

# A QoS Assurance Framework for Distributed Infrastructures

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

## ABSTRACT

Enforcing SLAs (Service Level Agreements) for services deployed on large-scale distributed infrastructures, such as grids and clouds, is complex owing to fluctuating customer demand and unpredictable resource availability. Current systems either address specific application domains or fail to provide a complete QoS assurance solution. This work proposes a generic framework to assist service providers in enforcing quality properties in distributed environments. The framework provides a rich set of QoS management functions, including negotiation, translation, and resource provisioning. Importantly, the framework supports dynamic adaptation; that is, it automatically modifies service behavior and resource usage in order to maintain agreed service levels while satisfying service provider-specific constraints. We have implemented an initial prototype in a grid environment and demonstrated its effectiveness in minimizing SLA violations.

## Keywords

Service-Oriented Architecture, QoS assurance, Self-Adaptation

## 1. INTRODUCTION

Service-Oriented Architectures (SOAs) promote building software systems by integrating loosely-coupled services discovered over the network [18]. The interaction between service consumers and providers is governed by service level agreements (SLAs), which constrain not only functional aspects, but also non-functional requirements such as QoS properties of the provided service. Maintaining the promised QoS properties is a major concern for service providers in order to avoid losses and penalties.

Most research on QoS assurance in SOAs targets composite services, where managing QoS typically involves changing the service composition (e.g., replacing services by more suitable ones) [21]. Such work does not address how individual, atomic services guarantee QoS properties, which

unavoidably requires controlling the underlying infrastructure. On the other hand, work that targets atomic services in the SOA context, such as [4, 19, 11], fails to address all the phases of the SLA life-cycle. This paper considers QoS assurance for atomic services, taking into account the complete SLA life-cycle. In particular, the paper concentrates on atomic services that build on large-scale distributed infrastructures, such as clusters, grids or clouds. Examples include services that expose scientific HPC (high-performance computing) applications deployed on grids and services that expose multi-tier applications deployed on clusters.

Guaranteeing QoS compliance for such services is complex and involves multiple, interrelated management activities, such as negotiating QoS properties with customers, translating SLA terms to resource requirements, allocating resources and deploying service implementations on the resources. The problem is further complicated by fluctuating service workloads and unpredictable faults, very common in large-scale environments. Accommodating this dynamism requires continuous monitoring of the operating conditions and performing corrective actions, such as re-allocating resources, migrating computational elements or re-negotiating agreements. The objective of these actions is to avoid violations of SLA terms while satisfying service provider-specific constraints, such as maximizing resource utilization. In order to address these challenges, reusable mechanisms and tools for providing and enforcing quality properties are essential. Such tools must be configurable to service-provider needs while remaining applicable to a large variety of distributed infrastructures.

This work proposes a generic framework to assist service providers in enforcing quality properties in distributed environments. The framework integrates a rich set of QoS management mechanisms, including negotiation, translation and resource provisioning. Importantly, the framework supports dynamic adaptation; that is, support for monitoring and automatically modifying service behavior and resource usage. Dynamic adaptation is not well-supported by existing approaches [19, 4, 12, 19, 25, 11] and is a major focus of this paper. To increase its applicability, the framework builds on the standard SAGA API that provides a uniform and consistent interface to the most commonly used distributed functionality. A prototype of the framework has been developed; this prototype builds on the XtremOS [7] grid and leverages the Dynaco [6] adaptation model which is based on the MAPE Autonomic Computing control loop [13].

The remainder of this paper is organized as follows. Section 2 discusses the gap in the QoS life-cycle. Following

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MONA '10, 01-DEC-2010, Ayia Napa, Cyprus  
Copyright 2010 ACM 978-1-4503-0422-1/10/12 ...\$10.00.

that, the Section 3 explains the foundations of our proposal. In Section 4, we introduce the QU4DS framework, use cases and details about its current implementation. The flac2ogg service provider is exposed as a case study which is followed by an evaluation in Section 5. Related work is summarized in a table and further discussed in Section 6. Finally, Section 7 concludes the work and discusses future research directions.

## 2. THE GAP IN THE QoS LIFE-CYCLE

The Service-Oriented Architecture [18] relies on the service abstraction to support building loosely-coupled, distributed applications. Relationships between service customers and providers are defined through electronic contracts, called Service-Level Agreements (SLAs). SLAs define the expected quality of service as well as rules that apply if these expectations are violated [5]. Ensuring the quality aspects of SLAs and thus avoiding SLA violations is important to service providers in order to avoid penalties, reduce costs, and improve their reputation.

