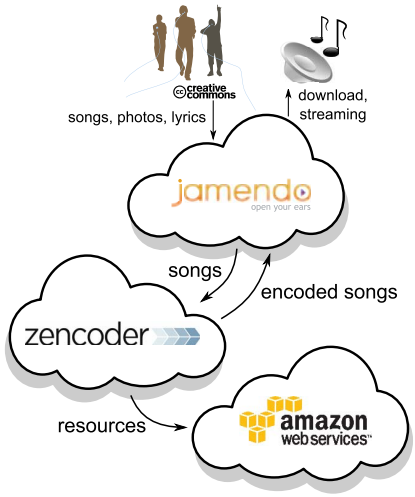

Fig. 1. An example of how cloud services can be used to support a music sharing service.

The example depicted by Figure 1 can be realized using current cloud services – in fact, Zencoder uses Amazon EC2. However, *the challenge is to provide QoS in this context*, e.g., to define which level of QoS Zencoder can offer to Jamendo in order to ensure that the songs will be encoded successfully and that the encoding process will not take longer than an agreed timeout. The SLA agreed between Zencoder and Jamendo should include QoS metrics and the penalties that should be applied if Zencoder does not guarantee QoS. In order to address this problem, the first issue is to define which QoS metrics can be provided in such a dynamic, unpredictable, and distributed environment. Secondly, it is necessary to design and implement QoS assurance mechanisms able to ensure the defined QoS. Thirdly, a mapping between high-level QoS metrics and low-level means is necessary in order to configure QoS assurance mechanisms. Finally, the SLA should also include a complete billing model which includes pricing, infrastructure costs and fine payments.

This work provides a solution for the stated problem by focusing on supporting cloud service providers in guaranteeing SLA terms. The approach integrates SLA description, negotiation and enforcement in a transparent way. In the following, we explain how SLA templates are created, the description of the billing model, the design of QoS assurance mechanisms, and the translation of QoS to system-level configurations.

III. Creating Service-Level Agreements

A. SLA Templates

SLA templates are useful for enabling service negotiation. This work assumes that SLA templates are provided by the service provider. Figure 2 depicts a contract establishment followed by a request treatment. First, costumers choose an SLA template based on the desired level of QoS. Once chosen, they set the contract duration and propose the customized template to the service provider. The provider may reject the contract proposal, otherwise it is accepted and a contract is established. Following that, customers are able to send requests. The idea is to rely on a simple SLA negotiation protocol based on contract templates which include the description of the involved parties, the terms as well as the QoS that should be met by the provider. For instance, the aforementioned negotiation requirements are supported by complete negotiation specifications and protocols such as WSLA [24] and WS-Agreement [3].

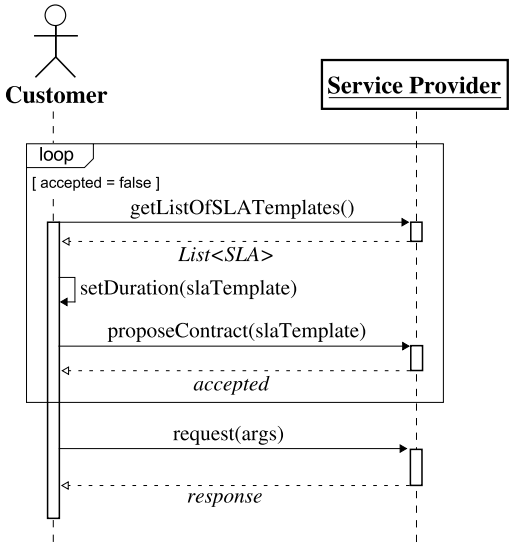

Fig. 2. Sequence diagram of an SLA negotiation based on templates defined by the service provider.

This approach includes SLA templates which describe performance and fault-tolerance QoS. Performance is addressed by defining the *response time* QoS metric, which specifies the maximum amount of time that a request treatment can take. The fault tolerance QoS metric is *reliability* which specifies the degree of dependability. Moreover, both response time and reliability are quantified by the high-level constraints *strong, medium*, and *weak* [27].

Finally, SLA templates are created based on the customization of QoS constraints. In order to ease the identification

of SLA templates, labels are used to name them according to their main quality characteristic. The Table I depicts four SLA templates whose labels are *fast, safe, classic* and *standard*. Setting SLA templates labels involves defining the degree of performance and fault tolerance of each SLA template. Because providing fault tolerance capabilities adds operational overhead, performance and fault tolerance cannot be prioritized simultaneously. Therefore, SLA templates are customized by prioritizing one QoS metric at a time or none of them. For instance, the *fast* SLA template prioritizes performance, while the *safe* SLA template prioritizes fault tolerance. On the other hand, the *classic* and *standard* SLA templates prioritize neither performance nor fault tolerance.

