Autonomous Service Execution Driven by Service-Level Agreements

André Lage Freitas

Thèse

THÈSE INSA Rennes
sous le sceau de l'Université Européenne de Bretagne
pour obtenir le grade de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Informatique

présentée par
André Lage Freitas
ÉCOLE DOCTORALE : MATISSE
LABORATOIRE : IRISA – UMR6074

Autonomous Service Execution Driven by Service-Level Agreements

Thèse soutenue le 29 mars 2012
devant le jury composé de :

Salima BENBERNOU
Professeur à l'Université Paris Descartes / *Présidente*
Zsolt NÉMETH
Chercheur Senior à MTA SZTAKI / *Rapporteur*
Christian PEREZ
Directeur de Recherche HDR à l'INRIA / *Rapporteur*
Jean-Louis ROCH
Maître de conférence à l'ENSIMAG / *Examinateur*
Jean-Louis PAZAT
Professeur à l'INSA de Rennes / *Directeur de thèse*
Nikolaos PARLAVANTZAS
Maître de conférence à l'INSA de Rennes / *Co-encadrant de thèse*

Contents

Introduction

On Leveraging Services

Currently, computing systems are built based on the integration of other systems in contrast to earlier systems which were partially integrated or even isolated. This integration is motivated by the Internet along with its high-speed backbones which promote the connection of different computing platforms. As a result, current applications are often built on top of electronic *services* transparent to users. In order to build service-based applications, new paradigms have been proposed such as Internet of Services, Cloud Computing and Web 2.0 which enable building high-level applications by using the Internet infrastructure. Examples include Google Maps [Goo04] and Facebook [Fac04] which bring together different kind of services through a common graphical user interface. These platforms include a great number of functionalities whose interface abstracts the complexity of the underlying environment. For instance, Google Maps allows customers to uniformly access location services such as maps, satellite and street photos, traffic information, and route planner.

Service providers take advantage of distributed runtime infrastructures such as clouds and grids. Cloud Computing provides resources on-demand by leveraging virtualization technologies. Thus, resources can be acquired at runtime in order to deploy service instances according to the customer demand. For instance, Zencoder [Zen10] provides audio and video encoding services on top of resources provided by Amazon EC2 [Ama06a]. Grid Computing enables building services by leveraging batch jobs. By using grids, low-level details concerning resource management are transparent to service developers. For example, the Simple Grid API (SAGA) [GJK+05] integrates various grid platforms by providing a uniform grid programming interface.

The functionalities provided by clouds and grids are useful but taking advantage of them is complex and requires manual effort. Moreover, service execution should be guided by Service-Level Agreements (SLAs) which describe the Quality of Service (QoS) that should be delivered along with the service. In order to meet the agreed QoS properties of an SLA, service execution should address service negotiation, instantiation, provisioning and termination driven by SLA constraints. This problem is challenging owing to the need of: *(i)* translating QoS metrics to system configurations; *(ii)* ensuring the agreed QoS; and *(iii)* managing distributed resources. An orthogonal but complementary concern with regard to managing service execution is to take into account provider profit interests. Indeed, profit augmentation is a main concern of

service providers. However, it is hard to implement policies that aim at profit augmentation since they cover various pricing aspects such as price, costs, and fines. For instance, enforcing SLAs may require using more resources; however, profit augmentation policies may rely on reducing costs on infrastructure. Therefore, it is a challenge to manage the service execution on top of distributed resources while addressing SLA life-cycle and profit augmentation simultaneously.

The Gap

Current work has handled the problem of managing service execution guided by SLA directives in various ways. Regarding pricing aspects, current approaches that rely on pricing models do not focus on describing the penalties owing to SLA violations. For instance, some commercial approaches [Ama06a, Sal99, Zen10] compute penalties based on a limited definition of availability and pay fines by means of service credits.

Other approaches have addressed SLA management concerning SLA life-cycle, QoS assurance and SLA translation. Specification of languages and protocol [ACD+07, LKD+03] were proposed to handle the SLA life-cycle, but these efforts do not implement underlying mechanisms that manage SLAs. The SLA@SOI project [THK+10] also addresses SLA management, but it does not define specific realization mechanisms. With respect to SLA translation [LTH09], approaches either rely on abstract translation schemes [KW10, SS09, HKSW06] or do not consider the distributed nature of the runtime environment [CIL+08]. Most approaches which deal with QoS assurance focus on the underlying infrastructure and do not address pricing aspects. These approaches rely on job replication [DG08], job rescheduling [LZ10, Hue04], extra resources [DFL11, LLJ10] and priority policies [KP11b] in order to improve performance or fault tolerance.

Finally, some approaches handle resource acquisition as part of the service execution management. In this context, resource acquisition is proposed to be decoupled from resource utilization [TBB+08, CGP+10, LLJ10] as well as automatic virtual machine deployment [KTKN11]. Moreover, in [GG11], the authors investigate different resource acquisition policies and their impact on performance and infrastructure costs.

The efforts of the aforementioned approaches are incomplete in a sense that they do not address the whole SLA life-cycle, being hence isolated solutions. Moreover, they fail to integrate a pricing model and further economic interests, specifically the augmentation of the service provider profit. Furthermore, current approaches are not automated hence requiring a lot of human effort.

Objectives

The main goal of this thesis is to provide *an autonomous solution for managing service execution under SLA constraints*. In addition to guiding the service execution by SLAs, this thesis also aims at increasing the provider profit. Firstly, the autonomous service execution should be able to negotiate contracts, deploy service instantiates, treat requests, and destroy the service instance automatically. Moreover, these tasks should

be governed by SLA guidelines which include translating QoS metrics to system configuration, acquiring resources, and prevent SLA violations owing to QoS degradation. The challenge is then to deliver the service while meeting the agreed quality properties. Secondly, the problem of increasing the provider profit should be addressed in parallel, which brings even more complexity to the service execution management. This leads to investigate pricing aspects which impact the profit such as price, fines, and operational expenses. Further, the dynamic, unpredictable and distributed runtime environment on which services are executed brings complexity to this problem.

Organization of the Thesis

This manuscript is organized as follows. The context of this thesis lies on the service abstraction as a means of building distributed applications. This approach is called the Service-Centric Paradigm (SCP) and is introduced in Chapter 1. The importance of SCP relies on the loosely-coupled way of conceiving modular applications whose interactions are defined by Service-Level Agreements (SLAs). SLAs describe service obligations which include the Quality of Service (QoS) that should be associated with the service. Moreover, Chapter 2 discusses how current work addresses QoS on top of distributed infrastructures.

This thesis deals with two problems: *service execution management* and *service profit augmentation*. These problems are discussed in Chapter 3 which presents a solution that addresses the SLA life-cycle while aiming at increasing the provider profit. In order to handle service execution management, this thesis relies on three contributions. Firstly, Section 3.3 addresses how contract templates are created in order to enable contract negotiation which includes SLA translation and pricing aspects. Secondly, QoS assurance mechanisms are proposed in Section 3.4 which are configured based on the translation of QoS metrics to system-level configurations. Thirdly, Section 3.5 deals with resource acquisition and allocation by reacting to contract proposals and request arrivals. Simultaneous to the service execution management, complementary and orthogonal actions are in charge of increasing the provider profit. These actions include rescinding contracts, reduction of infrastructure costs, preventing SLA violations, and minimizing fine payments.

In order to realize the aforementioned conceived solution, the Chapter 4 depicts the design and implementation of the autonomous Qu4DS (Quality Assurance for Distributed Services) framework. Qu4DS transparently manages service executions by providing a higher-level support for services which abstracts over distributed infrastructures. Qu4DS is in charge of automatizing SLA management tasks based on high-level policies. Furthermore, Qu4DS relies on a modular design which enables employing further policies.

Qu4DS is validated in Chapters 5 and 6. In Chapter 5, the experimental environment is introduced which includes the case study and configuration details. As proof of concept, Qu4DS is used to implement the flac2ogg service provider which compress audio files. Chapter 6 depicts Qu4DS evaluation performed on top of the Grid5000

testbed. Different experimentation scenarios are introduced in Section 6.1 and followed by the results in Section 6.2. The evaluation validates the solution proposed by this thesis and allows analyzing Qu4DS efficiency under different constraints.

Part I

State of the Art

Chapter 1

Service-Centric Paradigm

The Service-Centric Paradigm (SCP) refers to the idea of building distributed applications by leveraging elementary software blocks abstracted as electronic services. Indeed, SCP mostly takes advantage of two important concepts: *computing utility* and *separation of concerns*. Computing utility dates back to the 60s when computing features were proposed to be offered as public utilities by leveraging the time-sharing technology as if they were conventional services, e.g., electricity and water. Following that, more recent research has implemented technologies which allow computing utility to be conceived in large scale such as early data centers, grid computing and cloud computing. On the other hand, dating back to the 70s, the idea of separation of concerns has been introduced [Dij82, Par72] which is the basis for conceiving modular systems. The basic idea is to separate different concerns and tackle them separately. Technologies which leverage such separation of concerns have been evolved whose examples include Object-Oriented Programming, Aspect-Oriented Programming, and Web Services which rely on hiding implementation and exposing interfaces in order to ease the development, maintenance, and understanding of programs.

Therefore, SCP relies on loosely-coupled *services* which deliver computing utility capabilities in a distributed environment. The background of this thesis focuses on the SCP and is organized as follows. Firstly, Service-Level Agreements (SLAs) are introduced in Section 1.1. Secondly, Section 1.2 discusses Quality of Services (QoS) offered by SLAs. As services are provisioned under pricing constraints, Section 1.3 exposes some issues with regard to pricing models. Following that, Section 1.4 discusses systems whose management is autonomous. In Sections 1.5, 1.7 and 1.6 three important technologies are introduced in the context of the SCP: Web Services (WS), Cloud Computing, and Grid Computing respectively.

1.1 Service-Level Agreements

1.1.1 Overview

The Service-Level Agreement (SLA) is used to express the relationship between electronic services whose contacts are employed in short-term and highly-dynamic business

scenarios. SLAs mimic conventional contracts where the contracted part refers to the *service provider* while the consumer is represented by the *service customer*[1]. SLAs define not only how services ought to behave, but also how they should not behave. In addition, SLAs do not ensure any obligation; they only define the rules that may be employed in case of non commitment of such obligations [BLM08, WB12].

1.1.2 SLA Life-Cycle

Three phases govern the SLA life-cycle. The first phase refers to the *contract definition* when a contract template is generated by describing which quality is offered – commonly, models, meta-models and ontologies are usually used for representing QoS [UTU+08]. The second phase is the *contract establishment* when parties negotiate and agree on the contract terms which expose the QoS metrics and their associated values. The third phase comprises the *contract enactment* when the service is executed and both functional and non-functional aspects are monitored in order check whether the service is properly delivered under the terms of the agreed QoS. Moreover, monitoring the SLA should be done by a third party as [GFI05] for instance.

1.1.3 WSLA and WS-Agreement

Currently, there is no standard for specifying SLAs. In the literature, WSLA and WS-Agreement are the main specifications being often used and referenced [BLM08] [DMRTV07]. The WSLA (Web Service Level Agreement) [LKD+03] SLA specification aims at providing means for configuring both provider and customer systems in order to provide and supervise their services. The WSLA proposes a formal language based on XML by allowing automatizing agreement actions. Then customer and providers are configured by interpreting the WSLA document. The configuration structure includes: (i) the parties, their roles and action interfaces; (ii) service-level parameters (SLA parameters) which are composed of both resource and aggregate metrics; and (iii) service-level objectives (SLOs), composed by guarantee actions that represents the parties' obligations. The latter SLA specification is the WS-Agreement [ACD+07] specification that proposes a language and a protocol for describing the service capabilities, for creating agreements and for supervising agreement compliance. WS-Agreement assumes that guarantees depend on states based on the idea of stateful Web Services; as proposed by the WS-RF [CFF+04] which wraps resources as stateful Web Services.

WSLA and WS-Agreement provide enough details concerning high-level structures of agreements and how they can be negotiated since they have the same goal: to define how the service will be provide and how it will be supervised. Moreover, both specifications rely on the WSDL (Web Service Description Language) in order to describe the service functionalities. On the other hand, WSLA and WS-Agreement differ from the fact that WS-Agreement also proposes a terminology and a protocol for agreement management. Furthermore, WS-Agreement does not offer a robust QoS (Quality of

[1] Also referred as *service client* in the literature.

Service) model as WSLA provides which allows combining QoS and exploiting QoS semantics.

1.2 Quality of Service (QoS)

The SLA expresses the terms of the service which are divided in functional and non-functional requirements of the provided service. Functional requirements define the service essential features necessary to meet the service purpose. In contrast, non-functional requirements define further capabilities which are not directly related to the service main purpose such as performance, fault tolerance and data persistence. Moreover, non-functional requirements are also referred as *Quality of Services* (QoS) as they do qualify the service provider. The QoS has great importance to service providers since service providers can be distinguished based on quality properties. Thereby, QoS are used to offer differentiated services by enabling the service provider to be competitive against further competitor services. For instance, QoS plays an important role in the business model of current technologies such as the Internet of Things, Web 2.0 and clouds.

The next sections discusses fault tolerant and performance properties of services followed by a discussion of the trade-off performance and fault tolerance.

1.2.1 Fault Tolerance

Fault tolerance is a means to ensure that services comply with their specification in the presence of faults [Lap85]. A fault is a representation of a system malfunction which may come from design or execution time. In order to prevent a service failure, i.e., to not disturb the service execution, faults should be handled. In order to measure the degree of fault tolerance, availability and reliability are often employed. *Availability* measures how often a service is available for responding requests. Availability can be expressed as a ratio of the time which the service was available by the measured time interval. *Reliability* indicates how much a service is reliable with respect to responding a request. The degree of reliability of a system can be expressed by high-level non-functional constraints such as strong, medium and weak [RFCRJ04].

Most of commercial service providers rely on providing availability[2] QoS. The availability is often represented by means of percentage, exposing the ratio between the total time that the service could be accessed by the customer and a given time interval [WB12, Zen10, Ama06a, Mic08, Goo08]. For instance, Zencoder [Zen10] provides audio and video encoding services whose interfaces are guaranteed to be reachable 99.9% of the time in a month. Amazon EC2 [Ama06a] ensures that virtual machines will be available 99.95% of the time in a year. If Zencoder or Amazon EC2 fail to meet the agreed availability rate, the penalties are computed by means of service credits to customers; while Zencoder is willing to refund customer based on the whole contract

[2] Also referred as *service uptime*.

time, Amazon EC2 only refunds ten percent of it. With respect to the monitoring criterion, Amazon EC2 defines an unavailable virtual machine as not being accessible for more than five minutes while Zencoder monitoring interval is one minute. Moreover, both service providers rely on ambiguous and weak definitions of availability which exclude fundamental dependability concepts.

1.2.2 Performance

The performance characteristic of a service is related to quality properties involving time metrics. Various measures can be used such as *latency, throughput, operational time*, and *response time*. Latency is mostly used to express the delay of data transfer with regard to network connections. The throughput metrics is also commonly used in network in order to measure the transfer rate, but it may be used to express the rate of a request treatment as MB/sec for instance. The operational time refers to the time that the service provider takes to make the service operational. Finally, response time typically measures the time to treat a request. While latency and throughput are commonly related to the network-level services, operation time and response time are mostly employed on services at application-level. Specifically, current work addresses the response time metrics by means of trying to decrease the overall response time. Thus, customers might experience performance improvements based on non-accurate metrics which is often expressed as the *experimented response time*. In this context, examples include approaches which addresses experimented request response time [DG08, Hue04, KP11b] as well as experimented queue waiting time [KMB+12]. Further commercial cloud providers such as [Zen10, Ama06a, Mic08, Goo08] do not specify any performance QoS in their SLAs.

1.2.3 Trade-Off Between Performance and Fault Tolerance

The use of mechanisms which implement fault-tolerant techniques imply operational overhead [Lap85]. For instance, error recovery and redundancy techniques require time to be employed thus increasing the total time of the service request. Therefore, there is a trade-off between performance and fault tolerance when designing a system since it is not practicable to prioritize both aspects concurrently. In other words, it is not possible to implement a system that achieves high-performance with a high fault-tolerance degree.

Because performance QoS metrics are not accurately described by current service provider implementations, approaches that address both fault tolerance and performance QoS do not deal with the trade-off. On the one hand, current cloud providers are more concerned about delivering the agreed availability thus considering the performance assurance a secondary goal. On the other hand, further approaches which implement actual fault-tolerant and performance assurance mechanisms [DG08, LZ10, Hue04, KP11b] do not consider the stated trade-off as they are not interest in meeting specific QoS metrics, but improving response times.

1.3 Pricing

1.3.1 Introduction

Economic interests in software gained special attention in the 60s when software started to be sold as software licenses. Software licenses represent perpetual rights for using a copy of the software. In this model, the license depends on various factors as the number of machines in which the software could be installed, the number of users that can use the software concurrently and so forth. With the popularization of the Internet and the great investment on high-speed networks over the world, not only software, but computing features as virtual resources trended to be sold as a service. In order to sell computing services, service providers propose to customers to pay according to their usage. Perpetual licenses do not fit this business model as the delivered services are now used from a remote client machine thus the software is not installed in the client's machine anymore. Thereby, customers pay for a service usage instead of buying perpetual software licenses. The fact of trading software and resources as services leads to define how customers pay for them as well as to redefine the price formation. These issues are explained in Sections 1.3.2 and 1.3.3 respectively.

1.3.2 Accounting Assessment

Accounting assessment defines how the customer will be charged for using the service. In order to enabling the accounting assessment, different accounting metrics are used which may be related to specific provided services. For example, accounting metrics may refer to usage time, input or output metrics (file size, video time, etc.), request frequency, number of users per contract, number of functionalities and so forth.

Two main accounting assessments are employed by service providers. The former accounting assessment is *pay-per-use*[3] which has been popularized and promoted by cloud computing by selling virtual hardware and software as a service. The pay-per-use accounting assessment accounts the service price based on its usage, similar to the way of how taxis and electric energy are paid. For example, the Google App Engine (GAE) [Goo08] does the accounting based on data read/write operations ($ per number of operations) while Zencoder accounting is based on video output length ($ per minute of output video). The latter accounting assessment is called *subscription* which relies on periodical payments which grants access to the service. For example, service providers use hourly, daily, monthly subscriptions along with the specified accounting metrics.