QoS assurance is addressed by the SLA life-cycle which comprises three phases [20]:

1. *Description*. Defines the qualities, how they will be measured, and the penalties that should be employed in case of violations.
2. *Negotiation*. Covers the interaction between the parties in order to agree on the contract terms.
3. *Assurance*. Deals with enforcing the contract, that is, guaranteeing that the agreed terms will not be violated.

Describing, negotiating and assuring QoS have generally been examined from two distinct points of views. On the one hand, QoS description and negotiation have been widely investigated by means of *which are the service qualities that service providers should expose in order to enrich their business model*. The WSLA [16] and WS-Agreements [2] specifications are widely used to describe how to describe and negotiate QoS. On the other hand, QoS ensuring mechanisms at runtime have often been investigated on clusters, grids and clouds environments by means of *which are the qualities that the resource infrastructures are able to provide*. Basically, such approaches guarantee that resource provision will be delivered according to predefined QoS as, for instance, exposed by [10]. Interestingly, both points of view have investigated QoS by only considering their own needs, visions and scopes and not thinking of the problem of describing, negotiating and assuring QoS as the same problem. While the QoS description and negotiation phases delegate QoS assurance to infrastructure layers, the latter limit themselves to understanding resource-level QoS, which is not straightforwardly translated to higher-level QoS. Even though both points of view have adopted common standards such as Web Services (WS) [24], they still limit themselves to their specific problems. Therefore, there is a gap between *describing and negotiating QoS* and the *actual underlying mechanisms which ensure them*.

## 3. BASIC QoS ASSURANCE CAPABILITIES

We deal with the QoS life-cycle gap (see Section 2) based on the following three core capabilities: a uniform infrastructure interface, QoS translation, and self-adaptation. The following paragraphs describe these three capabilities, which are then put together in Section 4.

### 3.1 A Uniform Infrastructure Usage

Developing QoS assurance mechanisms relies heavily on the interfaces and constraints exposed by the underlying distributed infrastructure. Cloud infrastructures offer a simple interface [3, 14], composed basically of the `create()`, `terminate()` and `status()` operations for managing virtual machines. On the contrary, infrastructures such as grids and clusters offer complex interfaces based, for instance, on batch jobs and MPI (Message Passing Interface). Despite the benefits of relying on simple interfaces, the simpler the infrastructure usage is, the more limited the capability of providing QoS assurance mechanisms is. Moreover, simpler interfaces make it harder to take full advantage of existing infrastructures. There is clearly a trade-off between the simplicity and abstraction provided by infrastructure interfaces and their level of support for developing QoS assurance mechanisms.

An interesting solution to the aforementioned trade-off is the Simple Grid API (SAGA) [9, 8], an open standard maintained by the Open Grid Forum. SAGA proposes a high-level API that simplifies grid usage and promotes grid interoperability. The API is designed to be extensible, thus increasing its applicability to different contexts. To take advantage of existing distributed infrastructures while easing their usage, we propose a uniform and simple view of the distributed infrastructure based on extending a SAGA subset. Specifically, our interface is based on the SAGA job abstraction and associated life-cycle<sup>1</sup> and includes the most important job-related operations enhanced with price accounting. The details of each operation are as follows:

**create(jobDescription)** Creates a job based on a description, whose only mandatory attribute is the binary file. The description might optionally include job resource requirements.

**run()** Launches the job. The resource on which the job will run depends on the infrastructure scheduling policies and on the resource requirements, if the latter are provided.

**cancel()** Cancels the job execution.

**checkpoint()** Saves the current job execution state without suspending it.

**suspend()** Suspends the job execution.

**resume()** Continues the execution of a suspended job.

**migrate()** Migrates the execution of a previously checkpointed and resumed job to another resource. The resource to which the job will be migrated will be chosen by the infrastructure according to its scheduling policy.

<sup>1</sup>The SAGA job life-cycle has the following states: NEW, RUNNING, SUSPENDED, DONE, CANCELED, FAILED [8].

**registerCallback(metric, callback)** Subscribes to events about the current value of a job metric.

Finally, the use of the underlying infrastructure should be accounted for and linked to prices based on an appropriate pricing model, such as subscriptions, pay-per-use, or auctions. Linking prices to infrastructure-provided services is useful for using clouds as the underlying infrastructure, but this aspect is outside the scope of this paper.