QoS Metrics / Labels	**Response Time**	**Reliability**
Fast	*strong*	*weak*
Safe	*weak*	*strong*
Classic	*medium*	*medium*
Standard	*weak*	*weak*

TABLE I
SLA TEMPLATES, THEIR LABELS AND RESPECTIVE QOS CONSTRAINTS.

B. Pricing

It is important to set the price as a function of the expenses in order to prevent losses. Thereby the targeted profit margin is guaranteed as the price fluctuates according to the expenses. The service price is set based on operational costs as depicted by Equation 1. The price is represented by ρ where l is the SLA template label and t is the contract duration. The expenses for providing the service are represented by ϵ which quantifies the costs of resource acquisition; however, expenses may include software licenses as well as further required services. Finally, π represents how much the provider wants to profit from the given SLA template label and contract duration.

$$\rho(l,t) = \epsilon(l,t) + \pi(l,t) \tag{1}$$

Customer billing is assumed to be computed according to service usage, i.e., in a pay-per-use fashion [20], [21]. Service usage is measured based on the chosen contract template and on the contract duration. In other words, each contract template has its own cost per time unit (cf. Equation 1). Moreover, the time metric is used for generality as it is suitable for most type of services, e.g., monthly installments. However, further metrics such as number of requests, concurrent usage and key performance indicators can be added in order to support specific services. Note that this work is not limited to the pay-per-use model. For instance, subscription-based models are often employed by current cloud providers. They rely on a predefined amount of billing metrics to be consumed given a billing time unit (e.g., ten gigabytes per month). In this example, pay-per-use would assess how many megabytes were consumed given a time interval. However, if we impose to customers a predefined amount of data (i.e., ten gigabytes) given a time period (i.e., month), then pay-per-use is transformed in a subscription model.

This work assumes that penalties owing to SLA violations are payed in monetary terms. An SLA violation means the unsuccessful request treatment and implies the payment of the fine ψ as depicted in Equation 2; where f_ψ is a constant which is used to adjust ψ as a function of the cost of treating the request q_k, and $resp\_time_{q_k}$ is the response time of q_k. In contrast, current cloud providers [1], [10], [28], [34] rely on service credits as fine payments in order to prevent losses since it is very hard to ensure any QoS in a dynamic and distributed environment. However, providing QoS assurance mechanisms encourages cloud service providers to rely on monetary fines. The advantage of paying fines in monetary terms is that it allows the cloud service provider to positively differentiate itself from its competitors which rely on service credits.

$$\psi(l,q_k) = f_\psi \cdot \rho(l, resp\_time_{q_k}) \tag{2}$$

Ultimately, the provider profit P is calculated as depicted by Equation 3. Given the provisioning time interval $[t,t']$, the provider held n contracts in such a way that ρ_i is the price of the contract c_i and ϵ_i represents the expense of c_i. The term $\psi_{i,j}$ represents fine cost of the j-th request that belongs to c_i and will be taken into account if such a request could not be treated.

$$P_{[t,t']} = \sum_{i=0}^{n} \rho_i - \sum_{i=0}^{n} \epsilon_i - \sum_{i=0}^{n}\sum_{j=0}^{m} \psi_{i,j} \tag{3}$$

IV. QOS ASSURANCE MECHANISMS

A. QoS Translation

QoS translation refers to mapping QoS constraints to the right configuration which enables the system to deliver the targeted QoS level. The QoS translation is represented by the generic function τ as Equation 4 depicts where qos is a QoS constraint and sys_config is a system configuration.

$$\tau(qos) = sys\_config \tag{4}$$

This work describes two QoS assurance mechanisms which handle performance and fault tolerance. The configuration of the performance QoS assurance mechanism uses the *resource requirements* able to meet a given response time; hence the translation is done by $\tau(resp\_time_{cons}) = res\_req_{cons}$ where $cons$ is a QoS constraint. Moreover, the mapping between response time and resource requirements is done by profiling the service provider. The configuration of the fault-tolerance QoS assurance mechanism uses *replacement threshold* parameters as it relies on a job replacement algorithms. Thus the translation of the reliability QoS constraints is done by $\tau(reliability_{cons}) = (failure\_th_{cons}, delay\_th_{cons})$ where $failure\_th$ and $delay\_th$ are replacement thresholds for failed and delayed jobs respectively.

