Cost Reduction Through SLA-driven Self-Management

André Lage Freitas, Nikos Parlavantzas, Jean-Louis Pazat

Cost Reduction Through SLA-driven Self-Management

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*—A main challenge for service providers is managing service-level agreements (SLAs) with their customers while satisfying their business objectives, such as maximizing profits. Most current systems fail to consider business objectives and thus to provide a complete SLA management solution. This work proposes an SLA-driven management solution that aims to maximize the provider's profit by reducing resource costs as well as fines owning to SLA violations. Specifically, this work proposes a framework that comprises multiple, configurable control loops and supports automatically adjusting service configurations and resource usage in order to maintain SLAs in the most cost-effective way. The framework targets services implemented on top of large-scale distributed infrastructures, such as clouds. Experimental results demonstrate its effectiveness in maintaining SLAs while reducing provider costs.**

I. Introduction

Service-based systems are built by integrating loosely-coupled services from a range of providers. To handle varying service loads, providers are increasingly taking advantage of large-scale distributed infrastructures, such as clouds and grids, which deliver remote resources in a flexible, on-demand fashion. A major challenge for service providers is managing such infrastructures in order to meet their business objectives while maintaining conformance to service-level agreements (SLAs) with customers.

A large part of the research on SLA management in service-oriented architectures (SOAs) targets composite services; that is, services composed of simpler services, and thus shielded from the details of the underlying infrastructure. SLA management in this context typically involves replacing services by more suitable ones [7], [12]. Such work does not address how basic, atomic services guarantee QoS properties, which invariably requires managing the underlying distributed infrastructure, and is the focus of this paper. A significant amount of work has focused on SLA management for large-scale distributed applications, such as e-science applications deployed on grids, or multi-tier enterprise applications deployed on clusters [6], [2], [8]. However, such work does not address meeting the business objectives of service providers, such as maximizing profit.

This paper proposes a generic framework to assist service providers in honoring SLAs while reducing the costs of infrastructure usage. The proposed framework integrates a rich set of QoS management mechanisms supporting the complete SLA life-cycle, from SLA template creation to service termination. To manage infrastructure usage, the framework builds on a simple interface, compatible with modern grid and IaaS cloud APIs. To accommodate fluctuating service loads and unpredictable failures, the framework includes flexible support for self-adaptation in the form of multiple interacting control loops. Importantly, the control loops build on replaceable adaptation strategies, which can be combined in multiple ways, thus extending the applicability of the framework.

The rest of the paper is structured as follows. Section II introduces the Qu4DS (Quality Assurance for Distributed Services) framework while its adaptation strategies are exposed in Section III. Section IV discusses aspects of service provisioning, how Qu4DS profiles service providers and the assumptions on which it relies. Next, the Section V explains implementation details along with the service provider flac2ogg. Section VI discusses about the evaluation, its environment and the results. Ultimately, related works are commented in Section VII followed by the conclusion in Section VIII.

II. Qu4DS

A. Architecture

Qu4DS main goal is to provide SLA self-management that minimizes service provider's costs. Costs mean the payment of fines due to SLA violations, i.e. requests abortions, and the price for using the infrastructure. In order to decrease the costs on resource acquisition, Qu4DS shares the pool of booked resources among distinct contracts. With respect to minimizing fine payments, Qu4DS chooses the more suitable request to be aborted to handle the lack of resources and it manages the execution of treating requests by ensuring the agreed QoS. In order to deal with the environment dynamism, Qu4DS takes advantage of self-adaptation mechanisms based on strategies that reacts to certain events at runtime. Additionally, even though these features help on improving the service reputation, their most important advantage is the impact on increasing the service provider profit.

The Qu4DS architecture is described by Figure 1. Based on the cloud architecture layers, Qu4DS places itself in the PaaS (Platform as a Service) layer by using resources from an IaaS (Infrastructure as a Srevice) provider in order to provide a support to the upper SaaS (Service as a Service) layer. In the SaaS layer, customers contact the Web Service (WS) in order

to establish a contract. The contract proposal is forwarded to the *SLA Negotiator* that asks the *QoS Translator* to translate its QoS to the resource configuration able to ensure such QoS. The SLA Negotiator then checks through the *infrastructure management* interface whether the resource requirements can be met. If so, the SLA Negotiator configures and deploys the service instance on the infrastructure through the *job management* interface and commits the contract agreement to the right customer which is now able to send requests.