### 3.2 QoS Translation

Executing services on top of the earlier-defined infrastructure requires mapping service requirements (e.g., response time constraints) to abstractions understood by the infrastructure (e.g., number of resources). This is not trivial, even if considering high-level infrastructure interfaces [11]. Indeed, the QoS requirements must be *interpreted, understood* and finally *translated* to lower-level resource requirements which are oriented to infrastructure usage. QoS translation can rely on analytical models, application profiling techniques or service implementation details. Moreover, it should be possible to interpret resource configurations in order to determine which QoS the infrastructure is able to provide. The efficiency and accuracy of the QoS translation depends on the available knowledge of service behavior and implementation. For instance, consider a throughput QoS, defined as the capability of treating 50 requests per second. This requirement could be simply translated to a 'high-throughput' resource requirement or to the more accurate requirement of '64 resources, 3GHz CPU, 16GB memory, interconnected by a high-speed network', depending on the knowledge of service implementation details.

### 3.3 Self-Adaptation

Although translating QoS to resource configurations is necessary for conceiving QoS assurance mechanisms, it is insufficient for providing QoS guarantees. The mechanisms should take into account the dynamism inherent in the underlying infrastructure, the unpredictability of service demand, as well as possible SLA re-negotiations. To deal with this dynamism and unpredictability, the mechanisms should support *self-adaptation* [17, 1], that is, they should support adapting the service implementation dynamically and autonomously in response to changes in underlying resources and measured QoS.

To facilitate implementing self-adaptation, this work uses the Dynaco adaptation framework [6], which separates the adaptation behavior from functional interests and decomposes it into distinct adaptation concerns. Dynaco is based on the MAPE (Monitor, Analysis, Planning, Execution) autonomic model [13], as explained in the following.

**Monitor** Sensors gather information about specific metrics (e.g., completion time) and send it to a monitoring system, which informs subscribed entities using events.

**Analysis** Based on received events, the analysis decides whether any action should be taken. If so, it creates a strategy containing the adaptation goal to be performed. In Dynaco, the analysis phase is implemented by a generic decision-making engine which is driven by domain-specific policies. Policies can be defined as ECA (event-condition-action) rules of the form 'if a more powerful resource is available, migrate service

task to it' or 'if service task is late, replicate it and get the result from the first finished replica'. Importantly, these policies can be dynamically replaced, thus allowing Dynaco to handle unforeseen scenarios, not only predicted changes.

**Planning** The planning phase relies on a generic planning engine, which is driven by guides. Based on those guides, the engine generates a plan with the necessary commands to implement the chosen strategy. For example, a plan for the service migration strategy would be 'suspend(), checkpoint(), migrate(), resume()'.  
**Executor** The Executor is in charge of executing the plan, effecting the actual changes on application execution. If there is a problem during plan execution, the monitor informs the analysis entity, which decides how to react.

## 4. QU4DS: QUALITY ASSURANCE FOR DISTRIBUTED SERVICES

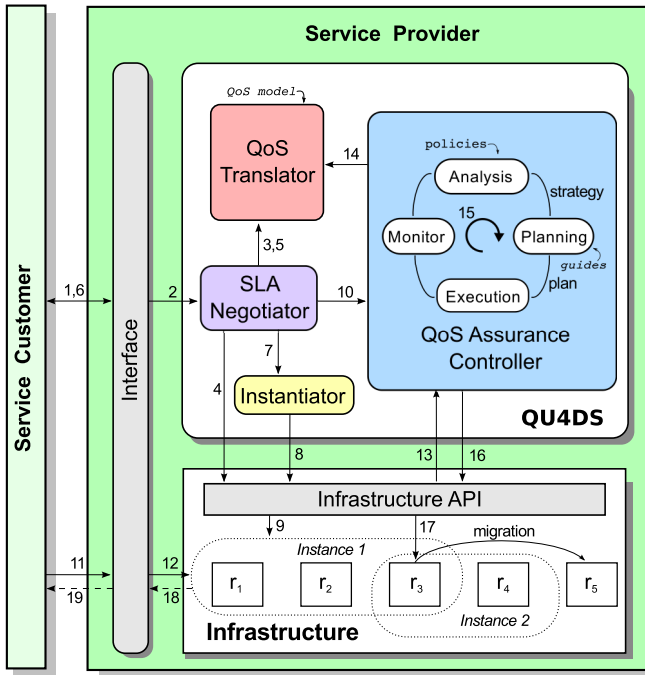
Our goal is to address the gap in the QoS life-cycle by providing mechanisms that ensure the agreed contract terms. In order to tackle this problem, we put together the aforementioned uniform way of using distributed infrastructures, the QoS translation and the support for self-adaptation (cf. Sections 3.1, 3.2, 3.3) into a flexible framework, called QU4DS (Quality Assurance for Distributed Services). Thus, we provide a solution which is able to: *(i)* negotiate QoS objectives with the customer; *(ii)* translate QoS parameters and resource configurations in a bi-directional way; *(iii)* automatically deploy the service on appropriate resources; and *(iv)* ensure the agreed QoS by reacting to underlying infrastructures changes while keeping compliant to the QoS objectives. The following sections detail the framework architecture as well as its uses cases and implementation.