Although the subscription accounting assessment is more profitable for service providers, it is important to also offer the pay-per-use accounting assessment. Charging customers in a pay-per-use fashion is attractive to customers because they pay only for the actual service usage. In contrast, pay-per-use is not interesting for service providers because it negatively influences the provider profit. If the accounting assessment does not depend on the usage, customers tend to pay more than necessary thus increasing the provider profit margin. However, the importance of pay-per-use lies on the follow-

[3] Also referred in the literature as *pay-as-you-go*.

ing reasons: *(i)* for competition purpose as pay-per-use is widely assumed to be an intrinsic characteristics of service providers; *(ii)* customers are used to think that this model is fair; and *(iii)* service providers still have a way of overcoming the pay-per-use disadvantages by setting a higher prices than subscription plans. As a result, most of service providers provides both pay-per-use and subscription plans.

Lastly, subscription accounting assessment can be understood as a customization of a pay-per-use which relies on the time metrics. If this pay-per-use is modified to rely on periodical payments which time metrics is fixed to a minimal interval, it becomes a subscription accounting assessment; which may also limit to a maximum amount of service-specific accounting metrics. Thus, service providers create subscription plans which usually offer tariff reductions as the maximum amount of specific metrics increases – for example, by mimicking mobile telephony companies. For instance, Zencoder's pay-per-use option accounts 0.05U$ per minute of video output while its *launch* subscription plan relies on monthly payments of 40U$ for a maximum of one-thousand minutes of output[4]. Moreover, some of these providers even do not provide the pay-per-use option, hence only offering subscription plans such as SalesForce[5] [Sal99]. In a brief, the subscription is widely accepted by customers as it is very hard for customers to predict the service usage. Thus, customers do not mind to rely on predefined subscription plans which brings the feeling that the service is pay-per-use. More details about these issues can be found in [LB09, LDBK10].

1.3.3 Price Formation

Price formation refers to how to define the price of a service. Basically, price formation may be formed in three ways: cost-based, value-based and competition-oriented [LB09]. Firstly, the *cost-based* price formation assumes that the price is defined based on its operational costs. These costs are represented by expenses which may include third party services, resource providers, storage providers and software licenses for instance. For instance, the Amazon DevPay [Ama12] provides an integrated billing tool which takes into account the use of Amazon EC2 instances. Thus Amazon EC2 customers can automatically define the price of the service based on the utilization of the underlying infrastructure in a cost-based fashion. Secondly, the *value-based* price formation depends on the demand for the service, i.e., how valuable the service is in the current market. Lastly, the price formation can be driven by the price of competitor services. The *competition-oriented* price formation is then useful for allowing new providers to be competitive or in order to dispute market share to other competitors.

Furthermore, another aspect with regard to the price formation refers to the decision of setting the price. The price can unilaterally be defined by the service provider or it can be set in a flexible manner which includes the price into the negotiation process between customer and providers. However, most of service providers does not allow changing the service price by relying on predefined prices and accounting assessment plans.

[4]Further minutes cost 0.04U$ per minute.

[5]Except for SalesForce Data.com service which also allows customers to pay based the stored data.

1.4 Self-Managed Systems

1.4.1 Dynamic Adaptation

Dynamic adaptation refers to changing the behavior of a system at runtime. Adaptable capabilities are useful in unpredictable scenarios where environment changes or users needs require modifying the system behavior. For instance, resource availability should be dealt with in order to ensure that the system will be able to execute its tasks. Similarly, faults should be handled in order to ensure that the system will behave as specified. Further adaptation goals include optimization interests where the system adapts itself by aiming at improving performance or reducing resource waste for example [Bui06, UPF+08].

Most approaches that offer support for conceiving self-adaptable systems separate the system functional and non-functional system concerns. Then adaptation interests are dealt with as a non-functional concern which simplifies implementing and maintaining the system as well as promoting reusability. The technologies that enable implementing self-adaptable systems in a modular fashion include Aspect-Oriented Programming, component-based design, Web Services and computational reflection. Furthermore, the adaptive behavior can be implemented either in the application-level or as a utility in the middleware-level. Further discussion about adaptation techniques can be found in [BAP05, KC03, AC03, MSKC04, CFI+09].

1.4.2 Autonomic Computing

Autonomic Computing [Hor01] proposes the idea of developing autonomous systems in order to ease system use and administration. Autonomous systems should then manage themselves in an autonomous fashion, inspired in the human autonomic nervous system which controls some functions as breath rate, pupil dilatation and heart rate. In [KC03], the authors propose the MAPE (Monitoring, Analysis, Planning, Execution) control loop which serves as the cornerstone of an architecture for developing self-managed systems. The MAPE control loop decomposes adaptation in four interests as explained next. The Dynaco adaptation model proposes a similar approach to the MAPE for distributed application by leveraging component-based design [BAP05, Bui06].

Monitoring Monitoring is in charge of gathering information about adaptation interests. Monitoring mechanisms may rely on pull and push flows in order to either keep the system aware about monitoring metrics periodically or to allow specific information to be required.

Analysis During the analysis, it is decided whether an event about the monitored data is relevant enough for triggering an adaptation action. If so, an adaptation strategy should be generated in order to change the application behavior to achieve the targeted state.

Planning The adaptation strategy is transformed in an adaptation plan in the planning phase. The plan is a set of instructions detailedly describing the actual

changes to be executed by the system.

Execution The previous plan is then executed in the execution phase. In order to execute the plan, the system execution flow is intercepted and the instructions are applied.

1.5 Web Services

1.5.1 Loosely-Coupled Distributed Applications

The Service-Oriented Computing (SOC) [PG03, PTDL07] proposes a modular and loosely-coupled design for building distributed applications. SOC takes advantage of the service abstraction whose interactions are described by SLAs. Aiming at instantiating this view, the Service-Oriented Architecture (SOA) was then proposed as a representation of the SOC. The SOA describes an architecture which enables conceiving service-based applications. In a brief, SOA architecture has three layers which respectively address: (i) service discovery and binding; (ii) service composition in which services are combined in order to conceive composite services; and (iii) service management features that are mainly related to service life cycle. Moreover, the SOA has been proposed to rely on Web Services [W3C10] as a mean of integration of different applications and platforms.

From another point of view, the SOA can be understood as a natural evolution of the component-based design [Szy03]. In [CH04, Yan03, PAB11], the authors understand components and SOA as complementary approaches: on the one hand, services are loose-coupling, dynamic and business-oriented, on the other hand, components offer an efficient development model that separates concerns and promotes re-usability. These latter properties simplify the service development. Examples of component approaches based on the SOA include the Service-Component Architecture (SCA) [Ope07], Declarative Services [OSG07] and iPOJO [EHL07]. However, both SOC and component point of views about SOA consider the utilization of Web Services for interoperability purpose[6].

1.5.2 Web Services

Two architectural styles have been used to implement Web Services: one based on WS-* standards, another based on the concept of Representational State Transfer (REST). The former technology relies on the SOAP (Simple Object Access Protocol) protocol as a means for interfacing Web Services [W3C10]. SOAP leverages lower-level communication protocols such as HTTP (Hypertext Transfer Protocol) by exchanging messages formated in XML (Extensible Markup Language) through the service descriptor represented by a WSDL (Web Service Definition Language) interface. The WSDL is an XML file which describes the service functionalities in a RPC-based fashion. In other

[6]The use of Web Services by Declarative Services and iPOJO are enabled by Distributed OSGi [ASF09]

words, services send requests to other services by invoking their operations, then these operations will be executed remotely and the result is sent by means of the request response. This way of building service-based application by leveraging Web Services in an RPC-like manner served as basis for several approaches which address service composition by means of BPEL orchestration or choreographies. In spite of the great advantage of not changing the way of how applications are designed, i.e., by calling operations, it is hard to manage SOAP/WS-* service compositions.

In contrast to the WS-* based architectural style, the second approach for implementing Web Services follows the REST style, which allows building service-based applications in a simpler way. Web Services based on the REST architecture are called RESTful Web Services and take advantage of the HTTP protocol as an application protocol, not only a transport protocol. In the REST architecture, services are exposed as resources identified by URIs that facilitate the referencing of services on a network of distributed servers. RESTful Web Services are accessible through the use of simple HTTP methods, e.g., GET, PUT, POST, UPDATE. In addition, RESTful Web Services take advantage of content negotiation to provide various data formats such as XML and JSON (JavaScript Object Notation,) which allow them to fit the data format according to the service needs. With respect to the service provider state, the stateless characteristics of RESTful Web Services is suitable for developing inter-domain Web Services in large-scale. On the other hand, RESTful Web Services requires that customer services understand the semantics of the data formats (media-types) which are exchanged.

1.6 Grid Computing

1.6.1 Overview

Grid computing was firstly used to aggregate clusters from different domains for performing high-performance computing in large-scale. Following that, grids addressed the utilization of further heterogeneous and low-cost resources such as desktop computers and mobile devices. In order to manage resource access policies, grids leverage the concept of Virtual Organization (VO) to define and ensure user privileges in each set of resources. Thereby, various organizations participating on a grid can configure the grid usage to suit their local policies [Fos02]. In order to make grids transparent for users and promote interoperability, grids leverage programming abstractions based on open standards that unify the way of how grid resources are used. For instance, grid users may develop distributed applications based on abstractions as GridRPC, batch jobs and files [SNM+02, GJK+08].

Figure 1.1 illustrates a common grid architecture. Firstly, according to VO policies, grid users reserve a set of resources and submit jobs through the broker. Secondly, jobs are sent to the grid scheduler which allocates jobs on resources given job resource requirements and according to a scheduling algorithm. Finally, when jobs are executed, users are informed. Moreover, distributed file systems [LS90] are widely used in grids in order to store and share data transparently.

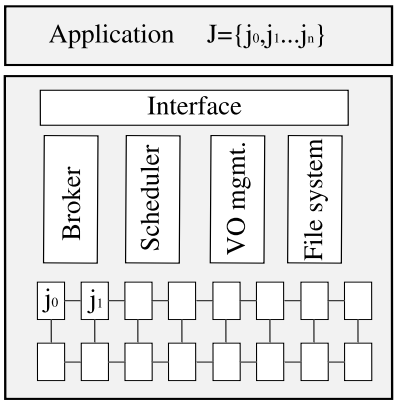


Figure 1.1: Applications are executed on top of grid infrastructures by submitting jobs to grid resources.

The main programming abstraction for developing grid applications is the *job* abstraction. Jobs can either hold other jobs a parallel program or represent a sequential program as part of a set of jobs. Moreover, jobs contain meta-information which includes their software the resource requirements. By leveraging jobs, various distributed applications can be developed such as BoT (Bag of Tasks) applications [CBC+04] , complex workflows [HWS+06] as well as applications that leverage the generic Master/Worker pattern.

1.6.2 Application Programming Interface

Grids offer complete Application Programming Interfaces (APIs) which allow grid users to have extensive control of grid resources. However, current grid offer different and complex APIs thus increasing the complexity of application development and maintenance. In order to ease the use of grids, to standardize the grid usage and to allow grid applications to be compliant with different grid platforms, the Simple Grid API (SAGA) [GJK+08] proposes a standard grid programming interface. SAGA addresses interoperability and complexity through simple grid programming abstractions and operations. SAGA enables grid application development by taking into account a unique interface while it enables the execution of such an application in different grids. For achieving interoperability, SAGA relies on adaptors which enable implementing backends for various grids as SSH, Globus, XtreemOS, Condor and so forth.

SAGA compiled the main abstractions for developing distributed applications based on various requirements. Firstly, the *file* abstracts transparently over a distributed file system an usual local file system. Secondly, the *data streaming* abstraction eases the utilization of remote communications over sockets. Aiming at a higher-level remote communications, the *RPC* abstraction is also provided by SAGA whose specifications are defined based on GridRPC [SNM+02]. The *task* abstraction encapsulates synchronous and asynchronous operations in order to facilitate the management of distributed calls.

Finally, the *job* abstraction is supported by SAGA which allows managing jobs based on statical and dynamic information sent through job callbacks. Additionally, jobs can be submitted in both interactive and batch modes.

Furthermore, grid usage can also take advantage of the *pilot-job* abstraction. The DIRAC (Distributed Infrastructure with Remote Agent Control) [CER04, TBB+08, CGP+10] grid proposes pilot-jobs as an abstraction for improving resource allocation in grids. Before submitting main jobs, meta-jobs called pilot-jobs are submitted to available resources by launching an agent. This agent is responsible for calculating the resource capability of execute main-jobs and send it to the scheduler. The scheduler then matches its job queue received from grid users to available resource capabilities. Thereby, the job scheduling is distributed among the resources by relying on updated resource metrics. Moreover, further grid approaches also takes advantage of the idea of pilot-jobs as SAGA big-job [LLJ10] and Condor [Uni12].

1.6.3 Grid Platforms

Globus The Globus Toolkit [Fos06] is a grid middleware as part of the Globus Alliance. Globus Toolkit architecture relies on Web Services in order to manage the grid components. It uses the WS-Resource Framework (WSRF) [CFF+04] and wraps grid resources in order to ease the management of the middleware. Globus Toolkit transfers data through the FTP (File Transfer Protocol) protocol adapted for grids and manage data replication to improve data availability. Moreover, Globus Toolkit also provides an execution management environment for specific applications.

XtreemOS The XtreemOS [CFJ+08] project targets providing a transparent grid execution environment as an usual operating system. Core components of the XtreemOS grid operating system are built as Linux kernel modules in user-space which makes common operating systems grid resources. Thus, the overall overhead of tasks that require detailed information about the system is decreased. Moreover, XtreemOS supports desktop computers, clusters and mobile devices. With respect to its interface, it not only provides an extensions of the SAGA interface, but it also provides a console which supports POSIX-compliant commands.

Grid5000 The Grid5000 aims at offering an integrated distributed environment where researches can deploy their prototypes in large-scale. The Grid5000 currently holds 1,594 nodes, a total of 3,064 processors and 8,700 cores over ten sites in France [7] shared among various research institutes. In order to use Grid5000, an operating system image is deployed on reserved nodes. Indeed, Grid5000 is a resource-oriented testbed and can be compared with IaaS clouds in the sense since both provide resource as abstraction for programmers. Although Grid5000 does not offer higher-level abstraction such as jobs, its extensible and flexible tools, e.g., OAR, Kadeploy, Adage, allow grid users to deploy further programming support as Globus Toolkit, XtreemOS, DIET, Nimbus, Apache Hadoop and so forth.

[7] Two more sites from Brazil and Luxembourg will be added to Grid5000.

DIET DIET (Distributed Interactive Engineering Tool-box) [CD06] is a grid middleware which tackles the development of grid applications by using GridRPC [NMS+05, SNM+02]. GridRPC lies on the RPC (Remote Procedure Call) paradigm with dynamic resource scheduling in an environment composed by different administrative domains. DIET components are in charge of allocating resources when the application remotely calls computation operations through GridRPC calls.

DIET is based on a hierarchical distributed scheduling design. DIET users send the operation to be computed to a DIET entry point (master agent) by means of a GridRPC call. The master agent dispatches the call to hierarchical lower-level entities called SeD (Server Daemon) which are in charge of executing the actual operation. When an SeD receives the call, it checks if it is able to respond to the call by checking dynamic information such as data availability and resource load and scores its ability. Then the SeD forwards the call to its SeD children which also calculate their ability based on the given call. All SeDs answer their upper SeDs which sort capability scores of their children and store in a list. Following that. the master agent sorts the capability lists of its SeD children in a merged list which is forwarded to the user. Finally, the user chooses which SeD is more suitable for executing the call and sends the call to the chosen SeD to be executed.

1.7 Cloud Computing

1.7.1 Overview

Cloud Computing does not have a common definition. At first, most definitions relied on the idea of outsourced service providers which deliver virtual machines on-demand in a pay-per-use accounting assessment. Later, the basic idea of outsourcing resources was extended to a larger and generic set of computing capabilities rather than virtual machines [MG11, Sch10, Bég08, VRMCL08, KTKN11, JMF09].

Cloud Computing targets adapting the provided services according to environment changes dynamically. This refers to the idea of *elasticity* which allows clouds to modify non-functional requirements not only according to demand changes, but also for consolidation purpose. In the context of IaaS clouds, elasticity may refer to either vertical elasticity, whereby virtual machines requirements are modified, or horizontal elasticity, which means changing the amount of virtual machines. For instance, an application which uses a single virtual machine may become overloaded, then it may trigger a vertical elasticity in order to accommodate the current load. On the other hand, the cloud provider may suspend, migrate[8] or shut down some virtual machines in order to save energy consumption during low demand peak.

In spite of the advantage of acquiring outsourced and elastic services on-demand, clouds main drawbacks lies on privacy, security and dependability [Vie09]. As customer data is stored in clouds, customers do not have control on their own data anymore. This can be used for gathering useful information about cloud customer for business

[8]Currently, the *live migration* technology enables migrating virtual machines at runtime without shutting down the virtual machine.

purpose. Furthermore, some applications require non-trivial security requirements such as critical applications. The way of how availability is offered by cloud providers are not enough for building dependable applications. For instance, the Amazon EC2 SLA terms exclude Amazon from any availability issue caused by factors outside of Amazon's reasonable control[9].

1.7.2 A Common Layered Architecture

Currently, a common cloud layered-architecture is found in the literature [Sch10, KTKN11, CBP+10, MG11] where computing capabilities are exchanged between service entities. Figure 1.2 illustrates this cloud architecture. The lowest layer refers to the *Infrastructure as a Service* (IaaS) which relies on the resource abstraction. The IaaS layer leverages virtualization techniques in order to deliver virtual machines as resources. These virtual resources actually are operating system images. The upper layer is called *Platform as a Service* (PaaS) which provides a developing support for building final applications. In the PaaS layer, programming tools and abstractions are offered as external services which play the role of conventional programming libraries. In the highest-level layer, the *Software as a Service* (SaaS)[10] layer provides applications for final users.

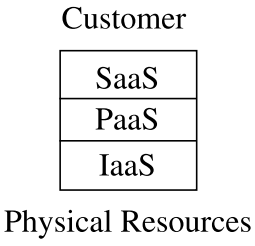


Figure 1.2: A common view of Cloud Computing as a layered architecture where layers interact with each other by means of services. Service requests are the primitives which enable the service to be delivered according to the SLA.

1.7.3 Infrastructure, Platform and Software as a Service

IaaS IaaS clouds rely on the virtualization technology in order to deploy virtual machine instances on top of physical resources. Virtualization allows running guest operating systems inside a host operating system. Thereby, virtual machine instances actually are operating system running on a virtual resource. The hypervisors are in charge of managing virtual machines by interacting with the physical machine operating system. Examples of current hypervisors include Xen [CS03], KVM [RH06], and VirtualBox [Ora07]. Moreover, kernel containers can also be used to manage virtual machines as a kernel built-in solution.