B. Performance

The response time QoS is ensured by booking resources, configuring the service instance, and deploying the instance based on the minimal resource requirements[1] able to meet the given response time. The performance QoS assurance mechanism is described by the sequence diagram depicted by Figure 3. When a customer proposes a contract, the service provider translates the response time to its respective resource requirements based on profiling data. Then the service provider acquires resources from the infrastructure according to the previous resource requirements until the end of the contract duration. The translated resource requirements are also used to configure and deploy the service instance. When the service instance is operational, the customer can send requests which are forwarded to the right service instance. Thus, requests are then treated by the service instance according to the resource requirements to which the service instance is configured.

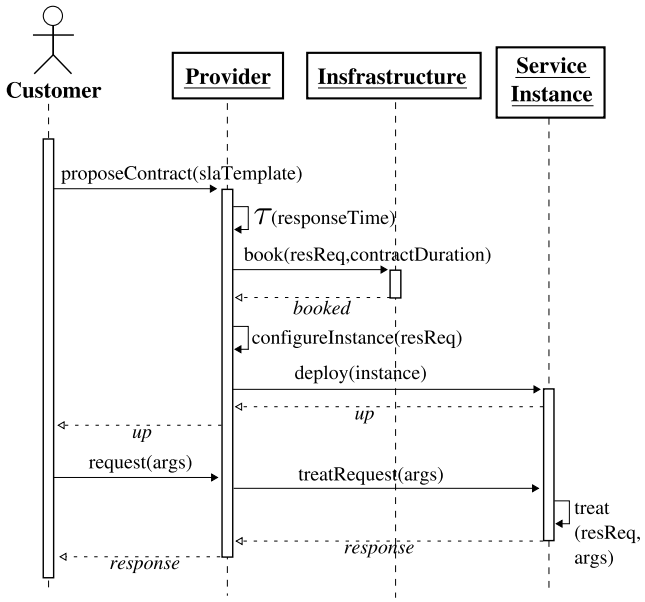

Fig. 3. Sequence diagram of the performance QoS assurance mechanism which books resources, configures and deploys service instances based on resource requirements able to meet the agreed response time.

C. Fault Tolerance

The goal of the fault-tolerance QoS assurance mechanism is to improve the provider ability of overcoming malfunctions during request treatment. In this context, malfunctions can be either job failures or job delays. While a failed job is considered to be an unsuccessful job execution (e.g., an I/O error), identifying job delays requires information regarding the provider profiling, i.e., job execution time and request response time.

[1]Minimal resource requirements are preferred to others that achieve the same response time but imply a greater operational cost.

The fault-tolerance QoS assurance mechanism relies on job replacement algorithms. The basic idea is to replace failed and delayed jobs in order to not compromise the request treatment. The Algorithm 1 replaces failed jobs up to the failed replacement threshold ($failure\_th_{j_i}$); where n indicates how many times the job was already replaced, J_{q_k} is the set of all jobs belonged to q_k, J^F is the set of failed jobs, J^R is the set of running jobs, $elapsed\_time_{q_k}$ is the request elapsed time, and $adapt\_th_{j_i}$ is the adaptation threshold, i.e., the time for triggering an adaptation action. Similarly, the Algorithm 2 replaces delayed jobs up to its replacement threshold ($delay\_th_{j_i}$) where J^C is the set of canceled jobs. In addition, diagnosing a delayed job requires comparing its elapsed time ($elapsed\_time_{j_i}$) with its execution time ($exec\_time_{j_i}$) gathered from the provider profiling.

Algorithm 1 Job Failure Tolerance

Require: $j_i^n \in J_{q_k} \wedge j_i^n \in J^F$
Ensure: $j_i^{n+1} \in J_{q_k} \wedge j_i^{n+1} \in J^R$
1: **if** $elapsed\_time_{q_k} < adapt\_th_{j_i}$ **and** $n \leq failure\_th_{j_i}$ **then**
2: $n \leftarrow n+1$
3: create and run j_i^n
4: **else**
5: abort $q_k, j_i^n \in J_{q_k}$
6: **end if**

Algorithm 2 Job Delay Tolerance

Require: $j_i^n \in J_{q_k} \wedge j_i^n$ is delayed
Ensure: $j_i^n \in J^C \wedge j_i^{n+1} \in J_{q_k} \wedge j_i^{n+1} \in J^R$
1: **if** $elapsed\_time_{q_k} < adapt\_th_{j_i}$ **and** $n \leq delay\_th_{j_i}$ **and** $elapsed\_time_{j_i} > exec\_time_{j_i}$ **then**
2: cancel j_i^n
3: $n \leftarrow n+1$
4: create and run j_i^n
5: **else**
6: abort $q_k, j_i^n \in J_{q_k}$
7: **end if**

Moreover, further fault-tolerance techniques can be used by the aforementioned Algorithms 1 and 2 such as including job replication, migration, or checkpointing.