### 4.1 Architecture

The QU4DS architecture is depicted by Figure 1. QU4DS aims to assist service providers in providing QoS assurance for services deployed on distributed infrastructures. In a nutshell, the service provider and customer agree on an SLA that includes QoS objectives through a negotiation process. This negotiation involves QoS translation and resource checking to verify that it is possible to satisfy the required QoS. After an agreement is established, a service instance is finally deployed. During service execution, the QoS Assurance Controller verifies periodically if the execution keeps compliant to the QoS. If the QoS Assurance Controller becomes aware of any event that may impact QoS provision (e.g., resource failure, establishments of a new agreement), it reasons about a strategy for reacting to such an event. The strategy is translated to a plan and finally executed on the infrastructure that will employ the changes.

The framework builds on a monitoring and actuation API that abstracts over heterogeneous infrastructures. QU4DS relies on a uniform infrastructure API based on SAGA as described in Section 3.1. The structure of the framework and its elements as shown in Figure 1 are explained next.

**QU4DS** It serves as the interface with service customers while coordinating the SLA management activities.



**Figure 1: The QU4DS (Quality Assurance for Distributed Services) framework is able to negotiate and ensure QoS by automatically managing the service execution on the infrastructure.**

**SLA Negotiator** Responsible for describing and negotiating SLAs with customers. It takes into account QoS objectives and their translation to resource configurations.

**QoS Translator** It converts QoS objectives to resource configurations and vice-versa. It may rely on analytical performance models or on profiling data from previous service runs.

**Service Instantiator** It discovers and allocates appropriate resources for hosting the service, and instantiates and configures the necessary service elements.

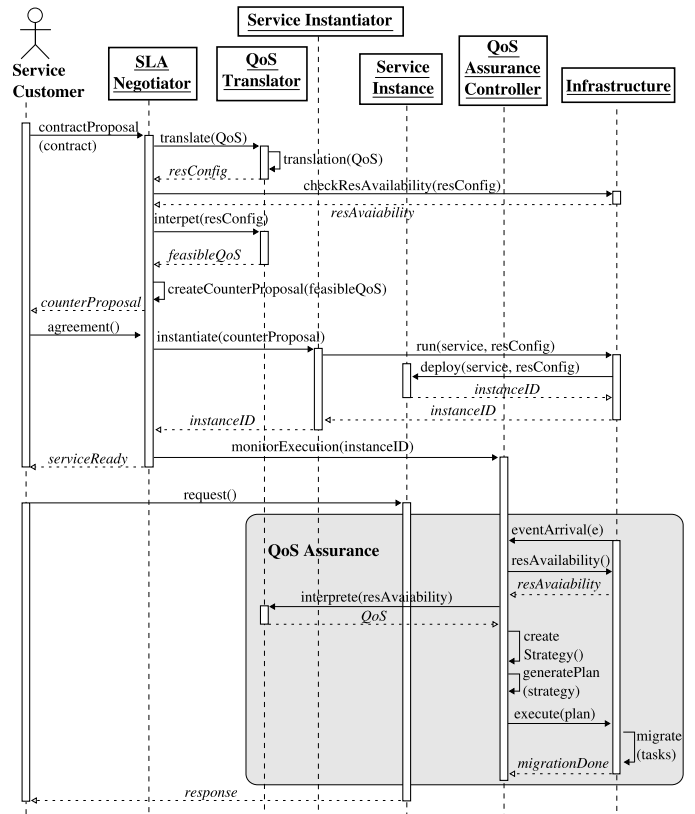
**QoS Assurance Controller** Analyses events, plans and executes appropriate actions based on both service-specific QoS objective and service-provider objectives.

## 4.2 Use Cases

Two use cases are exposed on the sequence diagram illustrated by Figure 2 and the QU4DS architecture (cf. the numbered flows on Figure 1). The first use case represents SLA negotiation and contract establishment. In this use case, the service customer proposes an initial contract to the SLA Negotiator by describing its desired QoS (1,2). The QoS Translator translates such QoS (3) to a resource configuration, whose availability and price are checked using the infrastructure API (4). As the required resources were not available, the SLA Negotiator contacts the QoS Translator (5) that interprets the current resource configuration and

its price to the possible QoS that is able to be provided. The service customer accepts the contract and establishes an SLA with the provider (6). Following that, the SLA Negotiator asks the Service Instantiator (7) to instantiate the service on the required resources taking into account the SLA (8,9) and configures the QoS Assurance Controller (10) to monitor the service execution.