When a customer sends a request, the SLA Negotiator asks the *request arrivals* control loop whether it can be treated. If so, the SLA Negotiator forwards the request to the right service instance deployed on the infrastructure. The service instance prepares the distributed tasks necessary to treat the request based on its configuration and asks Qu4DS to execute the tasks. These tasks are also deployed by Qu4DS on the infrastructure through the job management interface and monitored by the *job faults* and *job delays* control loops. If their executions are successful, Qu4DS answers the service instance the result of the tasks which are used to finish the request treatment. As follows, the instance tells Qu4DS that the request is treated and it forwards the result to the right customer. If any of the control loops informs that the request could not be treated, the SLA Negotiator aborts the request, tells the customer about such SLA violation and computes the penalties.

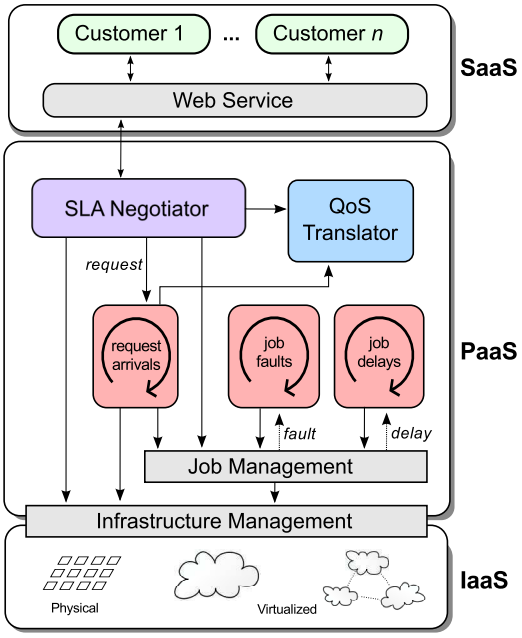

Fig. 1. Qu4DS Architecture.

B. Interfaces

Qu4DS interacts with the underlying infrastructure through the infrastructure management interface. As this interface abstracts over operating systems images, it can be easily implemented on top of existing IaaS cloud APIs, which provide operations to create and manage virtual machine (VM) instances. The infrastructure management interface is as follows:

- `List<InfraResourceType> getResourceTypes()` Gets the list of resource types which includes their classes (e.g.: A, B, C), CPU clock, RAM memory, price per hour and so forth.
- `int getNumberOfAvailableResources(String resourceClass)` Gets the number of available resources that can be booked based on the resource class.
- `int reserve(int nOfResources, String resourceClass, String startTime, String endTime)` Reserves resources according to their parameters and the time. If `startTime` is *now*, it represents a resource booking against a reservation (late booking). Finally it returns either the reservation ID or -1 if it could not meet the required resources.
- `boolean resize(String resourceClass, int newNumberOfResources, String endTime, List<Integer> resourcesId)` Allows to resize the pool of reserved resources, i.e., the quantity of reserved resources, their class as well as the end provision time. If the resize operation consists of reducing the number of resources, `resourcesId` may be used to define which resources will be discarded.
- `List<InfraResource> getReservedResources(int reservationId)` Gets the reserved resources given a reservation ID.

Qu4DS is then allocated at the PaaS layer, however it is still useful to define a higher-level interface to let it manage service instances on top of the bare IaaS resources. Thereby we define a *job management* interface based on SAGA (Simple Grid API) [5]. This interface leverages the job abstraction and defines the operations over them as well as callbacks for monitoring purpose. The abstraction level, the operations and the job life-cycle (Cf. Figure 2) were extended from the SAGA Task Model. Moreover, we assume that the VM images deployed on the IaaS layer include an implementation of the job management interface. The job management interface exposes the following operations:

- `InfraJob createJob(InfraJobDescription jobDescription)` Creates a job based on a description whose only mandatory attribute is the binary file to be executed.
- `boolean runJob(InfraJob job, String resourceAddress)` Launches an already created job (`NEW` state) on a specific resource. If the `resourceAddress` is null, it chooses the resource based on a previously configured scheduling policy.
- `boolean cancelJob(InfraJob job)` Cancels the execution of a `RUNNING`, `MIGRATING` or `SUSPENDED` job.
- `boolean cancelAllJobsOnResource (InfraResource resource)` Cancels all jobs on the specified resource.
- `int checkpointJob(InfraJob job)` Saves the current execution flow of a `RUNNING` job without suspending it.
- `boolean suspendJob(InfraJob job)` Suspends the execution of a `RUNNING` job.
- `boolean resumeJob(InfraJob job)` Continues the execution of a `SUSPENDED` job.
- `boolean resumeJob(InfraJob job, int checkpointVersion)` Continues the execution of a `SUSPENDED` job based on a

specific checkpoint.