V. Proof of Concept

We use the Qu4DS (Quality Assurance for Distributed Services) framework [8] to validate the approach proposed in this paper. Qu4DS is a research prototype for supporting the development and management of cloud services. The Qu4DS architecture abstracts over distributed infrastructures as depicted by Figure 4. Qu4DS creates SLA templates under a pricing model as described in Section III. Then Qu4DS manages SLAs which includes service negotiation, instantiation and provisioning. Qu4DS also implements the QoS assurance mechanisms presented in Section IV whose configurations

are generated by translating QoS constraints. Thereby, performance is ensured by booking resources and configuring the service instance according to the right resources requirements. Fault tolerance is ensured by replacing failed and delayed jobs during the request treatment.

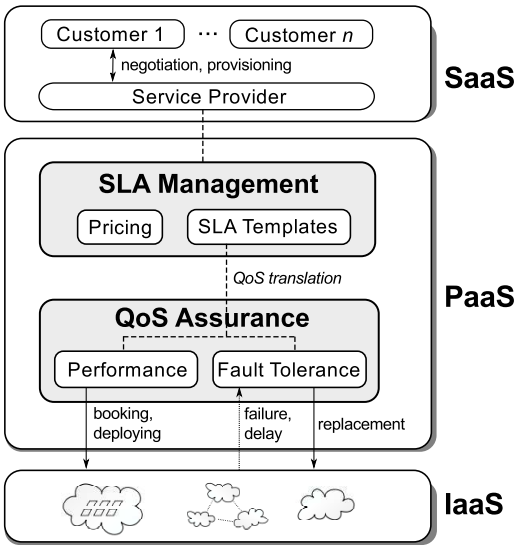

Fig. 4. Qu4DS architecture.

As case study, we implemented the *flac2ogg* service provider by using Qu4DS. The flac2ogg provider compresses FLAC audio files to the OGG format. In relation to the example introduced in Section II, the flac2ogg provider represents Zencoder whose customer is Jamendo. In order to improve performance, the flac2ogg provider leverages the Master/Worker pattern to compress the FLAC file in parallel. Qu4DS supports the development and management of the flac2ogg service as Figure 5 illustrates. At development time, the developer imports the Qu4DS library which requires implementing two methods. The first method is `treatRequest(args)` which is used to call the instance when the customer sends a request. The second method is `workerResults(args)` which notifies the instance that the workers were executed. Secondly, the service developer exports a Java jar file which is used by the Qu4DS framework to deploy service instances on the infrastructure which is similar, for instance, to the way Amazon MapReduce [2] works. Lastly, Qu4DS profiles the flac2ogg provider and creates SLA templates which are used for negotiating SLAs.

With respect to service execution management at runtime, SLA negotiation is delegated to Qu4DS (arrow 1 in Figure 5) which triggers resource acquisition and service instantiation on the infrastructure (arrow 2). When a customer sends a request (arrow 3), Qu4DS forwards the request to the right service instance (arrow 4) by calling its `treatRequest(args)` method. The service instance splits the FLAC file according to its configuration (i.e., n number of workers), prepares the workers, and sends them to Qu4DS (arrow 5). Following that, Qu4DS executes the workers on the infrastructure in accordance to the agreed QoS, i.e., it reacts to job failures and delays according to replacement thresholds. When all the FLAC file parts are compressed to the OGG format, Qu4DS sends the result files to the service instance by calling the `workerResults(args)` method (arrow 6). Then the instance merges the OGG files to a single OGG file and sends it to Qu4DS (arrow 7) which forwards the response to the customer (arrow 8). Finally, Qu4DS destroys the service instance when the contract ends; thus releasing the previously booked resources.