The second use case represents QUADS-supported adaptation to avoid SLA violations. The service customer requires a service through the interface (11,12). While the service is being executed on a number of resources, an event is sent to the QoS Assurance Controller (13) warning about a resource availability limitation (e.g., resource overload, non-responding resource). Aiming at preventing an SLA violation, the QoS Assurance Controller searches for an alternative resource and verifies with the help of the QoS Translator that this resource can maintain the required QoS (14). The QoS Assurance Controller then creates a strategy that migrates the service tasks running on the previous resource to the alternative one (15,16,17). Finally, the service provider responds the request with no SLA violations (18,19).



**Figure 2: Sequence diagram of an SLA negotiation and service instantiation followed by a self-adaptation for ensuring the agreed QoS.**

## 4.3 Implementation

The current QU4DS implementation is written in Java and builds on SAGA, in particular, on its extension for the XtremOS grid called XOSAGA. QU4DS relies on the Apache CXF framework for Web Service (WS) support and

targets services implemented according to a Master/Worker pattern. The Service Instance treats service requests by splitting the work in  $n$  tasks and asking the QoS Assurance Controller to execute them. The QoS Assurance Controller wraps these tasks as grid jobs, manages their parallel executions as  $n$  workers on XtreamOS and informs the Service Instance when they have finished. The Service Instance then merges the task outputs and responds to the service request.

#### 4.3.1 QoS Translation and SLA Negotiation

The QoS Translator is based on log analysis profiling. The QoS parameter considered is the time to answer a request, which depends on the number of workers. Thus, QoS translation and interpretation involve determining how many workers ( $n$ ) are needed to satisfy a given response time ( $rt$ ), and, inversely, which response time  $n$  workers are able to provide. QU4DS is designed to support the WS-Agreements [2] specification which addresses both negotiation and provision interface modules by means of a language and protocol. However, the current QU4DS version only supports fix contracts represented by a WSDL binding. Furthermore, all QU4DS configuration parameters including the QoS Translator and QoS Assurance Controller parameters are set in a general configuration file called `quads.properties`.

#### 4.3.2 The QoS Assurance Controller

With respect to the QoS Assurance Controller, QU4DS implements a simplified version of Dynaco. The details of the QoS Assurance Controller elements are explained next:

**Monitor** Implements a publish-subscribe communication pattern. Any QU4DS entity can subscribe to events about a metric or a subject (i.e., a set of pre-defined metrics). Metrics can be related to jobs, resources or QoS. For example, it is possible to subscribe to events about job elapsed execution time, consumed CPU and memory as well as QoS request elapsed time. Monitoring relies on grid and QoS sensors. Grid sensors are designed to be XOSAGA customized metrics callbacks. However, due to current XOSAGA limitations, sensors are also based on UNIX scripts that monitor job metrics (CPU and memory usage, execution elapsed time, number of threads). QoS sensors are currently implemented as Java classes that feed the monitoring system with the elapsed request response time.

**Analysis** It is implemented as an ECA (event-condition-action) decision-making engine whose policies are specified as parameters loaded from the QU4DS configuration file. QU4DS currently supports an adaptation strategy called *Single Replacement for Late Jobs* (SRLJ) as summarized in Table 1. It checks if a job execution elapsed time is greater than a threshold ( $jobETime > j$ ), and if so, it cancels the job and launches a single replacement for it. The SRLJ strategy assumes that jobs can only be replaced once (i.e., replacement jobs cannot be replaced by others), and it only decides to adapt if there is enough time for that ( $requestETime < rt$ ).

**Planning** The plan is generated based on the SRLJ strategy actions. In the planning phase, each action is translated to `GridInterface` methods that use an XOSAGA backend as guides.

Policies	Conditions	Actions
$j$ : TimeThreshold	jobE- AND	if ( $jobETime > j$ )
$rt$ : respTimeThreshold	(requestETime < $rt$ )	1) create a job to replace the late job 2) cancel the late job 3) submit the job replacement

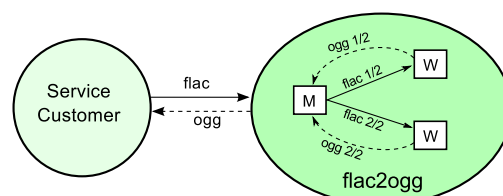
**Table 1: The SRLJ strategy implemented by QU4DS: it launches a replacement for a late job if there is enough time to adapt.**

**Executor** Ultimately, the XOSAGA backend executes the plan by calling the specified methods with the right parameters. Specifically, XOSAGA communicates with the XtreamOS broker which will finally perform the actual actions.