- `boolean migrateJob(InfraJob job, String resourceAddress)`
  Migrates the job to the specific resource.
- `List<InfraJob> getAllJobs(int reservationId)`
  Gets all submitted jobs within the current set of reserved resources.
- `List<InfraJob> getJobs(String jobState)`
  Gets all submitted jobs within the current set of reserved resources whose state is jobState.
- `List<InfraJob> getJobsOnResource(String resourceAddress)`
  Gets all the jobs submitted to the specific resource.
- `void registerCallback(InfraJob job, String metric, boolean on, Observer observer)`
  Enables or disables event notification of job metrics, e.g., state, elapsed time, resource on it runs, utilization of CPU and memory and so forth.

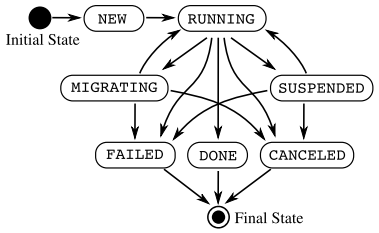

Fig. 2. The job life-cycle extended from the SAGA Task life-cycle.

III. Self-Adaptation

The employment of adaptation mechanisms depends on the Qu4DS request life-cycle which is described by Figure 3. When a customer sends a request, a request is created containing information about its SLA and further dynamic information as the input data. Its state is automatically set to `NEW` and Qu4DS decides whether it can be treated or not. If it cannot be treated, it gets an `ABORTED` state. Else, the request state is set to `TREATING` and keeps that state untill the end of the request treatment. If the request is successfully treated, it is then considered `TREATED`, otherwise it is `ABORTED`. Additionally, during the request treatment, Qu4DS registers information about request abortions in order to compute the penalties subsequently.

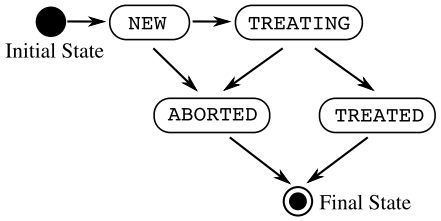

Fig. 3. Qu4DS request life-cycle.

Qu4DS takes advantage of Autonomic Computing [9] by implementing three MAPE (Monitor, Analyze, Plan, Execute) control loops as exposed by Figure 1. The first control loop is driven by *request arrivals* events and is then applicable to `NEW` requests. It is in charge of checking whether this just-arrived request can be treated or not. The request arrivals control loop asks the QoS Translator the request resource requirements and then checks their availability with the infrastructure. If the resource requirements can not be met, the control loop decides if it will abort this request or another request that is being treated and is able to free enough resources to fit the resource requirements.

The other control loops ensure the proper execution of requests and operate over `TREATING` requests. Indeed, these control loops act as self-healing mechanisms in order to prevent SLA violations. While the *job faults* control loop reacts to job faults events by providing reliable request treatment, the *job delays* control loop reacts to job delays and ensures performance aspects of the request treatment.

In addition, the control loops take decisions based on the adaptation strategy to which each control loop is configured. The adaptation strategies are thus driven by high-level guidelines which are summarized as minimizing the service provider cost on fine payments. Thereby, we have designed the following adaptation strategies described in Table I.

Request Arrival Strategies	
Name	**Description**
VFC (Violation based on Fine Cost)	Aborts the request whose fine is the cheapest.
VBP (Violation Based on Priority)	Aborts the request with lowest priority.
RVC (Random Violation Criterion)	Random choice of request to be aborted.
Job Fault Strategies	
Name	**Description**
FJV (Faulty Job implies Violation)	Aborts the request immediately.
SRFJ (Single Re-Submission of Faulty Jobs)	Replaces the fault job once. If the same job is faulty again, aborts the request.
Job Delay Strategies	
Name	**Description**
LJV (Late Job implies Violation)	Aborts the request immediately.
SRLJ (Single Replacement of Late Jobs)	Replaces the delayed job once. If the same job gets late again, aborts the request.

TABLE I
Qu4DS adaptation strategies are triggered by the arrival of new requests, the occurrence of job faults and delays.

IV. Cost-Reduced Service Provisioning

A. Service-Level Agreement

We compiled a basic SLA scheme based on common aspects of current SLA specifications [1], [10] as followed exposed. The SLA templates are thus created by filling the values of $priority, th, x$.