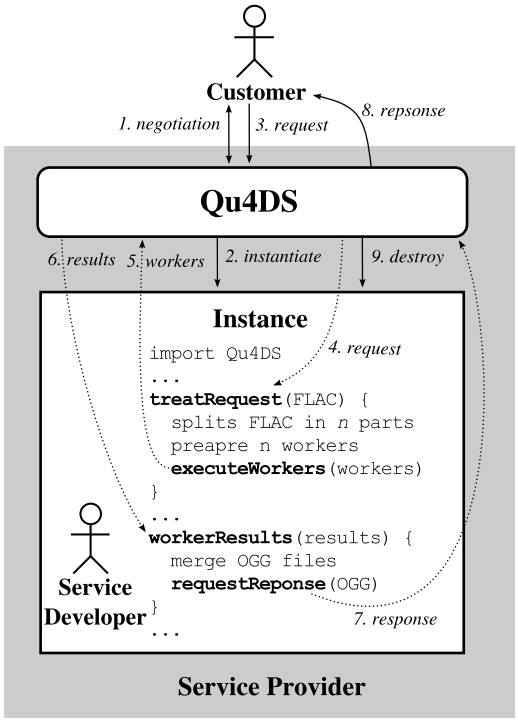

Fig. 5. Qu4DS supports service development and execution management automatically and transparently. This figure shows how Qu4DS supports the flac2ogg provider in compressing FLAC files to the OGG format.

The Table II depicts the QoS translation of QoS constraints to the configurations of QoS assurance mechanisms. The flac2ogg service provider was profiled in order to map the translation of response time to resources requirements, i.e., number of workers necessary to reach a given response time. The reliability QoS constraints were statically set with greater replacement thresholds for constraints which express stronger reliability.

VI. Evaluation

A. Scenarios

In order to validate that the proof-of-concept of the approach can effectively ensure QoS, we evaluated the efficiency of the fault-tolerance mechanism in the previously-mentioned case study. The goal is to analyze the impact on the provider

Labels \ Translation τ	Number of Workers	Replacement Thresholds $(failure, delay)$
Fast	$\tau(strong) = 4$	$\tau(weak) = (0, 0)$
Safe	$\tau(weak) = 2$	$\tau(strong) = (1, 1)$
Classic	$\tau(medium) = 3$	$\tau(medium) = (1, 0)$
Standard	$\tau(weak) = 2$	$\tau(weak) = (0, 0)$

TABLE II
THE TRANSLATION OF QOS TO SYSTEM-LEVEL CONFIGURATIONS USED BY THE FLAC2OGG SERVICE.

profit in the presence of job malfunctions for different fine values. The rate of job misbehaviors varies from zero to forty percent of the total amount of jobs. The misbehaved jobs were randomly chosen[2] where half of the given percentage was set to fail and the other half was set to be delayed. The fines were set by assuming that they can cost half of the price to treat the request ($f_\psi = 0.5$), the same price ($f_\psi = 1.0$) or two times the price to treat the request ($f_\psi = 2.0$). The experiments were performed on the Grid'5000 [4] testbed which served as the IaaS resource provider. An operating system image was customized and used to deploy Qu4DS[3]. In addition, a total of fifty-one nodes of the *paradent* cluster at the Rennes site were used.

Three scenarios were created in order to represent different customer profiles. All scenarios rely on fixed contract duration set to the experimentation time. SLA templates were used to customize the scenarios as explained next.

- *high-FT*: composed of ten customers whose contract template label is *fast*.
- *hybrid*: composed of twelve customers, three customers of each contract template label (*fast, safe, classic, standard*).
- *high-RR*: composed of ten customers whose contract template label is *safe*.

B. Results

The Figures 6, 7, and 8 depict the results of the experimental evaluation. Each dot represent a different experiment configuration whose experimentation time is nine-hundred seconds (i.e., 15 minutes). The Y-axis shows the provider profit (cf. Equation 3) in currency units while the X-axis shows the rate of misbehaved jobs in percentage. The curves represent different configurations concerning the value of the fine ψ which depends on the fine constant f_ψ (cf. Equation 2). The experiments were performed by disabling Qu4DS fault tolerance mechanism whose curves are name as *FT-off*; except for the *high-RR* scenario as it holds no SLA template which requires fault tolerance. Finally, the *loss* horizontal line represents the limit when the profit starts getting negative values, i.e., losses.

The first general observation is that the greater the misbehavior rate is, the lower the profit is. This expected behavior

[2]Only jobs used to deploy the workers were candidate to be failed or delayed.