## 5. CASE STUDY AND EVALUATION

### 5.1 The flac2ogg Service Provider

As case study, we implemented the *flac2ogg service provider* which encodes FLAC (Free Lossless Audio Codec) [22] audio files to the lossy Ogg [23] audio format as depicted by Figure 3. Customer requests contain the audio file(s) to be encoded and then the service applies the *oggenc* encoder and returns the resulting Ogg file to the customer. This service implementation relies on a Master/Worker model. The master is responsible for receiving and treating service requests. When a request arrives, the master splits the contained FLAC file into segments, which are encoded in parallel by workers. QU4DS launches the workers, wrapped as jobs, on distinct resources and manages their execution by trying to minimize violations of the response time objective. Finally, the master merges the Ogg worker files and returns the audio encoded to the customer.



**Figure 3: The flac2ogg service provider encodes FLAC audio files to Ogg based on the Master/Worker pattern.**

The flac2ogg case study focuses on the request response time ( $rt$ ) as the QoS parameter to be assured. Therefore, the QoS Translator maps required QoS values to actual service instance configurations. The configuration parameter is the degree of parallelization, i.e., number of resources. The less the required response time is, the higher the required parallelization degree is. The QoS translator currently uses a simple mapping from QoS values to number of resources based on experimental data from previous executions.

## 5.2 Preliminary Evaluation

To evaluate the effectiveness of QU4DS in QoS assurance, we used the flac2ogg case study. The flac2ogg service provider was executed within a virtual machine configured with 2.4 GHz CPU clock and 1.5GB memory; the machine was acting as both an XtremOS core and resource roles. The evaluation was based on 30 sequential customer requests to encode a 22MB flac song. The agreed SLA specified 500 seconds as a fixed response time. Its translation resulted in splitting the file in 12 parts which were encoded in parallel.

Two experiments were performed in an environment characterized by faults as exposed by Figure 4. The first experiment analyzed the system behavior in the presence of infrastructure faults without active QoS ensuring mechanisms. The faults were represented as non-responding jobs, a common situation in grid infrastructures. In particular, seven jobs were randomly stopped<sup>2</sup> which made XtremOS and our monitoring mechanisms assume that they were still running. We expected to have seven SLA violations, but there were twelve instead, owing to an overhead side-effect. Specifically, the stopped jobs were not fully killed by XtremOS at the end of their requests as programmed in QU4DS due to an XtremOS issue. This caused an extra overhead in both XtremOS and our monitoring scripts which periodically gather information about job executions. As a consequence, there were more SLA violations than expected, mainly concentrated at the end of the request executions (Requests 20 to 30).

The second experiment analyzed the system behavior in the presence of faults but configuring QU4DS to self-adapt using the *Single Replacement for Late Jobs* adaptation strategy. As one can see in Figure 4 and in Table 2, QU4DS was able to decrease to half the SLA violations in comparison to the earlier experiment. When employing the SRLJ QoS assurance mechanism, there were only six SLA violations thanks to three QU4DS adaptation actions which were successfully employed to prevent further violations. While the Requests 4, 10 and 24 were successfully adapted, QU4DS could not avoid the violation of Request 19 which had a replacement job that got late and could not be replaced again. Finally, QU4DS also failed to react to Requests 8, 16, 27, 29 and 30. The cause is the amount of XtremOS I/O operations combined with the overhead side-effect explained earlier. During these requests, an event was sent to QU4DS to inform it of the misbehaving.

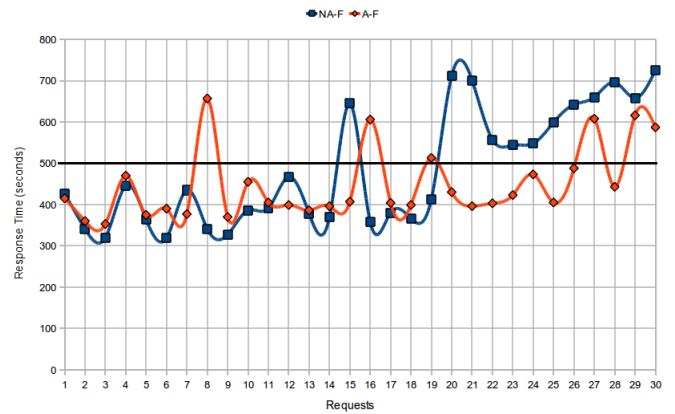
Experiment	Violated requests	# adaptations	# successful adaptations
NA-F	12 (40%)	–	–
A-F	6 (20%)	4	3

**Table 2: QU4DS prevented further SLA violations by successfully reacting three times to environment changes.**

## 6. RELATED WORK

We have summarized some related work in Table 3 by analyzing them according to the following criteria:

<sup>2</sup>By using the UNIX command `kill -STOP PID` to suspend the execution of the *oggenc* process.