- Parties
  - Service Customer
  - Service Provider
- Terms

  - Description: *the service functional requirements*
  - Duration: t time unit
  - Customer's obligations: are not allowed to exceed the maximum request frequency ϕ_{max}, i.e., the maximum number of requests per t.
  - Provider's obligations: should provide the agreed service and ensure its QoS.
  - Price model: pay-per-use based on the SLA type and varies according to the priority. While $p_{low} = 0$, it is added $x \in$ for each priority level it increases.
  - SLA type: [silver | gold | platinum]
  - Priority: [low | normal | high]
  - Price: $p_{type} + p_{priority}$
- QoS
  - Throughput (*amount of data* / sec): th
  - Priority level: $priority$

B. Profiling

Qu4DS relies on profiling the service provider to collect information to fill the QoS table managed by the QoS Translator and calculate further data used by Qu4DS. The QoS table values are then used to create the SLA templates. The collected information includes the proper response time threshold ($rp_{threshold}$) to process a given amount of data d, the job execution time threshold ($j_{threshold}$) and the calculation of both standard deviations rp_{st_dv} and j_{st_dv} respectively. The first metrics calculated by the QoS Translator is the job execution time threshold $j_{threshold}$ which is defined in the following equation where e_1 and e_2 are adjusting coefficients:

$$j_{threshold} = e_1 \times j_{mean} + e_2 \times j_{st\_dv} \tag{1}$$

Then it calculates the request response time $rp_{threshold}$. It represents the response time threshold that should be ensured to not imply an SLA violation since if the request elapsed time r_{etime} is greater than $rp_{threshold}$, then the request is aborted. Next, $rp_{threshold}$ is defined where $ad_{overhead}$ is a fixed time in seconds to consider the overhead due to adaptation actions during request treatment.

$$rp_{threshold} = rp_{mean} + rp_{st\_dv} + ad_{overhead} + j_{threshold} \tag{2}$$

As follows, it sets the adaptation threshold $ad_{threshold}$ which defines if there is still enough time to trigger an adaptation action:

$$ad_{threshold} = rp_{mean} + rp_{st\_dv} \tag{3}$$

Ultimately, the QoS Translator calculates the throughput th in the following equation where d is the size of the data used to profile:

$$th = \frac{d}{rp_{threshold}} \tag{4}$$

C. Assumptions

Service provisioning depends on how the service provider business model which includes defining the SLA prices and fines. In order to model such requirements, let us rely on the following assumptions and equations that define details of the service provisioning.

Assumption 1: The service provider's profit is given by the difference between its total revenue and total costs on fine payments and infrastructure utilization.

Based on Assumption 1, the Equation 5 defines the service provider's profit P_t given an operational time t. The $\sum_{i=0}^{n_t} p_i$ is the sum of provider's revenue from all agreed contracts where n is the total number of contracts and p_i is the total revenue of a contract in such a way that $p_i = p_{type} + p_{priority}$, where p_{type} is the price of a contract whose SLA type is $type$ and $p_{priority}$ is the price of the priority chosen by p_i. Let us define the set F_t as the set of all fines during t where $f_k \in F$, hence $\sum_{k=1}^{|F_t|} f_k$ means the sum of all the fines the service provider has to pay during t. Last, $c_t \times a_t$ is the total cost for all booked resources during t where c_t is the price for using a single resource during t and a_t is the total amount of booked resources.

$$P_t = \sum_{i=0}^{n_t} p_i - \sum_{k=1}^{|F_t|} f_k - c_t \times a_t \tag{5}$$

Assumption 2: If all requests are violated, the service provider will make zero profit.

Let us define the Equation 6 which calculates the maximum frequency a customer can reach by means of number of requests during the contract duration t for contract i :

$$\phi_{max_i} = \frac{t}{rp_{i_{threshold}}} \tag{6}$$

Based on Assumption 2 and supposing that customers perform the maximum feasible number of requests (ϕ_{max_i}) per t, we deduce from Equation 5 the fine price of the contract i where r_i is the number of resources required by i:

$$f_i = \frac{p_i - c_t \times r_i}{\phi_{max_i}} \tag{7}$$

Assumption 3: The service provider wants to share the resources distinct contracts in order to save costs on resources acquisition.