[3]The details of this image is available on-line: https://www.grid5000.fr/mediawiki/index.php/Lenny-x64-quads

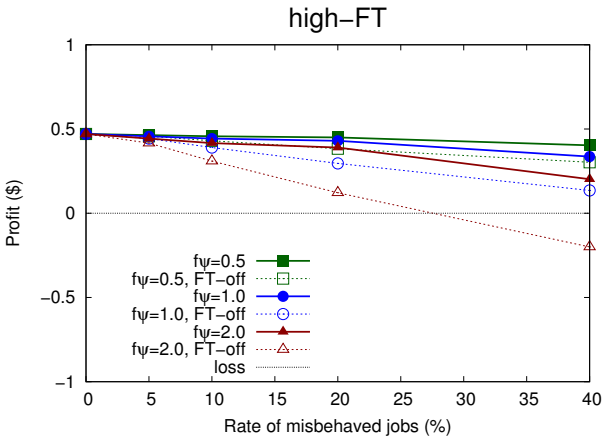

Fig. 6. Scenario *high-FT*.

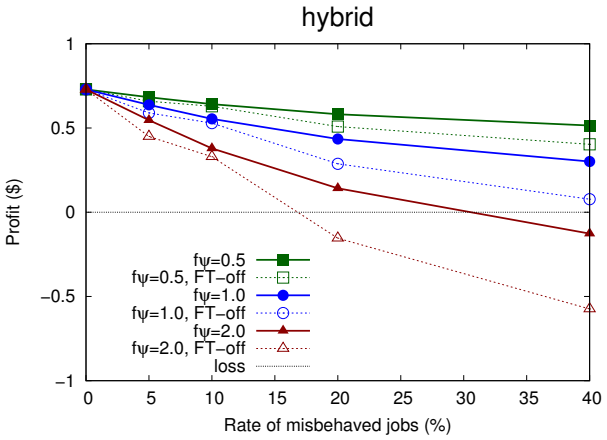

Fig. 7. Scenario *hybrid*.

happens because increasing misbehaved jobs increases the chances of implying SLA violations. Regarding the different scenarios, the more a scenario holds contract template labels which require fault tolerance capabilities, the more moderately the profit drops. For instance, the curves of the *high-RR* scenario in Figure 8 (i.e., no fault tolerance is required) decrease sharper than the other scenarios. Specifically, the *high-FT* scenario was able to keep a moderate decrease of profit curves (cf. Figure 6).

Secondly, we can observe the difference between profits with same fine value when the fault tolerance is disabled. In Scenarios *high-FT* and *hybrid* (cf. Figures 6 and 7), all the full line curves are plotted above the dashed lines (i.e., *FT-off*). This means that disabling the QoS assurance mechanism negatively influences the profit. Hence, for the experimented situation, the fault-tolerance QoS assurance mechanism showed to be effective.

Finally, the greater the fine is, the sharper the profit decreases. An interesting aspect to note is how the fault-tolerance QoS assurance mechanism smooths the profit drop for significant job misbehavior rates. For instance, even when the job

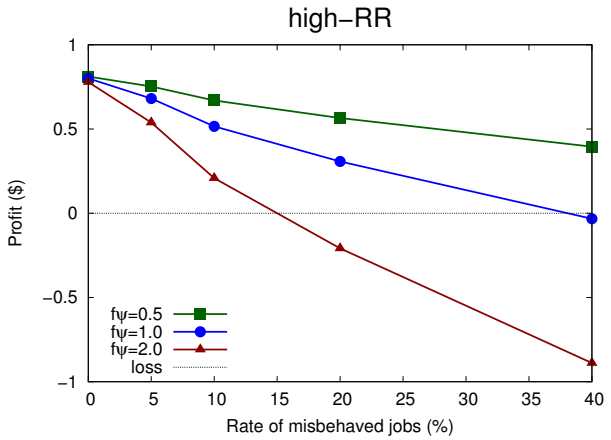

Fig. 8. Scenario *high-RR*.

misbehavior rate is 40% (cf. Figure 6), Qu4DS prevented the profit to reach negative values.

VII. Related Work

Considering work related to SLA management, the SLA@SOI project [33] proposes a hierarchical and integrated architecture for services. The SLA@SOI architecture wraps software and resources as services and separates service management from SLA management. The SOA4ALL [18] project proposes an architecture for service composition driven by quality aspects. It includes comprehensive service discovery and bindings based on semantics. However, these architectures define neither QoS assurance mechanisms nor pricing models.

In the context of SLA description and pricing, some approaches well describe SLAs and pricing models such as Amazon EC2 [1] and Zencoder [34]. Other approaches such as Macías et al. [25] and Genaud and Gossa [9] address profit concerns under a pricing model. Nevertheless, these approaches do not address QoS assurance. Moreover, these approaches fail in including currency fines into their pricing model.