[9] For the full Amazon EC2 SLA terms, see [Ama08].
[10] Also referred as *Service as a Service*.

The first IaaS cloud created was the Amazon EC2 [Ama06a] as a result of Amazon's data center improvement. Amazon EC2 relies on its proprietary AMI (Amazon Machine Image) virtual machine format and its own API. Following that, open-source IaaS clouds were developed such as Nimbus, Eucalyptus, OpenStack and OpenNebula, all of them supporting the Amazon EC2 API. Moreover, cloud IaaS solutions are commonly used together with storage services, e.g., Amazon S3 [Ama06b].

Aiming at standardizing IaaS cloud interfaces, the Open Grid Forum published the OCCI (Open Cloud Computing Interface) [ME11] specification. OCCI relies on the compute, storage and network abstractions in order to represent cloud IaaS resources. The *start, stop, restart* and *suspend* operations govern resources through a RESTful API. Furthermore, proprietary interfaces as Amazon EC2 also relies on these simple operations.

Furthermore, IaaS clouds can be also classified as private, public and hybrid. In order to tackle privacy and security issues, IaaS clouds can be used in an intra-domain scope. This characterizes a *private cloud* whose virtual instances are managed by the organization which then takes advantage of managing virtual resources instead of physical resources. In contrast, *public* clouds consist of third parties which provide virtual machines in an outsourced manner. Lastly, clouds can also be classified as *hybrid* whose virtual machines are held by both private and public clouds. Hybrid clouds is a complementary solution for private clouds in a sense that it enables handling unpredicted resource demands in a private cloud.

PaaS and SaaS While the IaaS layer has been established on top of well defined interfaces [Ama06a, ME11], there is a lack of agreement about PaaS and SaaS cloud interfaces. Despite the OCCI effective description of a cloud IaaS interface, it does not tackle neither PaaS nor SaaS cloud interfaces. Indeed, in these latter layers, a huge number of functionalities should be provided which makes it very difficult to define a simple and clean interface. For instance, the PaaS layer aims at providing services which support data base access, authentication, parallelization and domain-specific libraries as SalesForce [Sal99] and the Google Application Engine (GAE) [Goo08]. With respect to the SaaS layer, although SaaS customers needs are more customized rather than developer needs, SaaS interfaces similarly aim at providing specific features. As a consequence, the lack of uniform and standardized PaaS and SaaS interfaces implies the incompatibility of cloud providers; thus subject to vendor lock-in and integration issues.

Chapter 2

Related Work

This chapter presents some current approaches which support the development of service providers on top of distributed infrastructures. The main focus is on approaches that provide quality guarantees. In order to organize related work, three main groups were defined based on the following aspects: pricing (cf. Section 2.1), SLA management (cf. Section 2.2) and resource acquisition (cf. Section 2.3). Moreover, the SLA management aspect was divided in three subgroups which address SLA life-cycle (cf. Section 2.2.1), SLA translation (cf. Section 2.2.2), and QoS assurance (cf. Section 2.2.3).

Finally, a comparison table is exposed in Section 2.4 which summarizes the main related work here presented.

2.1 Pricing

In general, the solutions which address pricing aspects do not include in their SLA the penalties in case that the agreed quality is not met. Some of them [Ama06a, Goo08, Sal99, Zen10] are willing to offer service credits if a percentage of availability is not met. However, these approaches rely on a very limited and often not clear definition of SLA violation because service providers are not able to properly ensure the agreed QoS. Thereby, refunding policies would seriously compromise provider profit.

In [MFG10, MG10], Macías et al. address the maximization of IaaS cloud providers by means of resource over-provisioning and by dynamically setting the service price according to resource usage. Virtual machines are then shut down based on their usage pattern thus enabling to allocate other virtual machines. In case of resource shortage, the approach minimizes losses by considering either violating SLAs or rescinding contracts. Even though these approaches consider pricing aspects, the authors only tackles the availability QoS. Moreover, similar to Amazon EC2, these approaches only consider an SLA violation as a virtual machine that has been shut down after a fixed and significant amount of time. Furthermore, these solutions tackle the IaaS layer thereby not providing higher-level programming abstractions for building SaaS service providers.

2.2 SLA Management

2.2.1 SLA Life-Cycle

The SLA@SOI project [THK+10] proposes a hierarchical and integrated architecture for building service-based applications. The architecture is composed by three SLA managers which address business, software and infrastructure SLA managements in a layered fashion. In the highest-level, business aspects between the provider and its customers are dealt with. The following layers address software and infrastructure wrapped as services. By considering software and resources as service, the SLA@SOI architecture aims at separating service management from SLA management. However, by aiming at a generic approach for supporting arbitrary services, the architecture does not define specific realization mechanisms; it assumes that services dependencies are responsible for meeting service software and resource requirements.

The SOA4ALL project [KLH+10] aims at easing the conception of service-based applications by leveraging the SOA, Web 2.0 and Semantic Web in a context-aware environment. The SOA4ALL proposes a layered architecture which lies on a graphical user interface, an intermediate communication channel and an underlying infrastructure. Moreover, the communication channel enables third party service to be integrated. In the highest layer, business processes are conceived and then translated to BPEL (Business Process Execution Language) compositions which are executed in the infrastructure. Thereby, the SOA4ALL approach tackles service composition driven by quality aspects through comprehensive service discovery and bindings based on semantics. Nevertheless, the architecture does not specify how resources are acquired nor managed and pricing aspects are not defined. The authors only explain how further technologies might meet these needs. Moreover, actual QoS assurance mechanisms are not proposed; QoS properties are supposed to be met by proper service compositions.

Macías et al. [MFG10] addresses the negotiation and provisioning of virtual machines driven by economic aspects. The architectural details of this approach is described in [MG10]. However, it does not tackle performance, relies on a relaxed definition of SLA violation based on availability as well as it does not support higher-level programming abstractions; as discussed in Section 2.1.

2.2.2 SLA Translation

The authors of [KW10] model the translation of SLA based to service dependency properties; however they do not tackle how services translate their SLA to infrastructure-level configurations.

In [CIL+08], the authors profile tier applications to know the response time based. SLAs are translated by means of CPU requirements to application servers. However, this work does not provide fault-tolerant mechanisms and it does not tackle distributed applications.

Stantchev and Schröpfer [SS09] service replication is used in order to improve and ensure performance and fault-tolerance QoS. The authors translate QoS to infrastructure-level by means of setting the service replication degree. Nevertheless, the authors do

not explain how assurance mechanisms work nor how SLA and resources are managed. Furthermore, the work does not consider economic aspects such as service price, resource costs and fines owing to SLA violations.

The GridNEXT project [HKSW06] investigate how SLA objectives are translated to infrastructure configurations in order to enable high-performance computing service providers to meet the agreed QoS. The translation process relies on a knowledge database previously filled. However, the architecture remains conceptual thus not providing enough information about how to implement actual QoS assurance mechanisms. The authors assume that the infrastructure should support QoS assurance mechanisms.

Moreover, further work about SLA translation can be found in [LTH09].

2.2.3 QoS Assurance

The MapReduce programming model [DG10, DG08] addresses data processing in large scale. It leverages the concept of map and reduce primitives of functional languages and considers a data shared space such as a distributed file system. Developers who take advantage of MapReduce should implement the application-specific operations *map* and *reduce* and provide information about the location of input and output data. The MapReduce Library [DG08] 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, more efficient fault-tolerant techniques can be conceived based on dynamic job metrics as job elapsed time and job state. Thereby, malfunctioning jobs can be earlier identified thus triggering repairing actions at this very moment instead of postponing them to the end of the computation. Moreover, MapReduce does not address a complete PaaS solution, i.e., it misses further supports for dealing with the service negotiation and provisioning. Therefore, MapReduce is understood to be an intermediate-level tool that ease the development of PaaS solutions which aim at data-processing in large-scale, while further service higher-level aspects are not tackled.

Clouds resources have been used to improve performance of distributed applications. In [LZ10], the authors propose the RC^2 job scheduling algorithm for grids which relies on rescheduling previously queued jobs. The RC^2 algorithm replaces grid resources by cloud resources if a grid resource is identified as delaying job executions. Then, the jobs previously assigned to the grid resource are rescheduled to a cloud resource. The argument used by the authors of the RC^2 algorithm is that cloud resources are more reliable rather than grid resources. In [DFL11], the authors introduce the SpeQuloS framework which aims at providing QoS to desktop grids. The framework adds further resources from clouds in order to decrease the application response time. Moreover, SpeQuloS relies on a credit service which computes the costs of using extra resources from clouds.

Lastly, the SAGA big-job instrumentally used clouds to ensure deadline [LLJ10]. The main drawbacks of these solutions is that they only tackle performance aspects and they do not tackle pricing aspects completely.

The GridWay [Hue04] project proposes an adaptive framework for job execution on grids. Tools for monitoring and analyzing jobs are added to the Globus middleware in order to check if job performance degrades. If so, the framework reschedules the job in order to improve its performance. However, the scheduling policy does not consider further queued jobs belonged to other applications. This issue can be improved by considering distributed scheduling techniques such as work stealing [TGT+10]. Moreover, this work does not support neither SLA management functions nor pricing aspects and exposes low-level resource details to applications.

In [KP11b], the authors propose the CooRM architecture for supporting complex high-performance distributed applications with dynamic resource requirements. In CooRM, job scheduling is distributed and employed at the application level thus the central scheduler is in charge of resource discovery and scheduling appliance while applications are in charge of selecting the available resources. In order to improve performance, the authors propose to not delay application executions by prioritizing previously submitted applications based on their estimated execution time. Furthermore, in [KP11a], the authors explains how CooRM tackle resource wastage. Nevertheless, fault-tolerance, pricing and SLA management are not addressed.

In [Jos08], the authors propose SLAWs as a flexible architecture for enforcing SLAs. Services are decoupled in order to address non-functional service concerns wrapped in a separated service. However, specific SLA enforcement mechanisms are not described.

In [Per05], the author proposes a hierarchical SLA management for SLA enforcement. The approach relies on adaptation policies guided by high-level objectives which adjust network traffic based on current QoS values. Nevertheless, this work is limited to low-level network traffic control thus not offering higher-level support for services which builds on distributed infrastructures based on the job abstraction.

Zencoder [Zen10] uses Amazon EC2 virtual machines to encode video and audios. In [Clo11], Zencoder showed to perform faster than some competitors. However, Zencoder does not publicly explains how the performance is improved and does not include QoS in its public SLA description.

2.3 Resource Acquisition

Pilot-jobs [TBB+08, CGP+10] provides a solution for decoupling resource acquisition from resource utilization in a grid environment. In order to also take advantage of IaaS cloud resources, SAGA [GJK+05] leverages the idea of pilot-job by introducing the big-job [LLJ10] abstraction which acquires cloud resources, deploys virtual machines and submits grid jobs to the booked resources. With regard to fault tolerance, SAGA big-job replaces pilot-jobs which fail to acquire resources from clouds. However, SAGA big-job assumes that it is the developer charge to customize virtual machines and defining their requirements. Moreover, SAGA big-job does not consider resource costs

in its pricing model and provide no means for increasing the provider profit.

In [KTKN11], Kecskemeti et al. propose an automatic approach for acquiring and deploying virtual machines in order to aid service developers and decrease the deployment time. The main idea lies on automatic mechanisms which split virtual images and replicate its parts on various repositories. Moreover, the authors complement this approach in [KMB+12] by addressing the automatic management of virtual appliances in different clouds. Nevertheless, the drawback of these solutions is that service developers have to directly deal with the lower-level resource abstraction as well as they still have to manage the life-cycle of virtual machines. In addition, the authors consider virtual machine instantiation driven by customer requests in contrast to instantiating virtual machines driven by contract establishments.

Genaud and Gossa [GG11] investigate various policies for acquiring and releasing resources for grid batch jobs. The idea is to fit job completion time and virtual machine booking in a pay-per-use pricing accounted hourly. Thus, the authors propose policies for meeting the trade-off between infrastructure costs and performance. However, the authors fail in addressing SLA management functionalities and performance is not ensured.

2.4 Summary of Related Work

2.4.1 Comparison Table

The Table 2.1[1] depicts the comparison of this thesis and main work previously identified in Section 2. The comparing criteria emphasizes the main features for enabling service execution management as explained next.

Layer Refers to the cloud architecture layer tackled by the approach (cf. Section 1.7.2). Approaches which provide resources as abstractions are classified as infrastructure-level (*IaaS*) while approaches which provide further higher-level abstractions such as jobs are considered to tackle the platform-level (*PaaS*).

Pricing If set *yes*, means that the approach addresses pricing aspects, e.g., price, costs and fines. Otherwise, *no* is used.

SLA Life-Cycle This attributed defines whether the approach offer SLA functionalities which enable addressing the SLA life-cycle. *Yes* means that the approach covers the SLA life-cycle while *no* means the contrary.

SLA Translation Refers to translation of SLA high-level properties or QoS metrics to infrastructure-level configuration. If the approach describes how the translation is employed, it is classified as *yes*, otherwise *no* is used.

QoS Aspect Expresses which kind of QoS is tackled by the approach. The referred QoS aspects are performance (*Perf.*) and fault tolerance (*FT*).

[1] Although Amazon provides the Amazon Elastic MapReduce [Ama09], MapReduce [DG08] refers to the Google MapReduce programming model.

QoS Assurance If the approach provides QoS assurance mechanisms, *yes* is used, else it is classified as *no*.

Resource Acquisition Refers to whether the approach addresses resource acquisition; it may refer to either virtual or physical resources. If so, the approach is classified as *yes*, otherwise *no* is applied.

Approach	**Layer**	**Pricing**	**SLA Life-Cycle**	**SLA Trans-lation**	**QoS Aspect**	**QoS Assur-ance**	**Resource Acqui-sition**
MapReduce [DG08]	PaaS	No	No	No	Perf., FT	Yes	Yes
SOA4ALL [KLH+10]	PaaS	No	Yes	No	Perf., FT	No	No
Macías et al. [MFG10]	IaaS	Yes	Yes	No	FT	No	Yes
Kecskemeti et al. [KTKN11]	IaaS	No	No	No	Perf., FT	Yes	Yes
GridWay [Hue04]	PaaS	No	No	No	Perf., FT	Yes	NA
CooRM [KP11b]	IaaS	No	No	No	Perf.	Yes	No
SAGA Big-Job [LLJ10]	PaaS, IaaS	No	No	No	Perf.	Yes	Yes
SLA@SOI [THK+10]	PaaS, IaaS	No	Yes	No	Perf., FT	NA	Yes
GridNEXT [HKSW06]	PaaS	No	No	Yes	Perf.	No	NA
Stantchev and Schröpfer [SS09]	PaaS	No	No	Yes	Perf., FT	No	No
Genaud and Gossa [GG11]	PaaS	Yes	No	No	Perf.	No	Yes
This thesis	**PaaS**	**Yes**	**Yes**	**Yes**	**Perf., FT**	**Yes**	**Yes**

Table 2.1: Comparison of main related work according to various criteria.

2.4.2 The Gap

The general observation is that approaches that focus on the infrastructure-level provide actual mechanisms which ensure quality aspects. However, these approaches do not consider higher-level aspects as pricing and SLA management. On the contrary, approaches that address SLA aspects do not specify how QoS properties can be enforced since they focus on high-level SLA management tasks.

Another observation refers to the pricing criterion. Although most approaches enable applying pricing models, they neither define the pricing model nor describe it.

In contrast, commercial clouds provide fully information about pricing. Nevertheless, it is not worthy to put these approaches in Table 2.1 because they rely on proprietary solutions which do not described how further criteria are addressed.

With regard to QoS aspect, both performance and fault tolerance are satisfactorily addressed. However, most of solutions rely either on experimented QoS metrics rather than accurate metrics. Regarding resource acquisition, most approaches solve the problem of acquiring resources. Nevertheless, these approaches often exposes resource-level details for final customers.

2.4.3 Filling the Gap

This thesis addresses the required features to build a complete solution for service execution management. The last row of the Table 2.1 shows this thesis addressing all the criteria as a solution in the platform layer. Moreover, it is important to remember that the previous criteria were chosen by aiming at enabling service execution management on top of distributed infrastructures. A summary of how this thesis fills the gap is exposed next.

Pricing This thesis relies on a pricing model whose price formation is based on provider's costs in a pay-per-use accounting assessment. Penalties for SLA violations are also taken into account whose penalties are computed as monetary terms. Moreover, this thesis prioritizes customers which positively contributes for increasing the provider profit.

SLA Life-Cycle This thesis designs and implements a complete set of SLA management functions which include service negotiation, instantiation, provisioning, and termination. Moreover, this thesis defines how pricing aspects are integrated into its solution.

SLA Translation Based on service profiling, this thesis translates performance and fault-tolerance QoS metrics to lower-level system configurations in order to configure the mechanisms responsible for ensuring QoS.

QoS Assurance This thesis provides mechanisms for ensuring both performance and fault-tolerance qualities for distributed services. These mechanisms rely on resource requirements and dynamic job metrics which enable immediately reacting to job malfunctions.

Resource Acquisition This thesis deals with resource acquisition by transparently acquiring, deploying and releasing resources thus freeing service developers from these time-consuming and error-prone tasks. Moreover, by tackling the whole SLA life-cycle, this thesis assumes that contract negotiation triggers resource acquisition in contrast to current approaches that acquire resources when requests are sent.

Part II

Conception

Chapter 3

SLA-driven Service Execution Management

The contribution of this thesis is in the context of service providers built on top of distributed infrastructures such as grid and clouds. It provides an autonomous support for service execution management aiming at increasing provider profit. This challenge is achieved by reducing costs on infrastructure usage, preventing SLA violations and prioritizing more profitable customers. In order to address this problem, this thesis translates high-level quality aspects to system configuration to ensure the proper service execution by acquiring resources on-demand. Moreover, service under-provisioning is also taken into account as a feature to increase provider profit.

The reminder of this chapter is organized as follows. Section 3.1 discusses definitions, assumptions and life-cycles. The main goal of this thesis is explained in Section 3.2. The Section 3.3 explains how contract templates are created. The following Sections 3.4 and 3.5 address service execution management aspects related to QoS assurance and resource acquisition and allocation respectively.

3.1 Underlying Concepts

This thesis relies on a terminology which is used by the research community but in different contexts. In order to avoid ambiguity during the lecture of this manuscript, the Section 3.1.1 introduces the definitions and assumptions used by this thesis. Following that, the Section 3.1.2 introduces the variables useful for understanding this text. Finally, the life-cycles of contracts, requests and jobs are defined in Section 3.1.3.