Thus let us assume that the service instances are deployed on dedicated resources and that the execution of their distributed tasks will share further resources. We then reduced $g\%$ from the total amount of shared resources. The following equation defines a as the total number of acquired resources by the service provider; where n is the total number of contracts, $type$ is the SLA type, u_{type} is the number of contracts of $type$ SLA type and w_{type} is the number of distributed tasks $type$ requires:

$$a = n + \frac{(100 - g)}{100} \times \sum_{type=silver}^{platinum} u_{type} \times w_{type} \tag{8}$$

V. Implementation and Case Study

A. Qu4DS Components

We implemented the *infrastructure management* interface based on the Grid'5000 [3] which was used as the IaaS layer. We then customized a Grid'5000 image [1] as we would customize an Amazon EC2 image, for instance. In such an image there are all required programs and libraries to execute Qu4DS including an implementation of the job management interface. When a resource is booked, it will be automatically operational and will contain the needed programs installed and configured to Qu4DS.

The implementation of the *job management* interface follows a layered design. The higher-level layer is the actual implementation of the job management interface that deals with job life-cycle state management (Cf. Figure 2). Moreover, it keeps Qu4DS informed about job metrics following the publish/subscribe pattern which maps job raw metrics (i.e., UNIX process metrics) to job higher-level metrics and job states. The middle layer is implemented by the common `InfrastructureBackend` abstract class that enables the use of distinct underlying job batch systems. In addition, if it is configured to simulate job misbehave, it will randomly choose the jobs will present faults and/or delays. Currently, two backend implementations are available: one that supports the XtreemOS grid [4] and another more generic on top of SSH (Secure Shell). The latter allows to let the job management implementation be used on any operating system that allows SSH connections to it and provides the Bash command-line interpreter.

Concerning the implementation of the *control loops*, they rely on an event-condition-action decision engine. The adaptation strategies that will govern them are loaded from Qu4DS configuration file and applied at runtime. Thereby, when an event is received, it is compared to the strategy conditions to decide whether it will trigger an adaptation or not. Moreover, while the request arrivals control loop always reacts to the arrival of new requests, the other control loops check more specific conditions. They check if there is enough time to adapt ($r_{etime} < ad_{threshold}$, Cf. Equation 3) and whether the misbehaving job is already a job replacement since replacements should not be replaced.

The negotiation and provision interface was implemented as a SOAP Web Service whose interface defines both negotiation and provision operations as follows:

- `List<String> getListOfContractTypes()` Gets the list of SLA templates.
- `String contractEstablishment(List<String> slaType)` Proposes a contract of type slaType which includes its duration and QoS. It returns the contract ID if the service provider accepts the contract, otherwise it returns null.
- `String request(List<String> args)` Sends a request together with its metadata. It returns the request results if there is no SLA violation, otherwise returns null .

[1]The details of this image can be accessed here: https://www.grid5000.fr/mediawiki/index.php/Lenny-x64-quads

B. Case Study: *flac2ogg* Service Provider

The current Qu4DS implementation addresses distributed service implementations based on the Master/Worker paradigm. The service instance represents the master deployed on the infrastructure while the distributed tasks represent its workers. Qu4DS assists the development of such services by freeing service developers from managing workers and by ensuring their proper execution conforming time constraints and re-acting to job faults. Nevertheless, Qu4DS also deploys and manages the master on the infrastructure

We have then implemented the *flac2ogg* service provider that encodes FLAC [13] audio files to OGG [14] as illustrated by Figure 4. Based on SLA templates, customers establish contracts with the service provider by defining the desired throughput th QoS (MB/secs) and the priority. The SLA Negotiator gets the translation of the QoS as the amount of resources necessary to satisfy the QoS and checks based on the number of booked resources if accepting this contract will not compromise its provision capacity. Then the SLA Negotiator configures the master to the right number of workers that can provide the required throughput and deploys it on the infrastructure. Once the master is deployed, an RMI connection is automatically configured to Qu4DS hence establishing a bi-lateral communication channel between the master and Qu4DS.

When a request sent by customer reaches the master through the SLA Negotiator, it splits the FLAC file in w number of workers, sets each worker to work on a split part and asks Qu4DS to execute the workers. Qu4DS wrappers the workers as jobs, submit them through the job management to be executed in parallel. The job metrics callbacks will be the triggers for the job faults and job delays control loops according to the adaptation strategy. When the workers finish encoding the FLAC parts, Qu4DS answers the master which merges them and calls the SLA Negotiator to let it forwards the OGG audio to the right customer.