**Figure 4: Response time for 30 requests for two experiments scenarios: no Adaptation with Faults (NA-F) and Adaptation employment with Faults (A-F).**

**Application Scope** Whether the targeted applications are generic or domain-specific.

**SLA/QoS Negotiation** If SLA or QoS negotiation is supported.

**Type of QoS** Which specific QoS are addressed.

**QoS Translation** If QoS requirements are translated to resource requirements.

**QoS Ensuring Mechanisms** If actual QoS ensuring mechanisms are proposed and their descriptions.

**Level of Autonomy and Adaptation Nature** If it uses self-adaptation for ensuring QoS and how much autonomous the QoS ensuring mechanisms are. Adaptation nature means whether dynamic or static adaptation strategies are employed. While static strategies can only handle predicted scenarios, dynamic strategies also address unforeseen changes.

All reviewed approaches address QoS management by dealing directly with the service infrastructure. While some of them propose effective but domain-specific approaches [26, 4, 12], others propose generic approaches with no definition of actual QoS ensuring mechanisms [19, 25, 11]. Few approaches address the whole QoS life-cycle by defining which qualities they provide and then negotiating and ensuring them. QU4DS stands out since it covers QoS description, negotiation, and automatic QoS assurance through self-adaptation in an integrated fashion.

The SLAWs [19] approach basically separates the non-functional service concern from the functional concern by wrapping it as another service. QoS aspects are thus addressed in the non-functional service, which can be customized in a decoupled way. However, SLAWs does not provide SLA enforcement mechanisms; it only proposes how they can be addressed by splitting a service in two others. In addition, it does not explicitly tackle SLA negotiation.

Hasselmeier Et Al. [11] investigate how SLA business-level objectives are translated to low-level infrastructure configurations on top of HPC providers in order to satisfy QoS.

Approach	Scope	Negotiation	Type of QoS	QoS Translation	QoS Ensuring Mechanisms	Autonomy and Nature
SLAWs [19]	Generic.	Allowed.	–	No.	Job priority management.	–
Hasselmeyer Et Al. [11]	Generic.	Yes.	–	Yes.	–	–
Benkner and Engelbrecht [4]	Domain-specific	Yes.	Response time.	Yes.	–	–
H. Zhang and Keahey [26]	Domain-specific.	Allowed.	Response time.	No.	Bandwidth limiting to prioritize high-priority clients connections	Full autonomous. Static.
C. Zhang Et Al. [25]	Generic.	Allowed.	Response time and throughput.	No.	Not specified. Assumes that clusters provided it.	Not specified. Static.
GridWay [12]	Domain-specific	No.	Performance.	No.	Re-schedule jobs if performance slows-downs and suitable resources are available.	Full autonomous. Dynamic.
QU4DS	Generic	Yes	Response time.	Yes.	The SRLJ strategy as exposed in Table 1.	Full autonomous. Dynamic.

Table 3: Comparison of QU4DS and related works based on main aspects related to QoS assurance.

They propose an architecture that considers different actors and service provider information when translating QoS to resource requirements. However, actual QoS assurance mechanisms are not provided; they assume that the infrastructure supports them by means of self-adaptation techniques. Moreover, the authors refer to autonomy as a means of translating from SLA to resource requirements, not of ensuring QoS.

The GridWay [12] project proposes a framework that self-adapts job execution on Globus. They provide a set of tools for monitoring the job and for analyzing if their performance degrades. If this happens, they re-schedule the job on more suitable resource. Unlike QU4DS, this work supports neither negotiation nor QoS translation, thus exposing low-level resource details to applications. Secondly, the way of how they employ job self-adaptation differs from QU4DS architecture proposal since QU4DS relies on existing monitoring mechanisms not being intrusive to the grid infrastructure.