3.1.1 Definitions and Assumptions

This thesis relies on the following definitions and assumptions:

Definition 3.1 *An **electronic service** – or just **service** – is a high-level abstraction that represents a way of delivering a software or a computing capability as a part of a*

software. The contract terms of a service is represented by a Service-Level Agreement (SLA) which is established between the service customer and the service provider.

Definition 3.2 *An **electronic contract** – or just **contract** – describes the contractual terms of a service between the service customer and the service provider. The contract is represented by a Service-Level Agreement (SLA).*

Definition 3.3 *A **job** is an intermediate-level software abstraction which is useful for building services and handling customer requests.*

Definition 3.4 *A **request** is the means of how customers use the service. Customers send requests to the service provider which should treat it in accordance to the SLA.*

Assumption 3.1 *Resources are acquired by the service provider from a resource provider in an outsourcing fashion.*

Assumption 3.2 *Jobs may present misbehaviors which means to* fail *or to be* delayed. *Moreover, a job misbehavior represents a fault for the service provider.*

3.1.2 Variables

Jobs, contracts and requests are represented by variables as explained next.

- J is the set of all jobs where:
  - j_i^n is the i-th job which holds n replacements
  - J_{q_k} set of jobs belonged to the k-th request q_k
  - J^F set of `FAILED` jobs
  - J^R set of `RUNNING` jobs
  - J^C set of `CANCELED` jobs
- Q is the set of all requests where:
  - Q^T set of `BEING_TREATED` requests
  - q_k is the k-th request
- C is the set of all contracts where:
  - c_i is the i-th contract.

3.1.3 Life-Cycles: Contract, Request and Job

The management of service execution is driven by the states of contracts, requests and jobs. The life-cycle of these abstractions are presented next.

Contract Life-Cycle The contract life-cycle aims at instantiating the SLA life-cycle which comprises the definition, establishment and enactment phases (cf. Section 1.1.2). The Contract life-cycle is depicted by Figure 3.1 which also illustrates the SLA life-cycle phases.

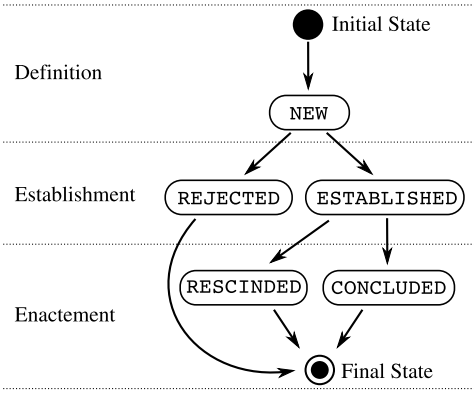


Figure 3.1: Contract life-cycle.

The contract life-cycle states are described as follows.

NEW A just-arrived contract proposal implies the creation of a contract whose state is `NEW`.

REJECTED If the service provider decides to not accept the contract proposal, the contract state is set to `REJECTED`.

ESTABLISHED If the service provider accepts the contract proposal, the contract state is set to `ESTABLISHED`.

CONCLUDED If the contract reaches its duration, the contract is successfully completed its and contract state is set to `CONCLUDED`.

RESCINDED The contract may me rescinded by one of the parties thus the contract state is set to `RESCINDED`.

Request Life-Cycle The request life-cycle holds the following states as depicted by Figure 3.2.

NEW A just-arrived request implies the creation of a request whose state is `NEW`.

BEING_TREATED If the request can be treated, its state is set to `BEING_TREATED` thus dispatching the request for treatment.

ABORTED Requests may be aborted either just after its creation or during its treatment. For both situations, the request state is set to `ABORTED`.

TREATED If the request is successfully treated, its state is set to `TREATED`.

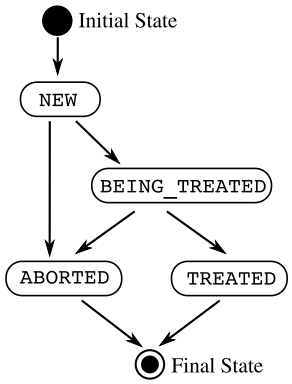


Figure 3.2: Request life-cycle.

Job Life-Cycle The job life-cycle is based on the SAGA [GJK+08] job state model as depicts Figure 3.3. The job life-cycle holds the states described next.

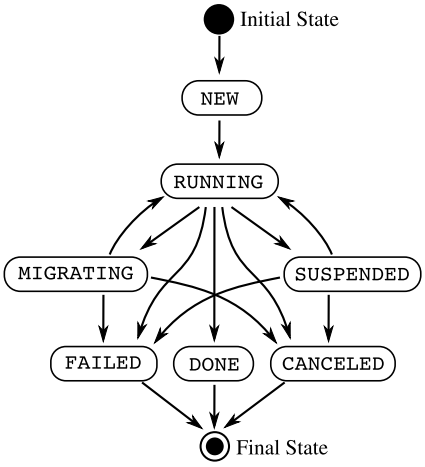


Figure 3.3: Job life-cycle.

NEW A job has the `NEW` state when it is instantiated.

RUNNING When the job is submitted through to be executed, its state is set to `RUNNING`.

MIGRATING Jobs can be under a migration operation which migrates a job to another resource. This implies changing the job state to `MIGRATING`.

SUSPENDED During job execution, jobs can be suspended then their state are changed to `SUSPENDED`.

FAILED If the job fails, it gets the final state `FAILED`.

DONE If the job successfully finishes, it gets the final state `DONE`.

CANCELED The job execution may be interrupted; thus the job gets the final state `CANCELED`.

3.2 Profit as a Goal

A common challenge in the Service-Centric Paradigm (see Chapter 1) is to manage the execution of services. Service execution may still include distributed infrastructures which brings complexity to service execution management. Additionally, service execution should be guided by high-level guidelines defined by SLAs while addressing to increase the provider profit. Dealing with service execution management by taking into account these issues is hard and requires significant effort from service developers and administrators.

The main goal of this thesis is twofold: **to provide an autonomous service execution management while aiming at increasing the provider profit**. In order to achieve this goal, this thesis is inspired by autonomous systems (cf. Section 1.4) which are guided by high-level guidelines. Hence it proposes a self-managed support for service providers whose highest-level guideline is to increase their profit. Moreover, the means for increasing the provider profit are based on heuristics owing to the high cost optimal solutions.

In order to let clear the objective of this thesis, service execution is defined as follows.

Definition 3.5 *Service execution is a set of actions which enables the service provider to address the contract life-cycle thus delivering the service in accordance to its agreed terms.*

The set of actions used by Definition 3.5 involves interactions between the service customer and provider as well as further interactions between the service provider and other entities related to the underlying infrastructure. These actions are organized in four groups as following described:

Negotiation Enables the negotiation of contracts represented by the SLA which holds details as the description, terms, penalties, qualities and so forth.

Instantiation Enables instantiating the service on the underlying infrastructure based on contract resource requirements.

Provision Enables delivering the actual service by means of responding to customer requests in agreement to quality specifications.

Termination Enables terminating the service instance.

This thesis addresses these four aforementioned actions by means of *an autonomous support for service execution management*. In order to deal with the goal of increasing the service provider profit, this thesis enhances the previous actions with further complementary actions that address economic aspects. These complementary actions are described next.

Reducing infrastructure costs Resource acquisition relies on under-provisioning as means for reducing costs on infrastructure. Moreover, resources are acquired and released according to contract durations.

Preventing SLA violations Request treatment relies on fault-tolerant and performance QoS assurance mechanisms. These mechanisms are guided by QoS metrics in order to meet the agreed SLA.

Rescinding contracts If the infrastructure cannot provide more resources, the provider may rescind on-going contracts if such an action implies profit augmentation.

In a brief, Figure 3.4 illustrates the actions that enable to address the service execution according to Definition 3.5. The latter complementary actions precisely aim at economic aspects by positively contributing to increase the provider profit. Ultimately, by preventing SLA violations on top of QoS assurance mechanisms is also useful for improving the service provider reputation. However, reputation is beyond the scope of this thesis which focus on increasing the service provider by the previously mentioned means.

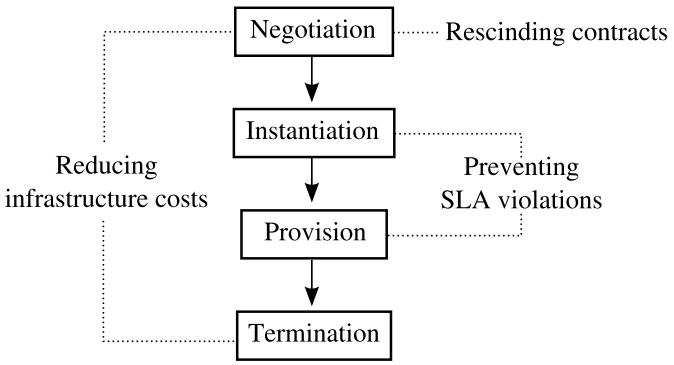


Figure 3.4: Actions that address the contract life-cycle along with complementary actions which aim at increasing the provider profit.

3.3 Creation of Contract Templates

In order to enable the negotiation between service provider and customers, this thesis leverages contract templates. It assumes that the service provider provides contract

templates which are chosen by customers in order to initiate the negotiation process. This section then explains how contract templates are created. In Section 3.3.1, labels are used to customize contract templates based on their QoS metrics. Following that, Section 3.3.2 explains how QoS metrics are translated to system configurations which are used to configure QoS assurance mechanisms. Finally, economic aspects about contract templates are exposed in Section 3.3.3.

3.3.1 Contract Template Labels

This thesis relies on a simple negotiation protocol: costumers choose a contract template, set its time interval and propose it to the service provider. If the provider agrees on the contract proposal, a contract is established. Otherwise, the contract proposal is rejected. In order to realize such a protocol, this thesis relies on *contract templates* whose QoS metrics are predefined by the service provider. Specifically, contract templates are customized based on QoS metrics related to performance and fault-tolerant aspects as explained next.

Response Time Means the maximum amount of time that a request treatment can take.

Reliability Refers to a degree of dependability.

In order to facilitate the identification of contract templates, this thesis names contract templates with labels which emphasize their main quality. The main quality of a contract template is defined based on the trade-off between performance and fault-tolerance (cf. Section 1.2.3). Based on this trade-off, labels are defined by combining different values of response time and availability QoS metrics. Moreover, QoS metrics values are expressed by the following high-level constraints: *strong, medium,weak* [RFCRJ04]. The contract templates, their labels and QoS metrics constraints are stored in the *QoS table*. The QoS table is depicted by Table 3.1 which presents four contract templates whose labels are fast, safe, classic and standard. As a result, the service provider is able to support differentiated contract templates which are easily identified by labels.

QoS Aspects	**Performance**	**Fault-Tolerance**
QoS Metrics / Contract Template Labels	**Response Time**	**Reliability**
Fast	*strong*	*weak*
Safe	*weak*	*strong*
Classic	*medium*	*medium*
Standard	*weak*	*weak*

Table 3.1: The QoS table stores information about contract templates and their respective QoS metrics constraints. Labels are used to name the contract templates which are customized based on the trade-off between performance and fault tolerance.

3.3.2 QoS Translation

In order to build mechanisms that actually ensure the quality properties, QoS metrics should be translated to low-level means understandable by the service execution environment. Therefore, *QoS translation* means to translate QoS metrics constraint to the right configurations which enable the system to deliver such QoS metrics constraint. The QoS translation is represented by the generic function τ as depicts Equation 3.1 where *qos* means a QoS metrics constraint and sys_config means a system configuration. Inversely, it is possible to know which QoS metrics constraint a minimal[1] system configuration is able to meet as depicts the generic function τ' in Equation 3.2. Thus τ' represents the interpretation of a minimal system configuration to which QoS metrics constraint it is able to ensure.

$$\tau(qos) = sys\_config \tag{3.1}$$

$$\tau'(sys\_config) = qos \tag{3.2}$$

This thesis designs QoS assurance mechanisms that ensure response time and reliability (cf. Section 3.4). Response time is ensured by acquiring resources and instantiating the service provider based on the right resource requirements. On the other hand, reliability is ensured by replacing failed and delayed jobs during request treatment. Thereby, the low-level configuration required by the QoS assurance mechanisms are *resource requirements* and *job replacement thresholds*. In order to configure the QoS assurance mechanisms, the function τ (cf. Equation 3.1) is used based on the QoS metrics constraints described in the QoS table (cf. Table 3.1 in Section 3.3.1). Thus the Equations 3.3 and 3.4 represent the translation of the QoS metrics response time and reliability respectively; where $constraint \in \{strong, medium, weak\}$ and $failure\_th_{constraint}, delay\_th_{constraint}$ are the replacement thresholds.

$$\tau(resp\_time_{constraint}) = res\_req_{constraint} \tag{3.3}$$

$$\tau(reliability_{constraint}) = (failure\_th_{constraint}, delay\_th_{constraint}) \tag{3.4}$$

The previous translation function τ is used to quantify a given QoS metrics constraint. This thesis considers different methods for quantifying reliability and response time. Reliability is quantified by defining the failure and delay thresholds statically. On the other hand, response time is quantified by profiling the service provider with various resource requirements. During the profiling, the *actual response time* and the *job execution time* metrics are gathered. These metrics are following described.

$actual\_resp\_time_{q_k}$ **Actual Response Time** Means the total amount of time to treat a request q_k.

[1] Minimal system configuration is preferred in order to discard further configurations which achieve the same QoS metrics constraint by implying a greater operational cost.

$exec\_time_{j_i}$ **Job Execution Time** Means the total amount of time to execute a job j_i belonged to request q_k.

As the fault-tolerance QoS assurance mechanism adds operational overhead to the request treatment, both profiling metrics are used to calculate the response time QoS metrics as depicts Equation 3.5. Thereby, response time constraints (i.e., strong, medium, weak) are represented by *resp_time* whose whose translation is the resource requirements used during the profiling of *resp_time*.

$$resp\_time = actual\_resp\_time_{q_k} + failure\_th \cdot exec\_time_{j_i} + delay\_th \cdot exec\_time_{j_i} \quad (3.5)$$

3.3.3 Pricing

This section gives details about how the price of the service is formed, how payments are realized and further aspects about the pricing model used by this thesis. In order to define the service price and assesses the accounting, the operational costs are used in a pay-per-use fashion. In addition, fines for aborting requests and rescinding contracts are defined and considered when calculating the net general profit of the service provider.

Price Formation This thesis defines the service price in a *cost-based* manner, i.e., the service price is defined based on its operational costs (cf. Section 1.3.3). Thus the profit actually means how much the provider wants to earn from the given contract. The Equation 3.6 defines the formation of the price ρ for a contract proposal whose label is l and duration is t. Hence the price is defined as a function of its expenses ϵ plus its targeted profit Π.

$$\rho(l,t) = \epsilon(l,t) + \Pi(l,t) \quad (3.6)$$

Moreover, because the price fluctuates according to its expenses, setting the price as a function of the expenses becomes interesting since it enables guaranteeing the targeted profit margin. In addition, the resource acquisition approach proposed by this thesis acquires resources according to customers demand (cf. Section 3.5). Thus, the cost-based price positively influences the provider profit because operational costs only exist if contracts are established.

Accounting Assessment This thesis leverages the pay-per-use accounting assessment by charging the customer according to the contract duration given a contract template label. The time dimension is used in order to keep the approach generic as time is a common metric for most type of services. Although further specific metrics such as number of requests, concurrent usage and key performance indicators are not considered, they can perfectly be taken into account when leveraging the approach proposed by this thesis. Additionally, the pay-per-use is chosen owing to the possibility of customizing it as a subscription plan (cf. Section 1.3.2), thus covering both accounting assessments.

Fines In order to define the fines, this thesis relies on the following assumption.

Assumption 3.3 *Penalties owing to SLA violations are payed by means of monetary terms.*

The straight advantage of Assumption 3.3 is that it allows the service provider to positively distinguish itself from its competitors which compute fine payments based on service credits (cf. Section 2.1). Because service execution depends on dynamic factors both from the environment and the service behavior itself, it is very hard to ensure any quality of service in such circumstances. This scenario does not encourage service providers to ensure QoS mainly when the performance aspect is addressed. However, the conception of novel approaches encourages or even requires novel metrics. As this thesis does conceive QoS assurance mechanisms, it enables the service provider to pay fines in currency without compromising its profit

Two types of fines are defined as following described.

Request Abortion Fine ψ : refers to the fine that should be payed owing to non-treatment of a request. This penalty should be then applied for requests that reach the state `ABORTED`.

Contract Rescission Fine Ψ : refers to the fine that should be payed in case of contract rescission, i.e., contracts that reach the state `RESCINDED`.

Profit The price of the contract and the fines are used to calculate the net general profit of the service provider. Basically, it is used Equation 3.6 to deduce that $\Pi(l,t) = \rho(l,t) - \epsilon(l,t)$ and then the fines are taken into account. Thus the Equation 3.7 defines P as the net general profit of the service provider during the time interval $[t,t']$ for each contract c_i the service provider held in $[t,t']$. In Equation 3.7, ρ_i represents the price of c_i, ϵ_i represents the expense of c_i, Ψ_i represents the contract rescission fine if the contract was rescinded by the provider, Ψ'_i represents the contract rescission fine if the contract was rescinded by the customer, and $\psi_{i,j}$ represents the request abortion fine of the j-th request fine of c_i.

$$P_{[t,t']} = \sum_{i=0}^{n} \rho_i - \sum_{i=0}^{n} \epsilon_i - \sum_{i=0}^{n} \Psi_i + \sum_{i=0}^{n} \Psi'_i - \sum_{i=0}^{n} \sum_{j=0}^{m} \psi_{i,j} \tag{3.7}$$

3.4 QoS Assurance

In order to enable service instantiation and the provision, this thesis relies on underlying mechanisms which deal with QoS assurance. Indeed, preventing SLA violations minimizes penalties. If requests are not treated successfully, fines are payed to customers which contributes to decrease provider profit (cf. Equation 3.7). In order to prevent this situation, this thesis proposes QoS assurance mechanisms which handle performance and fault-tolerance in order to ensure that service provisioning will comply to the agreed SLA. On the one hand, performance response time QoS is ensured

by executing requests with the right resource requirements that are able to achieve the targeted performance. On the other hand, fault-tolerance qualities are ensured by reacting to job misbehaviors which include job failures and delays. Both performance and fault-tolerant QoS assurance mechanisms are explained in Sections 3.4.1 and 3.4.2 respectively.

3.4.1 Performance

The goal of the performance QoS assurance mechanism is to ensure that the service instance is configured to meet a given response time. Therefore, performance is guaranteed by properly configuring the service instance. As instance configuration means resource requirements, the response time is translated to resource requirements. Then, the service instance treats customer requests based on the right resource requirements which meet the agree response time. Thereby, performance goals are achieved through service instance configuration at runtime.