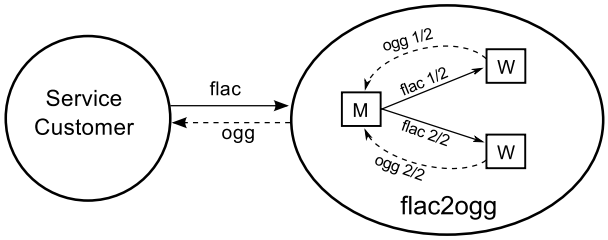

Fig. 4. The flac2ogg service provider encodes FLAC audio files to OGG based on the Master/Worker pattern. In this example, the number of worker w is set to two.

VI. Evaluation

A. Calibrating Qu4DS to *flac2ogg*

We profiled the flac2ogg service provider based on a 194MB FLAC file which means approximatively thirty minutes of recorded audio. Table II shows the QoS table together with

$rp_{i_{threshold}}$ and ϕ_{max_i} (Cf. Equations 2 and 6) where $e_1 = 1.5$ and $e_1 = 3$ (Cf. Equation 1).

SLA Type i	Throughput (MB/sec)	Resource Configuration	$rp_{i_{threshold}}$ (secs)	ϕ_{max_i} (#req/t)
Silver	1.49	2 workers	130	11
Gold	1.90	3 workers	102	15
Platinum	2.34	6 workers	83	19

TABLE II
QU4DS QOS TABLE CALIBRATED BASED ON A 194MB FLAC FILE. THE THROUGHPUT WAS CALCULATED BASED ON $rp_{i_{threshold}}$.

Figure 5 depicts the representation of customers' demand by means of a request scheduling for contract duration $t = 1800$ seconds. The beginning of the bars represents the start time of each request and the end represents $rp_{i_{threshold}}$ along the Time-axis in seconds. The Y-axis represents the customers' contract IDs whose total number is fifteen. Their priorities were whose chosen randomly where the boldness the line is, the highest priority it has. With respect to the SLA type, the first five contracts hold silver SLA type, the next five gold while platinum SLA types contracts vary from the eleventh to the fifteenth. It is important to remark that even though the end of some requests may overlap the beginning of others, it does not necessarily mean that they will be running in the same time. For instance, if a request was gracefully executed with no need to adapt, it will probably finish in time $rp_{mean} + rp_{std}v$ and not in time $rp_{i_{threshold}}$. Moreover, although the mean of requests per hour was set to 75% of ϕ_{max}, the actual request frequency ϕ_i for each customer i was got from a Poisson number generator which explains why the total number of requests of contracts that hold the same SLA type are not the same. Additionally, we reduced two requests from ϕ_{max} theoretical calculus (Cf. Equation 6) to make it feasible on empirical situations – to let them have a short waiting time between two requests and to have some spare time to shift request scheduling thus better representing a dynamic environment.

B. Scenarios and Testbed

The evaluation scenarios were defined based on the combination of the adaptation strategies as described by Table III. Further, the amount of jobs that were configured to misbehave was 10% for fault jobs and 10% for delayed jobs too knowing that a job could not be fault and late simultaneously which means that eighteen jobs were faulty as well as other eighteen were delayed. With respect to the late jobs, their never stopped aiming at letting them compromise the agreed throughput in order to force Qu4DS to adapt in case of Scenarios A. Additionally, all the contracts were established before starting the request scheduling to ensure that the time to establish the contracts would not compromise the request scheduling punctuality.

An evaluation was then performed in other to know Qu4DS effectiveness on SLA-drive self-management. The goal is to understand the benefit of employing distinct adaptation strategies given a customer demand. The experiments were performed on Grid'5000 resources which have the same characteristics: 8-core 2.5 GHz CPU, 32GB RAM memory computers interconnected through a Gigabit network connection. In order to set the number of booked resources, Qu4DS relied on Equation 8 thus setting $a = 43$ resources. Moreover, we assumed that the cost of using a resource was $c_{1800} = 0.05$ €.

Strategy	VFC	VBP	RVC
SRFJ and SRLJ	A1	A2	A3
FJV and LJV	B1	B2	B3

TABLE III
THE EVALUATION SCENARIOS WERE DEFINED BY COMBINING THE ADAPTATION STRATEGIES.