With regard to SLA translation, in [17], Kotsokalis and Winkler model the translation of SLA based to service dependency properties. In [5], Chen et al. translate response time metrics to CPU requirements. In [30], Stantchev and Schröpfer use service replication in order to improve and ensure performance and fault-tolerance QoS. The authors translate QoS to infrastructure-level by setting the service replication degree. Hasselmeyer et al. [12] investigate how SLA objectives are translated to infrastructure configurations in order to enable high-performance computing service providers to meet the agreed QoS. The main drawback of these approaches is that they fail to specify QoS assurance mechanisms and to apply a billing model. Lastly, in [22], Li et al. investigate various techniques for translating SLA properties to different layers in a tier architecture. Despite the comprehensive study, the authors also consider neither pricing nor QoS assurance aspects.

With regard to QoS assurance, in [15], the Kecskemeti et al. address performance by decreasing the virtual machine deployment time. The GridWay [13] project proposes a framework that reschedules jobs in order to improve performance. In [16], Klein and Perez propose to improve performance of distributed applications by prioritizing previously submitted applications. In [23], Luckow et al. propose the SAGA big-job abstraction to acquire cloud resources, deploy virtual machines and submit grid jobs to the booked resources. SAGA big-job replaces pilot-jobs which fail to acquire resources from clouds. In [19], Lee and Zomaya propose an algorithm for job rescheduling which replaces grid resources by cloud resources if a grid resource is identified as delaying job executions. These previously mentioned approaches concern low-level details; hence they do include neither higher-level SLA aspects nor billing models.

The MapReduce Library [7] handles worker failures by periodically checking if workers are reachable. If a worker fails, their task are re-scheduled to be executed on another idle worker. Moreover, the MapReduce Library relies on a replicating scheme for the remaining jobs since latest jobs are more susceptible to fail. In order to improve performance, the MapReduce library also relies on replication of remaining jobs by statically configuring the number of map and reduce operations as well as the number of workers. Indeed, the replication of remaining jobs is a solution for improving performance and fault-tolerance qualities. However, we propose to rely on dynamic job metrics such as job elapsed time and job state in order to immediately identify malfunctioning jobs. Thus repairing actions are triggered at this very moment instead of postponing them to the end of the computation.

VIII. Conclusion

In this work, we address the problem of integrating SLA specification and enforcement for clouds. Firstly, our approach supports creating SLA templates by combining performance and fault-tolerance QoS, represented by response time and reliability QoS metrics. A billing model is also described and integrated into the SLA templates. Secondly, our approach includes the design of two QoS assurance mechanisms, which dynamically ensure response time and reliability. The response time is ensured by booking resources, configuring, and deploying the service instance according to the resource requirements. Reliability is ensured through algorithms that replace failed and delayed jobs. Finally, the approach covers how to translate QoS to appropriate configurations of the QoS assurance mechanisms.

The approach is validated by the Qu4DS framework [8] which supports the development and management of cloud services on distributed infrastructures. A case study was used to evaluate the impact of faults on the provider profit. The evaluation was carried out on the Grid'5000, which served as an IaaS provider by providing fifty-one resources to run the experiments. The results demonstrated the effectiveness of assuring fault tolerance in different situations.

The contribution proposed by this paper opens up several directions for future work. With respect to billing, a natural first step is to take into account different IaaS clouds and to enable the provider to adjust its expenses based on different resource costs. Further future work involves evaluating the performance of the QoS assurance mechanism by dealing with resource shortages. Finally, another direction is to investigate the augmentation of the cloud provider profit through under-provisioning or prioritizing more profitable contracts during resource shortages.

IX. Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE). Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, January 2006. Accessed in January 2012.

[2] Amazon Web Services LLC. Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/, April 2009. Accessed in January 2012.

[3] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Global Grid Forum, 2007.

[4] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (GRID)*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. Translating service level objectives to lower level policies for multi-tier services. *Cluster Computing*, 11:299–311, 2008. 10.1007/s10586-008-0059-6.

[6] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In D. R. Avresky, M. Diaz, A. Bode, B. Ciciani, E. Dekel, O. Akan, P. Bellavista, J. Cao, F. Dressler, D. Ferrari, M. Gerla, H. Kobayashi, S. Palazzo, S. Sahni, X. S. Shen, M. Stan, J. Xiaohua, A. Zomaya, and G. Coulson, editors, *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 57–70. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-12636-9_4.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, Jan. 2008.

[8] A. L. Freitas, N. Parlavantzas, and J.-L. Pazat. Cost Reduction Through SLA-driven Self-Management. In *Proceedings of the 9th IEEE European Conference on Web Services (ECOWS)*, September 2011.

[9] S. Genaud and J. Gossa. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pages 1–8, 2011.