The instance configuration can be done when the service is instantiated or each time a request is sent to this instance. The former way is more interesting for service instances which are based on contracts that do not have their QoS changed during the service provisioning. On the other hand, configuring the service instance each time it receives a request allows changing contract QoS during service provisioning. However this adds unnecessary data overhead to request data as well as it requires configuring the service instance for each request the instance receives. Therefore, this thesis relies on configuring the service instance at its deployment time since no change on QoS is considered during service provisioning.

The Figure 3.5 depicts the configuration of a service instance at deployment time. When a customer proposes a contract, the service provider translates the QoS metrics to its respective resource requirements based on information stored in the QoS table (cf. Equation 3.3 in Section 3.3.2). Then the service provider acquires resources from the infrastructure according to the previous resource requirements. These requirements are also used by the service provider to configure and deploy a service instance for supporting this customer requests. 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.

3.4.2 Fault Tolerance

This thesis considers that system faults mean job misbehaviors while system failures mean request abortions. This section introduces two mechanisms which aim at overcoming job misbehaviors in order prevent requests to be aborted. In this context, *job misbehaviors* are job failures or job delays as explained next.

Job Failure A *job failure* is a crash fault of the job process or one of the job processes in case of multi-process jobs. This crash may occur due to a non-successful I/O operation for instance.

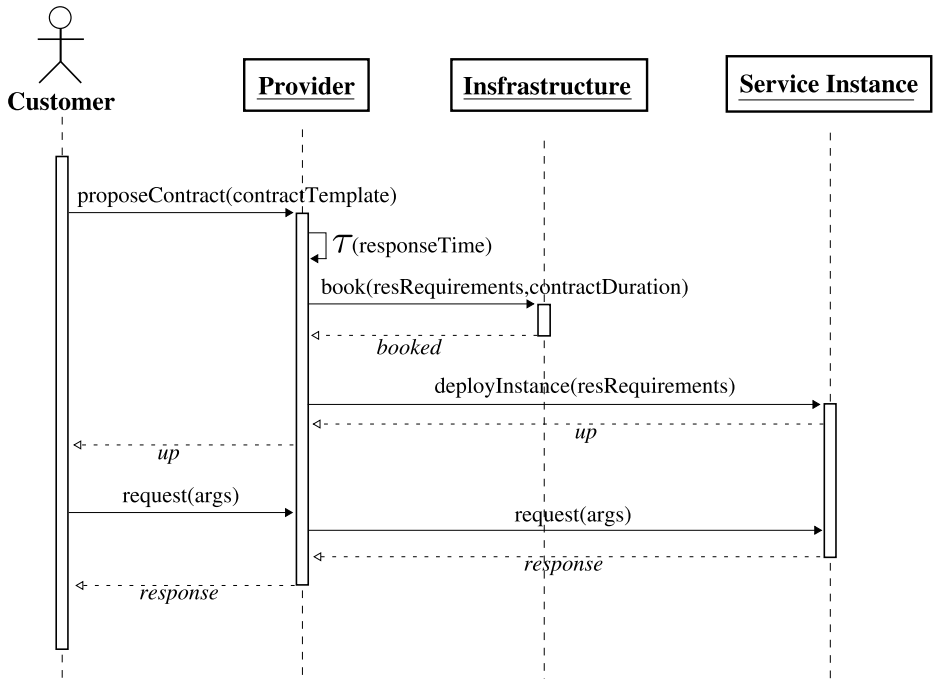


Figure 3.5: Sequence diagram of service instance configuration at deployment time followed by a request treatment. The service provider is able to achieve the targeted response time based on the resource requirements to which it was configured previously.

Job Delay A *job delay* is identified when the job elapsed time $elapsed\_time_{j_i}$ exceeds its expected execution time $exec\_time_{j_i}$. This situation may mean that either the job is correct and will finish eventually or the job is failed and will never finish owing to a software malfunction for instance. This thesis discards the former assumption and assumes that if a job is delayed, then it is failed. This pessimist choice is assumed since it is not possible to differentiate a slow process from a faulty process. Moreover, since the service provider relies on a time-constraint request treatment, it is more suitable to let it assume that a delayed job is failed for performance purpose. The drawback of this assumption is that it does not take advantage of delayed jobs that are up to finish. However, it can be easily circumvented by increasing the value of $exec\_time_{j_i}$ for instance.

Fault-Tolerance Algorithms This thesis proposes the Algorithms 1 and 2 which replace failed and delayed jobs respectively. Both algorithms rely on job dynamic metrics by reacting to events that inform about job failures and delays. The variables used by these algorithms are previously explained in Section 3.1.2. Moreover, the variable $adapt\_th_{j_i}$ represents the adaptation threshold, i.e., the maximum time allowed

until when an adaptation action can be triggered. The $adapt\_th_{j_i}$ is then calculated by considering the profiled job execution as time the following equation:

$$adapt\_th_{j_i} = resp\_time_{q_k} - exec\_time_{j_i} \tag{3.8}$$

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** $elapsed\_time_{j_i} > exec\_time_{j_i}$ **and** $elapsed\_time_{q_k} < adapt\_th_{j_i}$ **and** $n \leq delay\_th_{j_i}$ **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 could be used by the aforementioned Algorithms 1 and 2 which include job replication, migration, or checkpointing. However, replicating jobs implies more costs to the service provider, it is then preferable to rely on mechanisms which do not require further resources. On the other hand, migration and checkpointing are complex to be conceived in a distributed environment without compromising performance accuracy.

3.5 Resource Acquisition and Allocation

While the previous Sections 3.3 and 3.4 address service aspects related to contract template labels and QoS assurance mechanisms, this section addresses how resources are acquired and allocated aiming at decreasing the costs on infrastructure usage.

Indeed, resource acquisition is a crucial aspect of service execution management as it defines how resources are acquired by the service provider. A trivial resource acquisition approach is to rely on a fixed number of acquired resources to satisfy future contracts and only accepting contracts if their resource demand does not exceed the total amount of acquired resources. However, it is not an efficient solution since it wastes resources in a low-demand scenario and limits the number of contracts according to the predefined fixed number of acquired resources. In contrast, resources can be acquired on-demand, i.e., according to customer demand at runtime. This approach is more suitable to service providers since they can fit they resource acquisition to the actual resource demand. Therefore, this thesis relies on a dynamic resource acquisition approach which acquires and releases resources according to contract constraints.

Furthermore, service providers may take advantage of under-provisioning in order to increase their profit. Under-provisioning is applied by this thesis and is useful when customer request demand is often not the maximum allowed. Therefore there will certainly have idle resources which can be used by other requests. However, service under-provisioning eventually requires dealing with the lack of resources when allocating resources to requests. This implies SLA violations thus implying the payment of fines. In order to deal with concurrent resource allocation, the approach proposed by this thesis decides which request should be aborted aiming at minimizing fine payments.

Resource acquisition and allocation are dealt with in an event-driven fashion. While new contract proposals trigger the acquisition of resources, new requests trigger resource allocation. Sections 3.5.1 and 3.5.2 discuss the flowcharts of contract proposals and request arrivals respectively.

3.5.1 Contract Proposals

Figure 3.6 depicts the flowchart of a contract proposal. When a contract establishment is proposed, the provider should decide whether it will accept it or not. First, it tries to acquire the needed resources to support the contract. If this operation is successful, then it deploys the service instance on the infrastructure and accepts the contract. Otherwise, it decides whether it is more profitable to rescind an on-going contract in order to accept the just-arrived contract proposal. If so, it rescinds the chosen established contract and accepts the contract proposal, else it rejects the contract proposal.

This thesis proposes a simple, dynamic and efficient way of acquiring resources which is driven the contract proposals. The Algorithm 3 depicts how resources are acquired given a contract template label l, the contract duration t and the under-provisioning factor upf which represents the percentage of resources to acquire based on resource requirements of l. First, *acquire* algorithm translates the response time

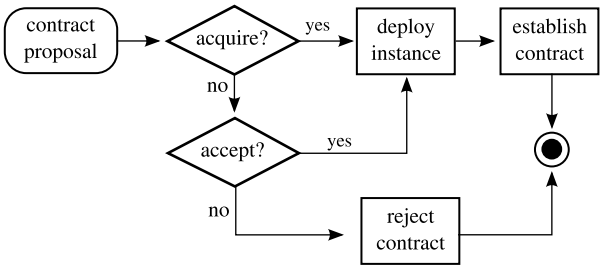


Figure 3.6: Flowchart of contract proposals.

to the resource requirements res_req_l by using the translation function τ (cf. Equation 3.1, Section 3.3.2). Second it applies the reduction factor through the operation *reduce*. Finally, the *acquire* algorithm acquires resources from the resource provider on-demand through the *book* operation. Thereby, the service provider is able to take advantage of under-provisioning in order to increase its profit according to its business model. Moreover, the *acquire* algorithm is configurable in such a way that its under-provisioning characteristic can be disabled by setting $upf = 100\%$, which means to acquire the exact resource requirements of l.

Algorithm 3 acquire(l, t, upf)

Require: The contract template label l.
Ensure: Returns true if upf percent of l's resource requirements is acquired.
1: $res\_req_l \leftarrow \underline{\tau}(resp\_time_l)$
2: $res\_req_l \leftarrow \underline{\text{reduce}}(res\_req_l, upf)$
3: **return** $\underline{\text{book}}(\ res\_req_l, t\ )$

The *accept* operation described in Figure 3.6 relies on Algorithm 4 in order to decide whether the contract proposal will be accept when no more resources can be acquired from the infrastructure. This algorithm compares the amount of money to be earned from the contract proposal with the fine to be payed owing to contract rescission. If the total amount of money to be earned is greater, the *accept* operation rescinds an `ESTABLISHED` contract and returns true. Otherwise, it returns false.

Algorithm 4 accept(c)

Require: a contract proposal c
Ensure: Returns true if there is an on-going (`ESTABLISHED`) contract c' whose resource requirements are enough to accept c and the profit for accepting c is greater than the fine for rescinding c' minus the profit of c'.

```
1: c' ← nil
2: for each c_i ∈ C^E do
3:    if (c_i resource requirements assist c) and (Π_c > Ψ_{c_i} − Π_{c_i}) then
4:       if c' = nil then
5:          c' ← c_i
6:       else
7:          if Ψ_{c'} > Ψ_{c_i} then
8:             c' ← c_i
9:          end if
10:      end if
11:   end if
12: end for
13: if c' = nil then
14:    return false
15: else
16:    rescind c'
17:    return true
18: end if
```

3.5.2 Request Arrivals

The flowchart of a request arrival event is depicted by Figure 3.7. If the provider does not rely on under-provisioning, the request treatment is dispatched normally since there are enough resources to treat it. Else, the provider should check if there are available resources to treat the request. If the resource reliability fits the request resource requirements, the request is treated. Otherwise, the request is aborted implying an SLA violation.

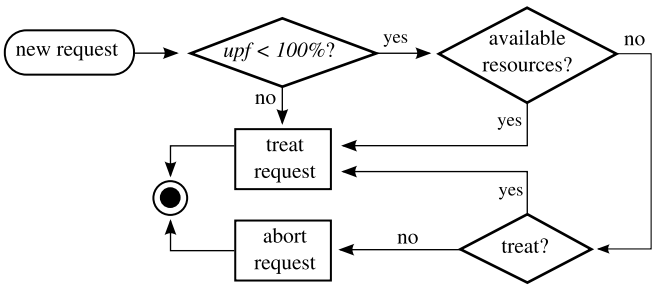


Figure 3.7: Flowchart of request arrivals.

Thus the problem is to decide which request will be aborted: the just-arrived request or another on-going request whose resource requirements are suitable to support the former request. As the provider aims at decreasing fine payments, it chooses which

request will be aborted based on request fine abortion values. Then the request to be aborted will be chosen aiming at minimizing the payment of fines as depicts Algorithm 5. The Algorithm 5 returns true if it managed to find an on-going request (`BEING_TREATED` state) whose resource requirements support the just-arrived request and whose fine is the cheapest.

Algorithm 5 treat(q)

Require: the arrival of request q
Ensure: Returns true if there is an on-going (`BEING_TREATED`) request q' whose resource requirements are enough to treat q and the fine for aborting q' is the cheapest.

```
1: q' ← nil
2: for each q_k ∈ Q^T do
3:    if (q_k resource requirements assist q) and (ψ_q > ψ_{q_k}) then
4:       if q' = nil then
5:          q' ← q_k
6:       else
7:          if ψ'_q > ψ_{q_k} then
8:             q' ← q_k
9:          end if
10:      end if
11:   end if
12: end for
13: if q' = nil then
14:    return false
15: else
16:    abort q'
17:    return true
18: end if
```

Part III

Implementation

Chapter 4

Qu4DS

The previous chapter presented the design of a solution for automatic service execution management which targets increasing the service provider profit. This current Chapter introduces the Qu4DS (Quality Assurance for Distributed Services) framework which implements this solution. Qu4DS addresses all the actions that enable addressing the contract life-cycle and the further complementary actions that deal with economic aspects. Hence, Qu4DS implements an automatic support for services. The development aspect is addressed by means of providing a PaaS (Platform-as-a-Service) support which fills the gap between the conception of higher-level SaaS (Services-as-a-Software) and the underlying infrastructure on which they are executed. Moreover, Qu4DS also assists the administration of the developed service by allowing service administrators to define high-level directions that will guide the service execution management in an automatic and transparent fashion.

Qu4DS features include contract negotiation, deployment and management of the service instance on the infrastructure and QoS assurance. These aforementioned features are driven by the contract quality aspects and duration by aiming at increasing the provider profit. Furthermore, Qu4DS supports the development of service providers that leverage the Master/Worker pattern. Qu4DS assists the development of such services by freeing service developers from managing workers and by ensuring their proper execution in accordance with time constraints QoS and reacting to job failure and delays at runtime.

The reminder of this chapter is organized as follows. Section 4.1 introduces the context of the targeted applications supported by Qu4DS. In order to investigate how to facilitate the development of these applications, the relationship between distributed applications and infrastructures is explored by Section 4.2. As follows, Section 4.3 describes Qu4DS architecture as a PaaS support for easing the development of service providers. Further details about this support are exposed in Section 4.4. Ultimately, Qu4DS usage is explained in Section 4.5.

4.1 Support for Master/Worker Applications

Some applications require great processing capacity owing to their intrinsic characteristics. For instance, applications which involve geoprocessing, photogrammetry, scientific simulations, huge data compressing take too much time to be executed in desktop computers. Indeed, they require non-trivial computational power to accomplish their tasks such as super-computers. However, super-computers are expensive infrastructures and running these applications on a single machine increases reliability issues.

Low-cost distributed computational infrastructures may be used to circumvent this problem. On the one hand, a set of desktop computers connected through a network provides together enough computational power to assist the requirements of these applications. In addition, they inherently provide a distributed architecture which enables to develop reliable applications. On the other hand, the drawback of this approach is that it is harder to develop distributed applications as a result of the complexity of managing distributed data and processes.

Master/Worker is a generic pattern for conceiving distributed applications. It relies on a *master* abstraction that coordinates other tasks executed by *workers*. When workers finish their tasks, the master may delegate another task to them until the end of the application execution. The simple design of Master/Workers pattern is an advantage for developers against complex distributed workflows for instance. Therefore, this thesis targets supporting applications that are based on the Master/Worker pattern.

4.2 Foundations

This thesis proposes a framework for supporting Master/Worker service providers which are built on top of distributed infrastructures. This requires investigating how to enable high-level service execution aspects as negotiation and provisioning by leveraging distributed infrastructure interfaces. However, these latter interfaces provide lower-level abstraction as jobs and resources as well as they often address job and resource concerns in a tight-coupled manner. This section discusses how Qu4DS decouples job management from resource acquisition and how it fills the gap between low-level underlying infrastructure and high-level service-oriented interfaces.

4.2.1 Separating Resource Acquisition from Job Management

The dynamic way of how this thesis deals with resource acquisition requires decoupling it from the management of distributed jobs. While resource are acquired driven by contract arrival events, job management are mainly driven by request arrival events. Therefore, the management of both jobs and resource acquisition should be able to be dealt with separately. This requirement implies understanding distributed infrastructure interfaces along with their set of operations and abstractions.

On the one hand, grid interfaces define an extensive set of functionalities [JMF09, CFJ+08, Fos06, GJK+05] by leveraging the job abstraction (cf. Section 1.6). For instance, the Simple Grid API (SAGA) [GJK+05] interface proposes to ease the use of

grids, promote interoperability and standardization while meeting grid applications requirements. In spite of these achievements, grid interfaces deals with the concerns of job management and resource acquisition together and requires static resource configuration and acquisition.

On the other hand, current interfaces of cloud IaaS providers rely on decoupled and simple resource provisioning interfaces [ME11, Ama06a, Nur09, Bég08] as discussed in Section 1.7.2. These interfaces are limited to the provisioning of bare resources, typically virtual machines, with no support for high-level programming abstractions, such as jobs. Despite the limitation of IaaS interfaces, further cloud-oriented approaches in the context of PaaS and SaaS take into account higher-level abstractions as support for MapReduce applications [DG08], workflows [Inf11] and application frameworks [Goo08, Sal99]. However, while MapReduce and workflow supports are limited to specific applications, application frameworks impose non-standard interfaces which compromises application interoperability and portability; thus enabling applications to vendor lock-in.

In order to take advantage of jobs by tackling resource acquisition separately, this thesis proposes to leverage the IaaS cloud decoupled way of acquiring resources and the grid standard interfaces which uses jobs as programming abstraction. The separation of resource acquisition from job management is similar to applying pilot-jobs for clouds, e.g., SAGA big-job[LLJ10], which is useful for acquiring resources dynamically. The proposed interfaces which address resource acquisition and job management are explained next.

Infrastructure Management Interface The *infrastructure management* interface addresses resource acquisition where resources mean physical or virtual resources. The infrastructure interface exposes a set of operations on top of the resource abstraction. These operations enable resources to be acquired, released or modified based on specific resource requirements. Listing 4.1 describes the infrastructure management interface where resource class is a short term for referring to a set of statical resource requirements.

```
1 public interface InfrastructureManagement {
2   public List<InfraResourceType> getResourceTypes();
3   public int getNumberOfAvailableResources(String resourceClass);
4   public int reserve(int nOfResources, String resourceClass,String
        startTime, String endTime);
5   public boolean resize(int reservationId, String resourceClass, int
        newNumberOfResources, String endTime, List<Integer> resourcesId);
6   public List<InfraResource> getReservedResources(int reservationId);
7 }
```

Listing 4.1: Description of the infrastructure management interface in Java syntax.