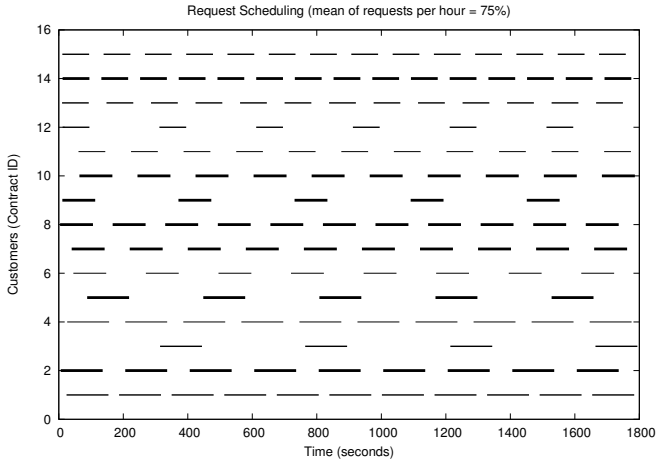

Fig. 5. The request scheduling took into account a mean of $\phi = 75\%$ of ϕ_{max}. Contract priorities were chosen randomly.

C. Results

The Figures 6 and 7 expose the results of the aforementioned evaluation scenarios. Each cluster represents the percentage of `TREATED` and `ABORTED` requests for each contract ID. Moreover, the profit of each sub-scenario is also exposed knowing that the maximum profit a scenario could touch is $P_{1800} = 15.225$, i.e., if there are no SLA violations. When analyzing the results, the first important aspect to observe is effectiveness of strategies SRFJ and SRLJ which is evident when comparing the percentage of aborted requests on Scenarios A and B as well as the profit. This was an expected behavior since FJV and LJV automatically abort requests when either a job fault or delay occurs in order to free the resources before letting the request elapsed time reach its threshold.

Interestingly, the results show the presence of a non expected behavior: the RVC strategy was more efficient than expected. It performed even better than the VBP in the Scenario A. The explanation is that all the adaptation strategies do not consider the future consequences on the request scheduling when taking decision. Indeed, it is obvious that they do not do

this since they do not rely on predicting customers' demand. Thereby, random choices of which request should be aborted might better fit to the request scheduling as it really happened in Scenario A. Actually, the way both VFC and VBP strategies act mimics a greedy algorithm which is not often suitable to achieve optimal values. However, we should consider these strategies efficient once we can observe that in Scenario B they both assured greater profit against RVC, mainly the VFC strategy that reached the greatest profit in Scenario A.

Moreover, there was not a significant difference among the profits of the same Scenario even though they clearly aborted distinct percentage of requests. It is owing to the fact that the amount of requests was not that high for the amount of booked resources along with the fact that requests were well dispersed thereby there were not rush moments on the request scheduling.

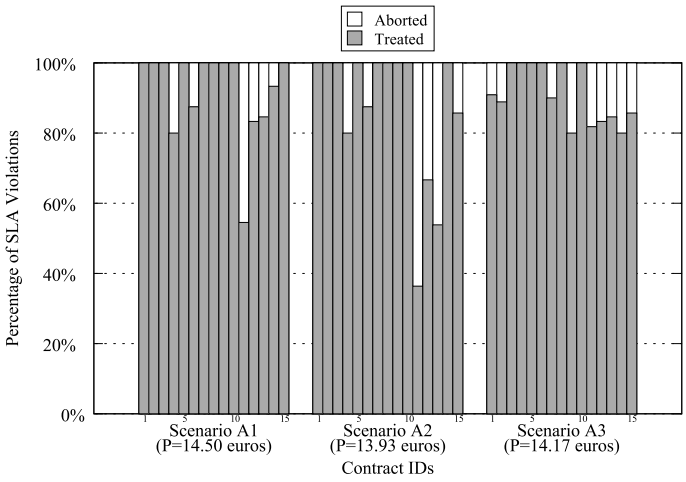

Fig. 6. Results of Scenario A: SRFJ and SRLJ activated.

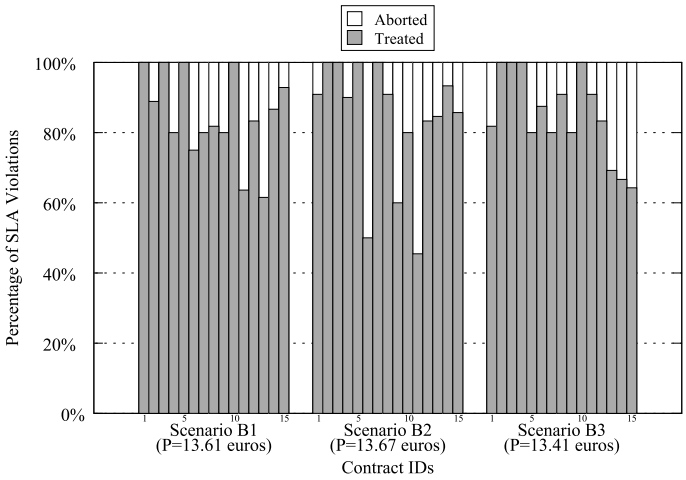

Fig. 7. Results of Scenario B: FJV and LJV activated.