[10] Google. Google Application Engine. http://code.google.com/appengine/, January 2008. Accessed in January 2012.

[11] I. Google. Google maps. http://maps.google.com/, October 2004. Accessed in January 2012.

[12] P. Hasselmeyer, B. Koller, L. Schubert, and P. Wieder. Towards SLA-Supported Resource Management. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC)*, pages 743–752. Springer, 2006.

[13] Huedo, Eduardo and Montero, Ruben S. and Llorente, Ignacio M. A framework for Adaptive Execution in Grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.

[14] I. Jamendo. Jamendo,. http://www.jamendo.com/en, January 2005. Accessed in January 2012.

[15] G. Kecskemeti, G. Terstyanszky, P. Kacsuk, and Z. Nemth. An Approach for Virtual Appliance Distribution for Service Deployment. *Future Generation Computer Systems*, 27(3):280 – 289, 2011.

[16] C. Klein and C. Perez. An RMS Architecture for Efficiently Supporting Complex-Moldable Applications. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC)*, HPCC '11, pages 211–220, Washington, DC, USA, 2011. IEEE Computer Society.

[17] C. Kotsokalis and U. Winkler. Translation of Service Level Agreements: A Generic Problem Definition. In A. Dan, F. Gittler, and F. Toumani, editors, *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, volume 6275 of *Lecture Notes in Computer Science*, pages 248–257. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16132-2_24.

[18] R. Krummenacher, J.-P. Lorre, C. Hamerling, A. Duke, M. Villa, F. Baude, E. Mathias, V. Legrand, C. Ruz, D. Liu, C. Pedrinaci, T. P. Lobo, M. Dimitrov, and P. Merle. Final SOA4All Reference Architecture Specification. Technical Report D1.4.2A, SOA4ALL – Service Oriented Architectures for All, 2010.

[19] Y. C. Lee and A. Y. Zomaya. Rescheduling for Reliable Job Completion with the Support of Clouds. *Future Generation Computer Systems*, 26:1192–1199, October 2010.

[20] S. Lehmann and P. Buxmann. Pricing Strategies of Software Vendors. *Business & Information Systems Engineering*, 1(6):452–462, 2009.

[21] S. Lehmann, T. Draisbach, P. Buxmann, and C. Koll. Pricing Models of Software as a Service Providers: Usage-Dependent Versus Usage-Independent Pricing Models. In G. Dhillon, editor, *Proceedings of the 8th Annual Conference on Information Science, Technology & Management (CISTM)*, August 2010.

[22] H. Li, W. Theilmann, and J. Happe. SLA Translation in Multi-layered Service Oriented Architectures: Status and Challenges. Technical Report IB 2009-8, University of Karlsruhe, 2009.

[23] A. Luckow, L. Lacinski, and S. Jha. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)*, pages 135–144, 2010.

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

[25] M. Macías, J. O. Fitó, and J. Guitart. Rule-based SLA management for revenue maximisation in Cloud Computing Markets. In *Proceedings of the 6th IEEE/IFIP International Conference on Network and Service Management (CNSM)*, pages 354 –357, oct. 2010.

[26] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology, September 2011.

[27] N. Rosa, P. Freire Cunha, and G. Ribeiro Justo. An Approach for Reasoning and Refining Non-Functional Requirements. *Journal of the Brazilian Computer Society*, 10:62–84, 2004. 10.1007/BF03192354.

[28] i. Salesforce.com. SalesForce. http://www.salesforce.com/, January 2012. Accessed in January 2012.

[29] Schubert, L. Et Al. The Future Of Cloud Computing, Opportunities for European Cloud Computing Beyond 2010. Technical report, European Commission, Information Society & Media, 2010.

[30] V. Stantchev and C. Schröpfer. Negotiating and Enforcing QoS and SLAs in Grid and Cloud Computing. In *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing (GPC)*, GPC '09, pages 25–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] The FLAC project. Free Lossless Audio Codec (FLAC). http://flac.sourceforge.net/, 2011.

[32] The Xith Open Source Community. Ogg Vorbis Audio Format. http://www.vorbis.com/, October 2011.

[33] W. Theilmann, J. Happe, C. Kotsokalis, A. Edmonds, K. Kearney, and J. Lambea. A Reference Architecture for Multi-Level SLA Management. *Journal of Internet Engineering*, 4:289–298, 2010.

[34] Zencoder, Inc. Zencoder – Cloud Video Encoding/Transcoding Software as a Service. http://zencoder.com/, May 2010. Accessed in January 2012.