Moreover, the infrastructure interface can be implemented on top of existing IaaS clouds which leverage the OCCI [ME11] specification (cf. Section 1.7.2) for instance. Thereby, OCCI instances can be mapped to `resources` in such a way that the *start*

OCCI action can be implemented by as the `reserve(int nOfResources, String resourceClass,String startTime, String endTime)` method where the `startTime` parameter should be set to the current time. In turn, the *stop* OCCI action can be mapped to the `resize(int nOfResources, String resourceClass,String startTime, String endTime)` method where the `nOfResources` parameter should be set to zero.

Job Management Interface The *job management* interface leverages jobs as higher-level abstraction. Jobs are useful for managing service instances and its distributed tasks on top of acquired resources. This thesis defines the job management interface based on SAGA in such a way that it is a subset of SAGA. First, this interface leverages the SAGA job life-cycle in order to define the job life-cycle (cf. Figure 3.3 in Section 3.1.3). Then, it provides some operations in the scope of SAGA job management which includes creating, canceling, migrating as well as enabling callbacks for monitoring purpose. Moreover, this interfaces also assumes that jobs share a common data space which may be a distributed file system, a database, a distributed shared memory and so forth. This decision is explained by the very fact that simplicity is prioritized against complexity. However the proposed interface is by far not trivial which allows it to meet several application development requirements.

The job management interface is described by Listing 4.2.

```
1 public interface JobManagement {
2   public InfraJob createJob(int reservationId, InfraJobDescription
        jobDescription);
3   public boolean runJob(int reservationId, InfraJob job, String
        resourceAddress);
4   public boolean cancelJob(int reservationId, InfraJob job);
5   public boolean cancelAllJobsOnResource(int reservationId, resource);
6   public int checkpointJob(int reservationId, InfraJob job);
7   public boolean suspendJob(int reservationId, InfraJob job);
8   public boolean resumeJob(int reservationId, InfraJob job);
9   public boolean resumeJob(int reservationId, InfraJob job, int
        checkpointVersion);
10  public boolean migrateJob(int reservationId, InfraJob job);
11  public boolean migrateJob(int reservationId, InfraJob job, String
        resourceAddress);
12  public void registerCallback(int reservationId, InfraJob job, String
        metric, boolean on, Observer observer);
13  public List<InfraJob> getAllJobs(int reservationId);
14  public List<InfraJob> getJobs(int reservationId, String jobState);
15  public List<InfraJob> getJobsOnResource(int reservationId, String
        resourceAddress);
16 }
```

Listing 4.2: Description of the job management interface in Java syntax.

Service Negotiation and Provisioning The previous sections described two interfaces which enable dealing with the underlying infrastructure. In turn, this section

describes the *negotiation and provisioning* interface which addresses the interaction between Qu4DS and the higher-level service-oriented layer. The negotiation and provisioning interface leverages the service abstraction by enabling the service provider to communicate with customer services. Through this interface, contracts are negotiated and customers send requests. The idea is to enable the provider to negotiate contracts rather than to define a complete specification or protocol for SLA negotiation, such as WSLA [LKD+03] or WS-Agreements [ACD+07].

In order to tackle service communication and membership, Qu4DS assumes that there exists a service registry which acts as repository of services. Then service providers subscribe to the registry from which service customers can look for providers based on their both functional and non-functional requirements. After selecting the service provider that fits its needs, the customer initiates a negotiation process which may conclude in a contract establishment if both parties agrees on the contract terms. This interaction between service providers and customers is commonly used by service-centric approaches owing to its simplicity. On the other hand, it has some drawbacks such as relying on a central service registry, not natively taking into account semantic service discovery, as well as further details that concern service communication protocols and specifications. For the sake of implementation, Qu4DS relies on a central service registry, however the way its architecture is designed allow Qu4DS to be easily coupled to other solutions that tackles service registry issues.

The negotiation and provisioning interface is described by Listing 4.3. While the `List<SLA> getListOfSLATemplates()` and `SLA proposeContract (SLA contractTemplate)` enable negotiation, the `String request(List<String> args)` method addresses provisioning by allowing customers to send requests. It is important to remark that further service provisioning methods can straightly be added to this interface according to service provider functional requirements.

```
1 public interface NegotiationProvisionsing {
2   public List<SLA> getListOfSLATemplates();
3   public SLA proposeContract(SLA contractTemplate);
4   public String request(List<String> args);
5 }
```

Listing 4.3: Description of the negotiation and provisioning interface in Java syntax.

The Figure 4.1 depicts a sequence diagram of both negotiation and provisioning processes based on the negotiation and provisioning interface. With respect to the negotiation, Qu4DS relies on a template-based contract negotiation. It assumes that the customer obtains contract templates from the provider, chooses the contract template that is more suitable to its requirements, sets the contract duration and finally proposes it to the service provider. In turn, the provider decides whether it will accepted. If the service provider accepts the contract proposal, the customer is able to send requests which are treated by the service provider. Furthermore, in Qu4DS current implementation, the service provider is in charge of auditing the provision and

monitoring penalties. However service auditing could be performed by a third neutral party.

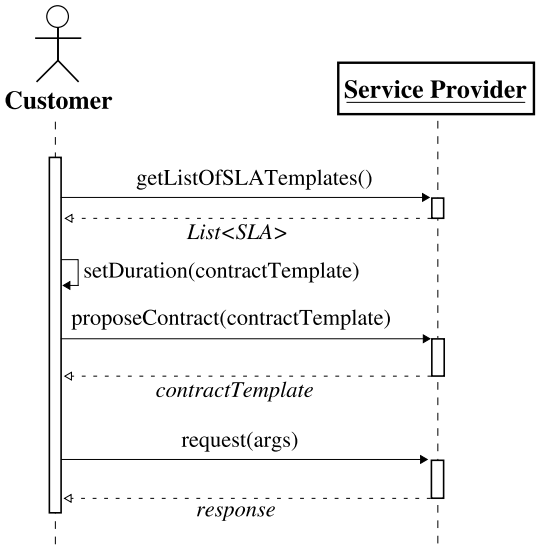


Figure 4.1: The negotiation and provisioning interface methods exposed in a sequence diagram which depicts the interaction between a customer and the service provider.

4.2.2 Assumptions

Qu4DS is implemented by taking into account some assumptions which concerns the following aspects.

Resources

Assumption 4.1 *Resource failures are not handled.*

The explanation of relying on Assumption 4.1 is that Qu4DS aims at overcoming job failures and delays instead of dealing with resource failures. Moreover, even though the solution presented does not consider resource failures, it can be addressed by conceiving further QoS assurance mechanisms that deals with resource failures, e.g., booking further resources to replace failed resources.

Assumption 4.2 *Resources have the same characteristics.*

The Assumption 4.2 simplifies the experimentation scenarios. Having different types of resources, i.e., resource whose hold different requirements, increases the number of experimentation scenarios. Moreover, this assumption does not limit Qu4DS applicability as infrastructures which deliver homogeneous resources are quite simple to realize on top of current virtualization techniques, e.g., IaaS clouds.

Profit and Expenses

Assumption 4.3 *The expenses $\epsilon(l,t)$ represent the costs spent on resource booking by a contract whose template label is l during the contract duration t.*

In order to calculate the expenses, Qu4DS relies on Assumption 4.3 and defines the expenses as a function of the number of resources $\omega(l)$ required by the label l and the cost $\iota(l)$ of using a single resource that meet l requirements. The following Equation 4.1 depicts how the expenses are calculated.

$$\epsilon(l,t) = t \cdot \omega(l) \cdot \iota(l) \tag{4.1}$$

Assumption 4.4 *The targeted profit $\Pi(l,t)$ is a function of the expenses in such a way that $\Pi(l,t) = \frac{\pi(l)}{100} \cdot \epsilon(l,t)$, where $\pi(l)$ is the percentage that represents the targeted profit for label l.*

Assumption 4.4 is useful for enabling the service provider to rely on various profit margins for each label. Based on Assumption 4.4 and from Equation 3.6 (cf. Section 3.3.3), the following Equation 4.2 defines how Qu4DS calculates the price $\rho(l,t)$ for each contract template label l whose contract duration is t.

$$\rho(l,t) = \epsilon(l,t) \cdot \left(\frac{100 + \pi(l)}{100}\right) \tag{4.2}$$

Fines

Assumption 4.5 *The request abortion fine ψ_{q_k} costs e_3 times the cost that request q_k takes to be treated.*

Based on Assumption 4.5, the fine cost for aborting the request q_k is given by Equation 4.3 where $resp\_time_{q_k}$ is the q_k response time if the request would be successfully treated.

$$\psi(l,q_k) = e_3 \cdot \rho(l, resp\_time_{q_k}) \tag{4.3}$$

Assumption 4.6 *Contracts can only be rescinded by the service provider which implies paying the customer the contract rescission fine Ψ.*

Assumption 4.7 *The contract rescission fine Ψ_i costs e_4 times the price of the contract i.*

According to Assumption 4.7, the fine for rescinding a contract Ψ_i is calculated by multiplying e_4 by ρ_i as exposed by Equation 4.4:

$$\Psi(l,t) = e_4 \cdot \rho(l,t) \tag{4.4}$$

Assumption 4.8 *If a contract i is rescinded, the customer will be only charged based on the contract rescission fine Ψ_i, request abortion fines $\psi_{i,j}$ will not be taken into account.*

4.2.3 Profiling

The Qu4DS monitoring system relies on sensors which periodically inform Qu4DS about job and request dynamic metrics. Monitoring sensors are configured based on a relaxed timeout which directly influences monitoring accuracy. It is not preferable to rely on strong accurate metrics in order to prevent overloading Qu4DS with the arrival of monitoring events. Furthermore, the accuracy of the Qu4DS monitoring system meets Qu4DS requirements since the timeout can be configured by taking into account the targeted accuracy.

Because distributed infrastructures are subject to fluctuations, the dynamism is an inherent characteristic of such an environment. In order to tackle this issue, Qu4DS defines new measures based on the actual measured values of the QoS metrics in order to avoid that dynamic fluctuations compromise the behavior of its QoS assurance mechanisms. Thus, the values of $exec\_time_{j_i}$ and $resp\_time_{q_k}$ are relaxed as depicted by Equations 4.5 and 4.6 respectively. In these equations, e_0, e_1 and e_2 are constants, $mean\_exec\_time_{j_i}$ and $mean\_actual\_resp\_time_{q_k}$ are the respective averages of the repeated measured values of $exec\_time_{j_i}$ and $actual\_resp\_time_{q_k}$ as well as $sd\_exec\_time_{j_i}$, $sd\_actual\_resp\_time_{q_k}$ are their standard deviations.

$$exec\_time_{j_i} = e_0 \cdot mean\_exec\_time_{j_i} + e_1 \cdot sd\_exec\_time_{j_i} \tag{4.5}$$

$$actual\_resp\_time_{q_k} = mean\_actual\_resp\_time_{q_k} + e_2 \cdot sd\_actual\_resp\_time_{q_k} \tag{4.6}$$

4.3 Architecture

Qu4DS serves as a framework for implementing service providers in the scope of the Service-Centric Paradigm. Qu4DS offers a support for increasing the service provider profit by automatically managing resources, service instances and requests according to agreed SLAs. In order to tackle these issues, Qu4DS design abstracts over distributed infrastructures while it assists higher-level service providers in a transparent manner. This section depicts the Qu4DS architecture by firstly explaining how it supports service developers and administrators, then introducing the Qu4DS architecture.

A PaaS Solution for Service Developers and Administrators The way of how Qu4DS design is placed between infrastructures and higher-level service customers is similar to the cloud common architecture introduced in Section 1.7.2. The cloud computing community has relied on a layered architecture for clouds which addresses resources (IaaS), platform (PaaS) and software (SaaS). The cloud common architecture also deals with distributed infrastructures which aim at supporting higher-level service providers. More specific, the PaaS layer offers a support for final service providers (SaaS layer) on top the infrastructure (IaaS layer). Therefore, Qu4DS and the cloud common architecture have a common interest of separating higher-level services from infrastructures resources. Specifically, both Qu4DS and PaaS clouds are placed in an

intermediate level which provides support for conceiving final services by abstracting bare resources.

Thereby, from the clouds point of view, Qu4DS indeed is a PaaS solution which tackles service execution management by including negotiating contracts, deploying and terminating service instances on resources acquired on-demand. However, Qu4DS objectives go beyond a PaaS approach with QoS assurance capabilities. It also addresses a support for service administrators by providing automatic service execution management. Qu4DS leverages the idea of an autonomous system (cf. Section 1.4) which is guided by high-level guidelines that define how the service provider behavior is changed at runtime.

Architecture The Qu4DS architecture is based on the cloud architecture layers as described by Figure 4.2. In the SaaS layer, contracts are negotiated between the service provider and its customers. When leveraging Qu4DS, the provider actually delegates negotiation, service instantiation and provision to Qu4DS. Thus when a contract is proposed, Qu4DS translates the contract QoS through the *QoS Translator* to the resource requirements able to ensure the contract QoS. The translated resource requirements are forwarded to the *resource management* control loop which books resources through the *infrastructure management* interface until the end of the contract duration. Following that, Qu4DS configures a service instance based on the translated resource requirements and deploys in the infrastructure through the *job management* interface. When the service instance is operational, Qu4DS commits the contract agreement to the right customer, who is now able to send requests.

4.4 Implementation

Qu4DS is implemented in Java and contains around 15,500 lines of code. It has been tested on Debian and Ubuntu distributions of the GNU/Linux operating system; however Qu4DS employment in further GNU/Linux distributions is also feasible. Qu4DS requirements include the Sun Java JDK 1.6, support for SSH connections, a distributed file system and the Bash command-line interpreter. Moreover, Qu4DS is a free-software thus allowing the community to perform further experimentations based on customized parameters. In addition, Qu4DS source code is open which allows researchers to directly take advantage of it[1].

This section exposes further information about Qu4DS implementation being organized as follows. Section 4.4.1 discusses the implementation of Qu4DS control loops. Following that, the implementation of the infrastructure, job management and negotiation and provisioning interfaces are described in Section 4.4.2. Ultimately, a sequence diagram is used to explain Qu4DS behavior at runtime in Section 4.4.3.

[1]Qu4DS will be publicly available under the Lesser GPL license in the following web site: http://gforge.inria.fr/projects/quads/

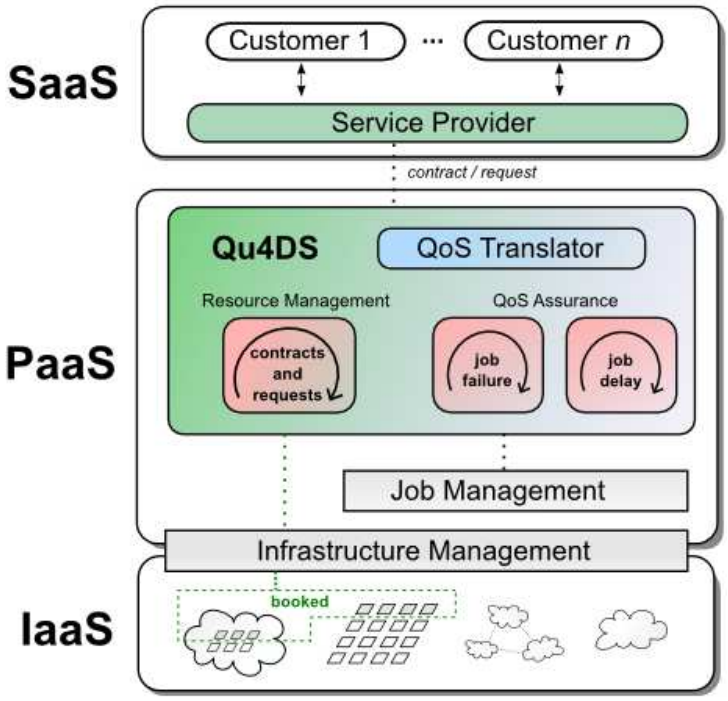


Figure 4.2: Qu4DS architecture in accordance to the cloud common architecture.

4.4.1 Dynamic Adaptation

Qu4DS uses dynamic adaptation in order to change its behavior at runtime. Qu4DS implements three configurable control loops (cf. Figure 4.2) based on an event-condition-action (ECA) decision engine whose actions depend on received events. These control loops are subscribed to communication channels to which events are published. When an event is dispatched, the right control loop receives it and decides whether any action will be employed. Moreover, the ECA decision engine leverages the separation of adaptation concerns as proposed by the Autonomic Computing MAPE control loop, thus decoupling monitoring, decision and adaptation employment.

The configuration of Qu4DS control loops are employed differently. The QoS assurance control loops have their adaptation policies set at runtime, based on the translation of the QoS metrics described in the contract template. On the other hand, the adaptation policy of the resource management control loop depends on how the provider books resources which is defined statically. Moreover, although this difference, both control loop flows depend on the service price and fines in order to take decisions (cf. Section 3.5.1). The resource management and QoS assurance control loops are explained in details next.

Resource Management Control Loop The resource management control loop addresses resource acquisition and allocation. It implements the flowcharts for both contract proposals and request arrivals introduced in Section 3.5 (cf. Figures 3.6 and 3.7). Thus the resource management control loop reacts to two kind of events: either a contract proposal or a request arrival. When a contract is proposed, the resource management control loop books resources to support the contract until the end of the contract duration. When a request is sent by a customer, the resource management control loop checks resource availability to treat this request.

Both previous actions triggered by contract proposals and request arrivals depend on the under-provisioning configuration to which the resource acquisition control loop is configured. The under-provisioning configuration refers to the under-provisioning factor upf as Qu4DS implements the resource acquisition algorithm described by Algorithm 3 in Section 3.5.1. Therefore, different configurations of the under-provisioning factor upf is used to compose different adaptation policies. Moreover, the current Qu4DS implementation applies under-provisioning by means of exploiting horizontal elasticity, i.e., reducing the number of booked resources and not their requirements.

QoS Assurance Control Loops The QoS assurance control loops handle job failures and delays. They are responsible for ensuring that the distributed tasks, i.e. the workers, are successfully executed and finish before the expected execution time. Thus, QoS assurance aspects involve performance and fault tolerance as discussed in Section 3.4. Performance is addressed by Qu4DS by configuring the service instance according to the right resource configuration that allows requests to be treated in the agreed response time QoS (cf. Section 3.4.1). On the other hand, fault tolerance is handled by Qu4DS by reacting to job failures and delays (cf. Section 3.4.2). Moreover, although performance is tackled by service instance configuration, fault tolerance mechanisms complement performance assurance. By handling delayed jobs, the fault tolerance mechanism ensures that delayed jobs will not compromise the request performance.