VII. Related Work

With regard to the SLA management, SOC has widely investigated how services can be discovered, composed, monitored and managed in order to guarantee the proper execution of service-based applications. For instance, SLA negotiation specifications such as WS-Agreements [1] and WSLA [10] specify how services can be negotiated, how to described the Quality of Service (QoS) and provides guidelines for monitoring and auditing service behaviors. Moreover, other approaches address further specific aspects of SLA management such as appliance on resource management [6], SLA enforcement [8] and an integrated SLA management [15]. Nevertheless, in [11], the author proposes a hierarchical SLA management which enforces SLA on top of distinct adaptation policies. This approach relies on policies that adjusts the network traffic based on current QoS values and guided by high-level objectives. However, these aforementioned approaches do not specifically tackle SLA self-management with precise techniques that aim at profit increase by means of reducing costs. They commonly address QoS assurance however not considering further aspects that impact on cost reduction.

The SLA@SOI project [15] addresses a similar problem with this work, but in a different way. Specifically, SLA@SOI proposes an integrated architecture for SLA management that associates SLAs with multiple elements of the software stack at multiple layers. On the other hand, Qu4DS addresses SLA management at a single (PaaS) layer. Furthermore, the SLA@SOI project provides a set of highly generic building blocks, intended to be applicable to arbitrary deployment contexts. Qu4DS provides a complete management solution for web-service providers that build on utility infrastructures, while allowing extensibility with respect to adaptation strategies and infrastructure technologies.

VIII. Conclusions

This paper has presented a framework, Qu4DS, that facilitates SLA management for services built on distributed infrastructures, such as IaaS clouds. The framework contributes to increasing the provider profit by dynamically managing services and resources, taking into account SLA prices, fines, and infrastructure costs. The framework is modularly structured as a set of control loops configured with replaceable adaptation strategies, thus increasing its applicability to different application domains and adaptation objectives. The framework includes mechanisms for SLA negotiation and QoS translation, thus supporting in an integrated way the full SLA life-cycle, from contract negotiation to service termination. The paper has also presented detailed experimental results demonstrating that the framework can effectively increase provider profits and maintain SLA compliance in dynamic environments.

We intend to continue this work in several ways. First, we intend to add support for dynamically adjusting the number of booked resources to match current demand and to avoid over-provisioning, taking full advantage of the elastic capabilities of cloud infrastructures. Second, we intend to develop additional adaptation strategies and to evaluate them in the context of

various workload and infrastructure conditions. Supporting the automated selection of suitable adaptation strategies is a longer-term goal of this work.

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] 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.

[2] S. Benkner and G. Engelbrecht. A Generic QoS Infrastructure for Grid Web Services. *Advanced International Conference on Telecommunications / Internet and Web Applications and Services, International Conference on*, 0:141, 2006.

[3] 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 '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[4] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto, and A. Reinefeld. XtreemOS: a Vision for a Grid Operating System. Technical report, XtreemOS Consortium, May 2008.

[5] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. Saga: A simple api for grid applications - high-level application programming on the grid. *Computational Methods in Science and Technology: special issue "Grid Applications: New Challenges for Computational Methods"*, SC05:8(2), Nov. 2005.

[6] P. Hasselmeyer, B. Koller, L. Schubert, and P. Wieder. Towards SLA-Supported Resource Management. In *HPCC '06: Proceedings of the 2006 International Conference on High Performance Computing and Communications*, pages 743–752. Springer, 2006.

[7] F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 97–104, New York, NY, USA, 2008. ACM.

[8] Jose Antonio Parejo and Pablo Fernandez and Antonio Ruiz-Cortés and José María García. SLAWs: Towards a Conceptual Architecture for SLA Enforcement. In *Services, IEEE Congress on*, volume 0, pages 322–328. IEEE Computer Society, 2008.

[9] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.

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

[11] P. R. Pereira. Service Level Agreement Enforcement for Differentiated Services. In *Second International Workshop of the EURO-NGI Network of Excellence*, Villa Vigoni, Italy, July 13-15 2005.

[12] S-CUBE Consortium. Taxonomy of Adaptation Principles and Mechanisms. Deliverable 1.2.2, May 2009.

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

[14] The Xith Open Source Community. The Ogg container format. http://xiph.org/ogg/, July 2010.

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