## 7. CONCLUSION AND FUTURE WORK

This paper has presented a framework, QU4DS, that facilitates QoS management for services built on distributed infrastructures, such as grids and clouds. The framework has three main features. First, the framework provides flexible support for dynamic adaptation, necessary for maintaining agreed SLAs in the face of fluctuating environmental conditions. Second, the framework supports in an integrated way the complete SLA life-cycle, from contract negotiation to service termination. Finally, the framework has a modular, extensible structure, cleanly separating the different QoS management functions and service implementations. Importantly, service implementations and underlying resources are managed through a general, consistent and uniform infrastructure API, which minimizes the framework's dependence on specific platforms and increases its applicability. The paper has also presented initial experimental results that provide evidence that QU4DS can effectively reduce SLA violations in dynamic environments.

There are two main directions for future work. First, we intend to expand the set of supported QoS properties, QoS objectives, and adaptation strategies in order to evaluate more thoroughly the extensibility and usability of the framework. The next version will include support for WS-based SLA negotiation, allowing the framework to be validated in a real-world WS environment. Second, we intend to investigate the required mechanisms to allow service

providers to integrate resources on-demand, thus allowing them to take advantage of the elasticity of cloud infrastructures and pay-per-use pricing models. In this context, work such as [15] may be used to smoothly couple SAGA and clouds.

## 8. ACKNOWLEDGMENT

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).

## 9. REFERENCES

- [1] M. Aksit and Z. Choukair. Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 84–89, Washington, DC, USA, May 2003.
- [2] 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.
- [3] M.-E. Bégin. An EGEE Comparative Study: Grids and Clouds - Evolution or Revolution. Technical report, CERN - Engineering and Equipment Data Management Service, June 2008.
- [4] 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.
- [5] P. Bianco, G. A. Lewis, and P. Merson. Service Level Agreements in Service-Oriented Architecture Environments. Technical Report CMU/SEI-2008-TN-021, Software Engineering Institute of The Carnegie Mellon University, <http://www.sei.cmu.edu/reports/08tn021.pdf>, 2008.
- [6] J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for Grid computing. In *EGC '05: Proceedings of The European Grid Conference*, pages 538–547, Amsterdam, June 2005.
- [7] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto,

- and A. Reinefeld. XtreamOS: a Vision for a Grid Operating System. Technical report, XtreamOS Consortium, May 2008.
- [8] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. A Simple API for Grid Applications (SAGA). Global Grid Forum, January 2008.
- [9] 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.
- [10] R. A.-A. Gregor, G. V. Laszewski, K. Amin, M. Hategan, O. Rana, D. Walker, and N. Zaluzec. QoS Support for High-Performance Scientific Grid Applications. In *CCGRID'10: 4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 134–143, 2004.
- [11] 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.
- [12] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.
- [13] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.
- [14] T. V. Lillard, C. P. Garrison, C. A. Schiller, and J. Steele. *The Future of Cloud Computing*, pages 319–339. Elsevier, 2010.
- [15] A. Luckow, L. Lacinski, and S. Jha. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems. In *CCGRID'10: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 135–144, 2010.
- [16] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. Web Service Level Agreement (WSLA) Language Specification. Technical report, IBM, 2003.
- [17] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, 2004.
- [18] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing, Introduction. *Commun. ACM*, 46(10):24–28, 2003.
- [19] J. A. Parejo, P. Fernandez, A. Ruiz-Cortés, and J. M. García. Slaws: Towards a conceptual architecture for sla enforcement. In *Services, IEEE Congress on*, volume 0, pages 322–328. IEEE Computer Society, 2008.
- [20] S-CUBE Consortium. Survey of Quality Related Aspects Relevant for Service-based Applications. Deliverable 1.3.1, July 2008.
- [21] S-CUBE Consortium. Taxonomy of Adaptation Principles and Mechanisms. Deliverable 1.2.2, May 2009.
- [22] The FLAC project. Free Lossless Audio Codec (FLAC). <http://flac.sourceforge.net/>, 2010.
- [23] The Xith Open Source Community. The Ogg container format. <http://xiph.org/ogg/>, July 2010.
- [24] W3C Working Group. Web Services Architecture. <http://www.w3.org/TR/ws-arch/>, 2010.
- [25] C. Zhang, R. N. Chang, C.-s. Perng, E. So, C. Tang, and T. Tao. An Optimal Capacity Planning Algorithm for Provisioning Cluster-Based Failure-Resilient Composite Services. In *SCC '09: Proceedings of the 2009 IEEE International Conference on Services Computing*, pages 112–119, Washington, DC, USA, 2009. IEEE Computer Society.
- [26] H. Zhang, K. Keahey, and W. Allcock. Providing Data Transfer with QoS as Agreement-Based Service. In *SCC '04: Proceedings of the 2004 IEEE International Conference on Services Computing*, pages 344–353, Washington, DC, USA, 2004. IEEE Computer Society.