Qu4DS QoS assurance control loops implement fault tolerant algorithms described in Section 3.4.2. The different adaptation policies of the QoS assurance control loops are based on different values of the replacement threshold parameters of Algorithms 1 and 2. In turn, replacement threshold values depend on the contract template label. Therefore, QoS assurance control loops are configured at runtime by translating the contract template fault tolerant aspects to system configurations (cf. Section 3.3.2).

4.4.2 Interfaces

Infrastructure Interface Qu4DS implements the *infrastructure management* interface according to its description depicted by Listing 4.1, in Section 4.2.1. Such an implementation is simply called as *infrastructure* for naming purpose. The infrastructure leverages Grid'5000 [CCD+05] which was used as the resource provider meeting

the role of the IaaS layer [2] . A Grid'5000 operating system image was customized[3]. In such an image, there are all required programs and libraries to execute Qu4DS including an implementation of the job management interface. Additionally, the Qu4DS current version 0.4.0 supports a single resource class.

As defined by the infrastructure management interface, the infrastructure does not have start and stop operations. Thereby, the interaction between Qu4DS and the infrastructure saves time when booking resources and dismissing them. Importantly, this decision does not compromise the interface, it only assumes that the time for starting an instance is zero. The reason for doing so is owing to the necessity of deploying the Grid'5000 images and configuring them to Qu4DS before using the grid. Since they were previously configured and deployed, it would be useless and time consuming to redo this very same operation. As a result, when a resource is booked, it will be automatically operational and will contain the needed programs installed. Then Qu4DS configuration scripts are in charge of configuring the environment with dynamic information such as IP addresses in order to configure RMI and Web Service remote connections.

Job Management Interface The implementation of the *job management* interface (cf. Listing 4.2, Section 4.2.1) is called *job broker* and follows a layered design. Firstly, the higher-level layer is the actual implementation of the job management interface that deals with job life-cycle state management (Cf. Figure 3.3 in Section 3.1.3). 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. Secondly, the middle layer is implemented by the common `InfrastructureBackend` abstract class that enables the use of various underlying batch-job systems. In addition, the job broker is able to simulate job misbehaviors. This is configurable by means of the percentage of the jobs that will fail as well as the percentage of the jobs that will be delayed. Moreover, these misbehaving jobs are either failed or delayed in a exclusive fashion and they are chosen randomly. Lastly, the final third layer refers to the actual backend implementations. Currently, Qu4DS implements two backends; one that enables the XtreemOS grid [CFJ+08] and another more generic on top of SSH (Secure Shell)[4]. The latter backend enables the utilization of the job broker by any operating system with an SSH server and a Bash command-line interpreter.

The job broker provides a job scheduling algorithm that ensures that resources are shared in an exclusive fashion. It means that resources may be used by any contract but one at once. As a consequence, jobs that run request tasks and jobs run service

[2]The use of the Grid5000 libcloud driver [Del11] perfectly meets the infrastructure interface requirements, however it was not available during the development of Qu4DS.

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

[4]Unfortunately, there was no SAGA SSH/FSH adaptor when Qu4DS was being implemented. Moreover, the SAGA SSH/FSH adaptor does not provide job dynamic metrics as SAGA job state nor job elapsed time. These facts are the reasons of not using the SAGA SSH/FHS adaptor and implementing the job broker.

instances have dedicated resources. This way of sharing resources is necessary to ensure performance constraints since application profiling is performed on dedicated resources. Indeed, it is not feasible to profile applications on non-dedicated resources because the variation of resource load compromises profiling results. Another consequence of the exclusive sharing scheduling algorithm is that applying under-provisioning implies sharing resources among job request tasks, and not among service instances.

Negotiation and Provisioning Interface The negotiation and provision interface allows performing the communication between Qu4DS and customers. The goal of this interface is two-fold: to enable the negotiation of contracts and the actual service provisioning by means of request calls. Qu4DS implements the service negotiation and provision interface as a SOAP Web Service whose WSDL file exposes the operations described by Listing 4.3. Thereby, the WSDL file is used by customers to know the service provider functionality as well as negotiation operations.

4.4.3 Sequence Diagram

In order to illustrate the runtime flow of Qu4DS behavior, the Figure 4.3 depicts a sequence diagram of a contract enactment followed by a request treatment. The customer is in charge of initiating the negotiation process whose first step is to get the list of contracts templates from the service provider. Then the customer chooses a contract template, sets its duration and propose the customized contract template to the service provider. In turn, Qu4DS asks the QoS translator the resource requirements of the proposed contract template. Based on these requirements, Qu4DS applies the under-provisioning factor upf, creates a new contract and dispatches an event to the resource management control loop which books the necessary resources according to the resource requirements. If resources cannot be booked, the resource management control loop checks if it is more profitable to rescind an on-going contract to accept the just-proposed contract. As resources are successfully acquired, Qu4DS configures a service instance with the right resource requirements and the contract identifier. Through the job broker, Qu4DS creates a job and run it. Following that, the job broker gets an idle resource from the infrastructure and run the job on this resource. This action implies to execute a UNIX process whose process ID is used for monitoring purpose such as the job state. Ultimately, when the job is running, Qu4DS tells the customer that the contract was accepted and the service instance is operational which allows the customer to send requests.

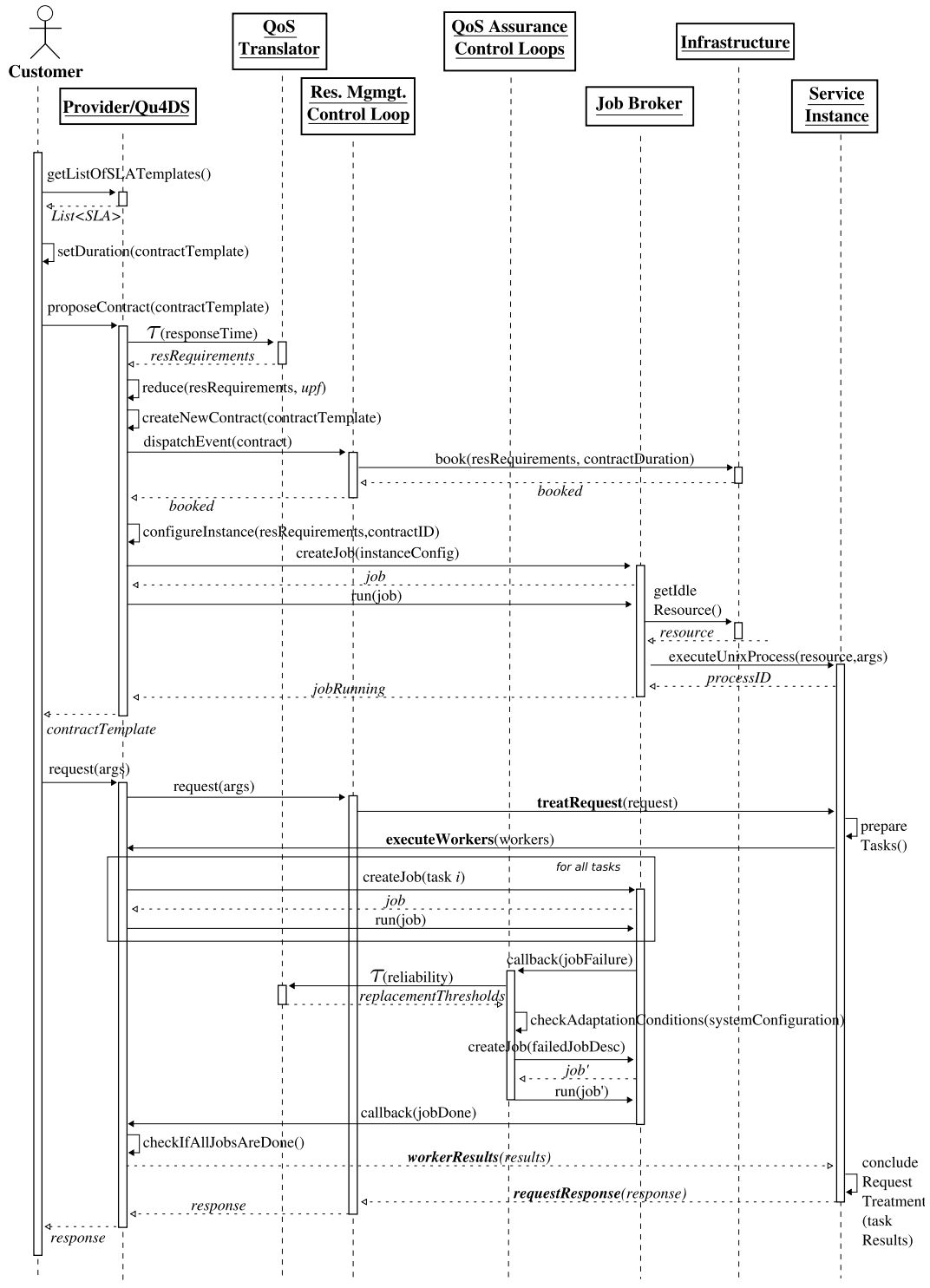


Figure 4.3: Qu4DS general sequence diagram. A customer establishes a contract, Qu4DS instantiates the service and then treats a request sent by the customer.

When a customer sends a request, it is forwarded to the resource management control loop which decides whether it can be treated. If so, Qu4DS forwards the request to the right service instance deployed on the infrastructure. Then 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 broker and monitored by the *job failure* and *job delay* control loops. If these control loops are informed about failures or delays, they will replaced failed or delayed jobs up to their respective replacement thresholds; which are known by translating the reliability QoS metrics. If the job executions are successful, Qu4DS answers to the service instance the result of the tasks which is used to finish the request treatment. Then, the service instance tells Qu4DS that the request is treated and Qu4DS forwards the result to the right customer. If any of the QoS assurance control loops informs Qu4DS that the request could not be treated, Qu4DS aborts the request, tells the customer about such SLA violation and computes the penalties.

4.5 Usage

Qu4DS provides a PaaS solution for increasing the service provider profit by providing automatic support for service execution management and development. In order to understand Qu4DS usage, three different roles are introduced. Firstly, the *customer* interacts with Qu4DS by negotiating contracts and sending requests as an usual service provider. Secondly, Qu4DS frees the *service developer* from development details involving as negotiation and worker execution. Finally, from the *service administrator* point of view, Qu4DS automatically manages the service provider execution by acquiring resources, instantiating and destroying the service instance. These roles are described next.

Customer The *customer* is the service consumer. It establishes contracts with the service provider and send requests to be treated by the service provider.

Service Developer The service *developer* is in charge of developing the service provider. In addition to service provider business logics, development tasks include preparing workers, executing them on distributed resources and processing worker results.

Service Administrator The service *administrator* manages the service provider execution. Management actions include dealing with resource acquisition, instance configuration, instantiation and destruction.

Qu4DS usage is illustrated by Figure 4.4. It depicts the service provider as the common entity among the customer, the service developer and the service administrator. During the contract negotiation (cf. arrow 1), Qu4DS books resources from the infrastructure in order to deploy the service instance. Since the service is instantiated (cf. arrow 2), the contract is established and customer requests (cf. arrow 3) are then forwarded by Qu4DS to the service instance by calling the remote method

`treatRequest(request)` (cf. arrow 4). The service instance prepares the workers and send them through the remote method `executeWorkers(workers)` (cf. arrow 5) to be executed by Qu4DS. Then Qu4DS manages worker execution according to the contract template reliability constraints. After executing the workers, Qu4DS sends their results to the service instance by calling the remote method `workerResults(results)` (cf. arrow 6). Following that, the service instance processes the previous results and answers Qu4DS the request response by calling the method `requestResponse(response)` remotely (cf. arrow 7). Then the request is finally forwarded to the customer by Qu4DS (cf. arrow 8). Moreover, when the contract duration finishes, Qu4DS destroys the service instance (cf. arrow 9) and releases the resources booked to assist this contract. Thereby, Qu4DS ensures the proper execution of workers by handling non-successful worker executions transparently as well as Qu4DS deals with resource acquisition and further runtime environment aspects also in a transparent fashion.

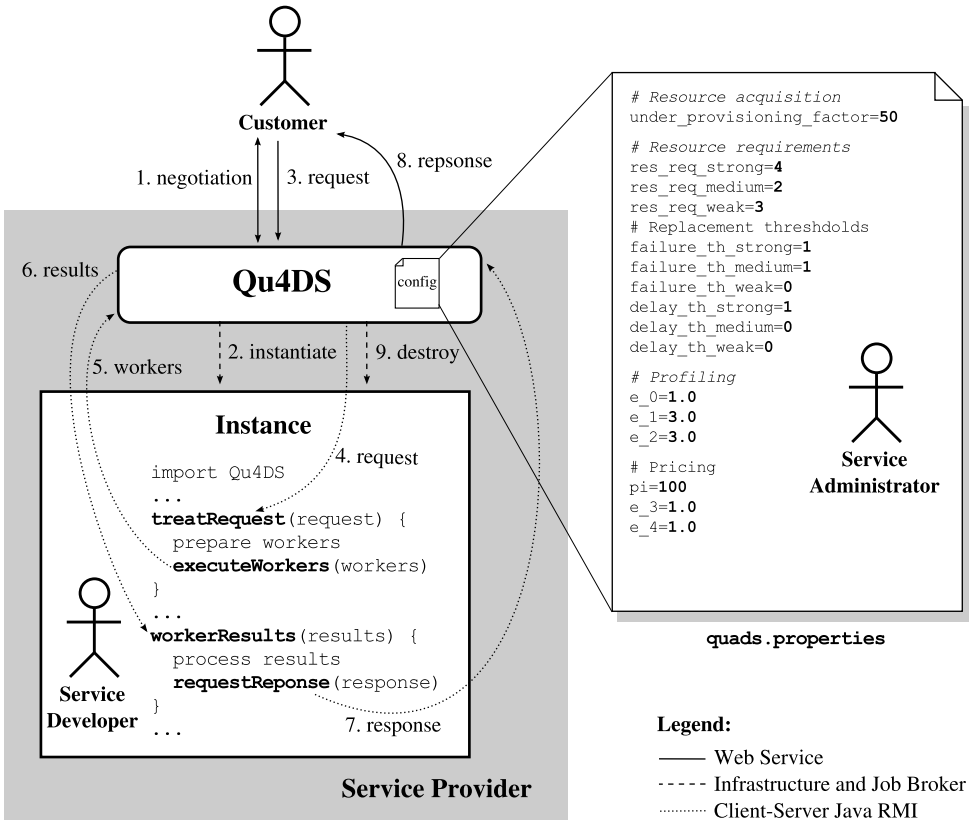


Figure 4.4: Qu4DS aids service developers and administrators automatically and transparently. On the one hand, Qu4DS usage assists service provider developer by managing the execution of workers. On the other hand, Qu4DS assists service administrators by instantiating the service provider and dealing with resource acquisition.

Regarding the service administration, configuring Qu4DS requires setting its configuration parameters in the `quas.properties` file as illustrated in Figure 4.4. Firstly, choosing the under-provisioning factor means to decide how much percent of the necessary resources will be acquired (cf. `under_provisioning_factor`). Secondly, the resource requirements and replacement thresholds should be fulfilled according to their constraints. Then, profiling parameters refers to setting the constants that will be used when profiling the service provider (cf. `e_0, e_1, e_2`). Finally, pricing parameters are the targeted margin profit (cf. `pi`) and the constants `e_3, e_4` are used to calculate request abortion and contract rescission fines, ψ and Ψ respectively.

With respect to the service development, the service developer should import the a Java library in order to take advantage of Qu4DS. This library uses distributed server-client communications and contains an abstract class that should be extended. This class requires implementing the `treatRequest(request)` and `workersResults(results)` methods which allow the service instance to be called by Qu4DS as a server. In turn, the service instance plays the role of a client by calling the `executeWorkers(workers)` and `requestResponse(response)` methods. These latter methods are implemented by Qu4DS library as a server. All these aforementioned methods are remotely invoked by using Java Remote Method Invocation (RMI) whose configuration is automatically done by the library when the service is instantiated.

Ultimately, the bold texts in the sequence diagram depicted by Figure 4.3 also represent the interaction between the service provider and Qu4DS depicted by Figure 4.4. Moreover, in spite of this tight-coupled way of delivering a PaaS service as a Java library, the `executeWorkers(workers)` method can perfectly be also implemented as an external Web Service for instance. This allows taking advantage of a dynamic binding between the service provider and Qu4DS thus exploiting Qu4DS as service provider completely separated from the service provider.

Part IV

Validation

Chapter 5

Environment

This chapter explains the experimentation environment. Section 5.1 introduces the flac2ogg service as a case study that involves parallel audio compression. Following that, Section 5.2 exposes the configuration parameters which concerns the Web Service customer generator, Qu4DS and the infrastructure. Lastly, Section 5.3 explains how the flac2ogg service provider was profiled and details about the QoS translation.

5.1 Case Study: The Flac2Ogg Service Provider

Audio Compression Nowadays, audio is widely utilized in different digital medias such as videos, streaming, radios, social networks and so forth. Owing to the huge quantity of data stored by media servers, audio compression is employed in order to save storage space. Moreover, audio contents is often compressed by applying algorithms that imply minor audio content losses which significantly reduces the media to be stored. For instance, the Free Lossless Audio Codec (FLAC) [The11a] is a compressed audio format which maintains the original audio quality. Although the great advantage of having a lossless compressed audio, FLAC is not suitable for storing a huge quantity of audio in a audio streaming server for instance. An alternative is to this issue is to compress even more FLAC audios to the lossy compressed OGG (Vorbis OGG) [The11b, The11c] [1] audio format thus reducing the audio file size.

The computational requirements for compressing audio files depends on the input file size. It can be done by a desktop computer, however its computational capability is not enough when compressing great quantity of audio. For instance, web sites which rely on audio streaming as Jamendo, Deezer, LastFM compress a huge quantity of audio for streaming lossy audio over the Internet. In order to not deal with the problem of compressing great audio data, these web sites can outsource this task by using external services such as Zencoder, Panda, Sorenson or Encoding.com for instance.

[1] OGG actually is an audio container which supports the Vorbis audio format. Since the `.ogg` file extension is commonly used to refer to a Vorbis audio supported by the OGG container, this thesis will refer to it as *OGG*. Moreover, OGG container also support FLAC audio formats, however this thesis will refer to the FLAC audio format simply as FLAC since it has its own container which is used widely.

Overview The *flac2ogg* service provider aims at providing an audio encoder [2] solution which compresses FLAC files to OGG. The flac2ogg provider relies on a distributed Master/Worker design as depicted by Figure 5.1. The flac2ogg service provider compresses FLAC files in parallel in order to improve its performance. The flac2ogg service provider splits the FLAC file given a number of workers to which it is configured and prepares these workers by setting up their input and output files based on the split FLAC parts. Then, the flac2ogg service provider encodes in parallel each split FLAC file on top of distributed resources. When the encoding process is finished, the flac2ogg provider merges the encoded OGG files to a single OGG file that represents the encoded FLAC file input. Moreover, the FLAC and OGG formats were chosen to be used by this case study not only because both are patent-free formats with open-source implementations, but also owing to their technical advantages against further formats such as WAV and MP3 for instance.

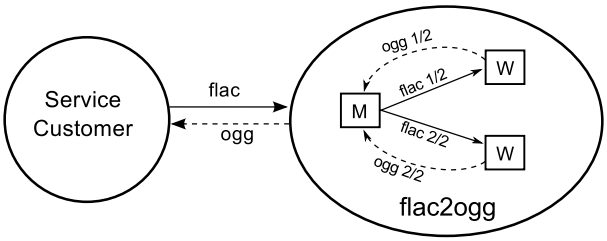


Figure 5.1: The flac2ogg service provider compress FLAC audio files to OGG based on the Master/-Worker pattern.

Implementation The whole process of encoding an audio file in parallel by the flac2ogg service provider comprises splitting, submitting, encoding, and merging phases as explained next.

Splitting Consists of splitting the input FLAC in n equal parts where n is the number of workers by using the `shnsplit` Shntool [Jor09] command. The performance of the splitting phase depends on the FLAC file size where the greater the FLAC file is, the more time it will take to be split. The `shnsplit` Shntool command does not actually split the original FLAC file, it creates n FLAC files where each file represents a copy of a part of the original FLAC file. Moreover, the `shninfo` Shntool command is used to get audio meta-information used by `shnsplit`.

Submitting Means to wrap the workers as jobs and submit them to be executed on the infrastructure through the job management interface. This phase does depend on the number of workers since the more fragmented the FLAC file is, the more time it take to submit all split FLAC parts to be encoded. However, it does not

[2] The terminology *encode* is widely used to refer to the action of converting audio file formats where compression is often employed. This thesis will use the verb *encode* to mean a compressing conversion of FLAC audio files to OGG audio files.

depend neither on the size of the FLAC size nor on the size of each FLAC split part.

Encoding Employs the actual compression of each FLAC split part to its respective OGG output. The more number of workers, the faster it is. In addition, the actual compression is performed by the `oggenc` Vorbis Tools [Xip11] command.

Merging Means to merge all the generated OGG files to a single OGG file that represent the compressed original FLAC input. The number of workers here is insignificant, however the FLAC file size affects the merging phase performance. Similar to the splitting phase, the greater the FLAC input is, the more it will take to accomplish this phase. However, because the size of the OGG output file is is smaller than the FLAC input, the merging phase is faster than the splitting phase since there is less data to be stored.

The Figure 5.2 depicts how the flac2ogg provider is implemented by using Qu4DS. The Qu4DS framework aids the service developer by handling the *submitting* and *encoding* phases. When a request is sent the developer splits the FLAC file and prepares the workers by setting each FLAC part as input and the `/usr/bin/oggenc`[3] as the executable to process the input. Then the developer asks Qu4DS to execute the workers which includes wrapping them as jobs and managing job submissions through the job broker. The encoding phase is performed in parallel on distributed resources and it is up to Qu4DS to ensure that jobs will be successfully ran. When all the FLAC parts are encoded, Qu4DS warns the service developer who then merges the OGG parts into a single OGG output and respond the request. Importantly, Qu4DS only requires worker inputs and executable which enhances Qu4DS generic characteristic thus be able to support different service providers.

5.2 Configuration Parameters

This section discusses the configuration parameters regarding to Qu4DS evaluation. First, it exposes the customer generator and its configuration aspects in Section 5.2.1. Then, Qu4DS configuration parameters are introduced by Section 5.2.2. The Section 5.2.3 explains the parameters which allow configuring the infrastructure and the job broker.

5.2.1 Customer Generator

A customer generator was designed and implemented in order to perform the experiments. The customer generator implements Web Service (WS) customers which have their own request schedule. The request schedule defines when customers will trigger the contract negotiation and send requests if the contract is accepted. The features

[3]For systems which use Debian packages, the Vorbis Tools [Xip11] package places the the OGG encoder binary at `/usr/bin/oggenc`.

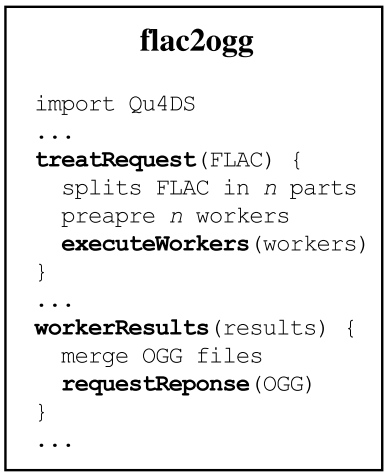


Figure 5.2: Qu4DS is in charge of managing worker execution in distributed resources thus letting the service developer concentrate in specific-domain tasks.

of the customer generator include customer scheduling; loading previously created customer schedules; launching simulation; and plotting schedules to a graphic. Moreover, configuring, the customer generator requires setting the following parameters.

Experiment duration It sets the total time of the experimentation. The more time the experimentation takes, the more suitable it is for creating heterogeneous scenarios, i.e., scenarios with greater scheduling variations.

Number of customers It sets the total amount of customers for each contract template label. The more customers, the more contract negotiations occur. Furthermore, contract template labels which rely on high resource requirements require more resources, while labels that prioritize fault tolerance use less resources. The experimentation includes all labels (fast, safe, classic, standard) and set the total number of customers to ten or twelve depending on the scenario.

Contract duration The lower contract durations are, the less requests can be scheduled. The experimentation relies on fixed and variable contract durations, varying from the time to treat one request up to the whole experimentation duration.

Request load (λ') Defines the expected request load within the contract duration. The parameter λ' represents a percentage of the maximum number of requests within the contract duration (ϕ_max). As the targeted request load λ' is the expected load and not the actual load, λ' is used to generate a random number which will be the actual number of requests sent during the contract duration. This random number is based on the Poisson distribution since it expresses the probability of the number of events to occur during a time interval where the average rate is known. Therefore, Poisson's $\lambda = \lambda' \cdot \phi\_max$. Moreover, if the

random number is zero, the customer generator sets it to one since it assumes that if a customer establishes a contract, it will eventually use the service. On the other hand, if the random number generated is greater than ϕ_max, the customer generator sets it to ϕ_max since the number of requests cannot exceed ϕ_max. Last, the greater λ' is, the higher is the probability of having greater number of requests during the contract duration. Thereby, the parameter λ' directly influences resource usage.

Demand Profiles The customer generator is used to create diversified customer demand profiles. The contract template labels are combined in order to qualify the targeted customer demands based on differentiated QoS. Table 5.1 exposes the customer demand profiles used by the experimentation scenarios. The former is called *high-FT* and represents a customer demand which requires a high-level of fault tolerance, being then composed by customers whose contract template label is safe. The next customer demand profile is called *hybrid* and expresses a heterogeneous customer demand whose customers hold different contract template labels by including fast, safe, classic and standard labels. Finally, the *high-RR* profile addresses representing a customer demand that has strong resource requirements, hence it holds customers whose contract template label is fast.

Demand Profile	**high-FT**	**hybrid**	**high-RR**
Fast	0	3	10
Safe	10	3	0
Classic	0	3	0
Standard	0	3	0
Number of resources	**30**	**45**	**50**

Table 5.1: The formation of customer demand profiles is based on the combination of different contract template labels. *Number of resources* means the total amount of resources required by all contracts of a demand profile.

Five request schedules were created for each demand profile which are divided in two groups as depicted by Table 5.2. The first group is based on four values of request load $\lambda' : 0.25, 0.50, 0.75, 1.00$ whose contract durations were fixed to the experimentation duration, i.e., nine-hundred seconds. The second group refers to demand profiles with variable contract durations and request load set to $\lambda' = 1.00$. These parameters along with the Table 5.1 were used by the customer generator to create the request schedules for each demand profile. Furthermore, request schedules with variable contract durations assume that cheaper contracts are firstly established, then more expensive contracts are proposed to the provider. The graphics representing all request schedules can be found in Appendix A.

5.2.2 Qu4DS

Qu4DS configuration parameters include the under-provisioning factor and parameters related to QoS translation constraints, profiling and pricing as discussed in Section 4.5.

Request schedule	**Request load**	**Total**
fixed contract duration (exp. duration)	$\lambda' = 1.00, 0.75, 0.50, 0.25$	4
variable contract durations	$\lambda' = 1.00$	1

Table 5.2: Each customer demand profile holds five request schedules which are divided in two groups based on the contract duration.

They are summarized next.

Resource acquisition The `under_provisioning_factor` parameter configures Qu4DS to apply the given under-provisioning factor when booking resources. The experimentation relies on the following values; $upf = 100, 70, 50$.

QoS translation constraints The parameters used to fill the QoS translation constraints refer to resource requirements and replacement thresholds constraint. The former depends on service profiling while the latter is statically defined.

Profiling It includes the fixed e_0, e_1, e_2 parameters which refer to setting the response times and job execution times.

Pricing The values of π is statically defined to $\pi = 100$ which means that the provider earns one-hundred percent from each contract. The e_3 and e_4 constants, which are used to set fine costs, assume the following values: $0.5, 1.0, 2.0$.

Furthermore, the Section 5.3 explains how fixed values of the QoS table, profiling and pricing parameters are set.

Under-Provisioning Profiles In order to experiment Qu4DS with different resource acquisition policies, three under-provisioning profiles are defined as depicted by Table 5.3. On the one hand, the *no-UP* under-provisioning profile does not reduce the resource requirements since $upf = 100\%$. On the other hand, the *UP-70* and *UP-50* under-provisioning profiles imply respectively acquiring seventy and fifty percent of the total amount of the required resources.

Under-Provisioning Profile	**no-UP**	**UP-70**	**UP-50**
Under-provisioning factor (upf)	100	70	50

Table 5.3: The under-provisioning profiles are created based on various values of the under-provisioning factor $upf = 100, 70, 50$.

5.2.3 Infrastructure

This configuration parameters related to the infrastructure and job broker described as follows.

Infrastructure capacity Sets the total amount of resources that the infrastructure can provide. The less resources the infrastructure holds, the more contracts may

be rescinded if the customer demand profile is great enough to saturate the infrastructure provision capacity. The experimentation sets the infrastructure capacity based on a percentage of the total amount of resources required by a customer demand profile whose values are $100\%, 75\%, 50\%$.

Percentage of job misbehaviors Sets the percentage of jobs which will present misbehaviors at runtime. The experimentation assumes that $0, 5, 10, 20, 40$ percent of the total jobs[4] misbehave. For instance, when set to 20%, implies 10% of the jobs will be failed and other 10% of them will be delayed. These jobs are chosen randomly based on the normal distribution. In addition, the greater the percentage of job misbehaviors, the more SLA violations may occur thus the more adaptation actions may be triggered.

5.3 Profiling

This section tackles profiling issues by explaining how the flac2ogg service provider was profiled. It comprises the choice of profiling, pricing and performance constants used to fill the QoS table.

5.3.1 Constants

Profiling: e_0, e_1, e_2 e_0 and e_1 are used to calculate the job execution threshold (cf. Equation 4.5, Section 4.2.3) which is used to decide whether a job is delayed by the QoS assurance mechanism presented in Section 3.4.2. These constants were set to $e\_0 = 1.0$ and $e\_1 = 3.0$. With regard to the constant e_2, it is used to calculate the response time (cf. Equation 4.6, Section 4.2.3). This constant was set to $e\_2 = 3.0$.

Pricing: π The constant π refers to the profit that the provider wants to earn (cf. Equation 4.2 in Section 4.2.2). The value for the constants `pi` (cf. Section 4.5) was set to 100 which means that the service provider earns 100% from each contract.

5.3.2 Response Time Constraints

This section explains how the performance constraint parameters *res_req_strong*, *res_req_medium*, *res_req_weak* were chosen by profiling the service provider. Service profiling consisted of executing the flac2ogg provider with based on different resource requirements.

The Size of the FLAC File The request response time depends on the size of the FLAC file. In order to rely on short experimentation durations, the size of the FLAC file was fixed. It is important to remark that this decision does not compromise the experimentation results since it can be easily overcome by dynamically checking the right request response time for the given variable FLAC file size. The size of the

[4]Service instance jobs excluded, only jobs belonged to requests can be failed or delayed.

FLAC file was fixed to 194MB as the benefits of the parallelization present significant performance differences for FLAC file greater than 194MB.

The Number of Workers In order to understand the relationship between the request response time and the number of workers, the number of workers vary from two to nine. Figure 5.3 depicts the profiling of the flac2ogg provider whose goal is to identify which numbers of workers are not worth to be used. With respect to the job execution time, the *job-exec-time* line shows that the more the FLAC file is divided, the faster each part is encoded since the smaller the split FLAC part is, the faster it is encoded. In contrast, the *actual-resp-time* line does not follow this behavior because the submitting phase of the encoding process increases the overall encoding time. It is explained by the fact that the more fragmented the FLAC file is, the more time the submitting phase takes.

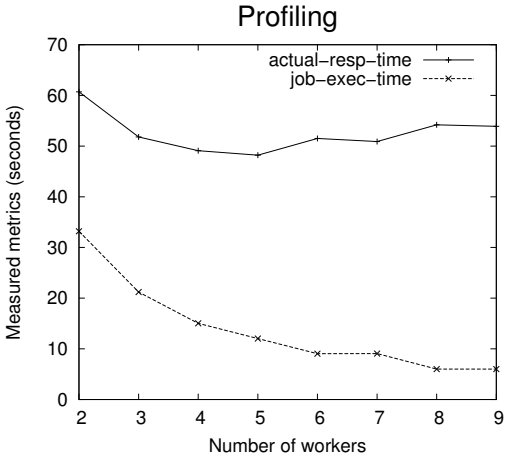


Figure 5.3: Profiling of the flac2ogg service provider based on different number of workers. The input FLAC file has 194MB.

Finally, the *actual-resp-time* line shows that when the number of workers is greater than five, the request response time starts to increase. Thus more than five workers are not worth to be used. In order to not require a great number of resources, the chosen number of workers to work with are two, three and four.

5.3.3 QoS Table

While the profiled data are used to enable the translation of response time, data concerning the translation of reliability is set statically. The Table 5.4 depicts the QoS table introduced in Section 3.3.1 (cf. Table 3.1) with the translation of each QoS metrics constraints. The translation of the response time constraints represent the number of workers and were set based on the previously chosen number of workers two, three and

four. The translation of the availability constraints were set statically and represent the replacement thresholds of the QoS assurance mechanisms.

QoS Aspects	**Performance**	**Fault-Tolerance**
QoS Metrics / Contract Template Labels	**Response Time**	**Reliability**
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 5.4: The QoS table used in Qu4DS evaluation whose QoS metrics are presented along with their respective translated system configurations. The translation of the response time means the number of workers while the translation of the reliability represents the replacement thresholds.

Chapter 6

Evaluation

This chapter discusses the evaluation of Qu4DS through empirical experimentation. First, the scenarios are introduced in Section 6.1 which explains how basis parameters, under-provisioning profiles and customer demand profiles are combined in order to create the scenarios. Section 6.2 then discusses the results. Note that this chapter refers to the Qu4DS current version 0.4. Previous results from earlier Qu4DS versions can be found in [FPP11] and [FPP10a].

6.1 Scenarios

The general goal of the experimentation scenarios is to analyze the impact on the provider general profit when *reducing infrastructure costs*, *rescinding contracts* and *preventing SLA violations* (cf. Section 3.2). Qu4DS configuration should be driven by the service provider business goals, customer demand and infrastructure conditions; thus different configurations of Qu4DS, customer demand and execution environment are employed.

Under-provisioning is used as a capability of reducing infrastructure costs which indeed is a mean of increasing the provider profit by compromising QoS assurance. Hence under-provisioning involves a risk which may be worthwhile to consider in some situations. Therefore, another goal of the evaluation is to identify these worthy situations and measure the negative impact of under-provisioning when it should not be applied.

With respect to preventing SLA violations and rescinding contracts, two patches were added to Qu4DS which allows disabling the fault tolerance QoS assurance mechanism and contract rescissions. The former patch ignores the replacement threshold values by setting them to zero. The latter patch bypasses the accept operation described by Algorithm 4 in the flowchart illustrated by Figure 3.6 (cf. Section 3.5.1).

In order to address the experimentation objectives, Scenarios A, B and C were created as depicted by Table 6.1. The objectives of each scenario are described next.

Scenario A Focus on reducing infrastructure costs. The goal is to understand how under-provisioning profiles applied to the customer schedule profiles on different

request loads impact on the profit.

Scenario B Focus on rescinding contracts. The goal is to understand how various infrastructure capacities impact on the profit with and without contract rescission.

Scenario C Focus on preventing SLA violations. The goal is to understand how different percentages of misbehaved jobs impact on the profit and analyze with and without the fault tolerance QoS assurance.

Configuration	**Scenario A**	**Scenario B**	**Scenario C**
Customer profile	`fixed contract duration`	`variable contract duration`	`fixed contract duration`
Request load	`0.25, 0.50, 0.75, 1.00`	`1.00`	`1.00`
Under-prov. profile	`no-UP, UP-70, UP-50`	`no-UP`	`no-UP`
Infra. capacity	`100%`	`100%, 75%, 50%`	`100%`
Misbehaved jobs	`0%`	`0%`	`0%, 5%, 10%, 20%, 40%`
FT QoS assurance	`on`	`on`	`on,off`
Contract rescission	`on`	`on,off`	`on`

Table 6.1: The configurations used to create the Scenarios A. B and C.

Furthermore, fine costs have a direct impact on the provider profit, hence it is important to let the experimentation scenarios rely on variable fine costs. Request abortion fines (ψ) are defined based on the price requests cost to be treated while rescission fines (Ψ) are defined based on the contract price in such a way that both fines depend on the e_3 and e_4 variables respectively (cf. Equation 4.3 in Section 4.2.2). Thus the relationship between fine costs and their commitments can be expressed based on different values of the e_3 and e_4 variables as presented next.

e_3 **or** e_4 **equals to 0.5** means that fines cost *half the price* of their commitments.

e_3 **or** e_4 **equals to 1.0** means that the fine costs *the same price* of their commitments.

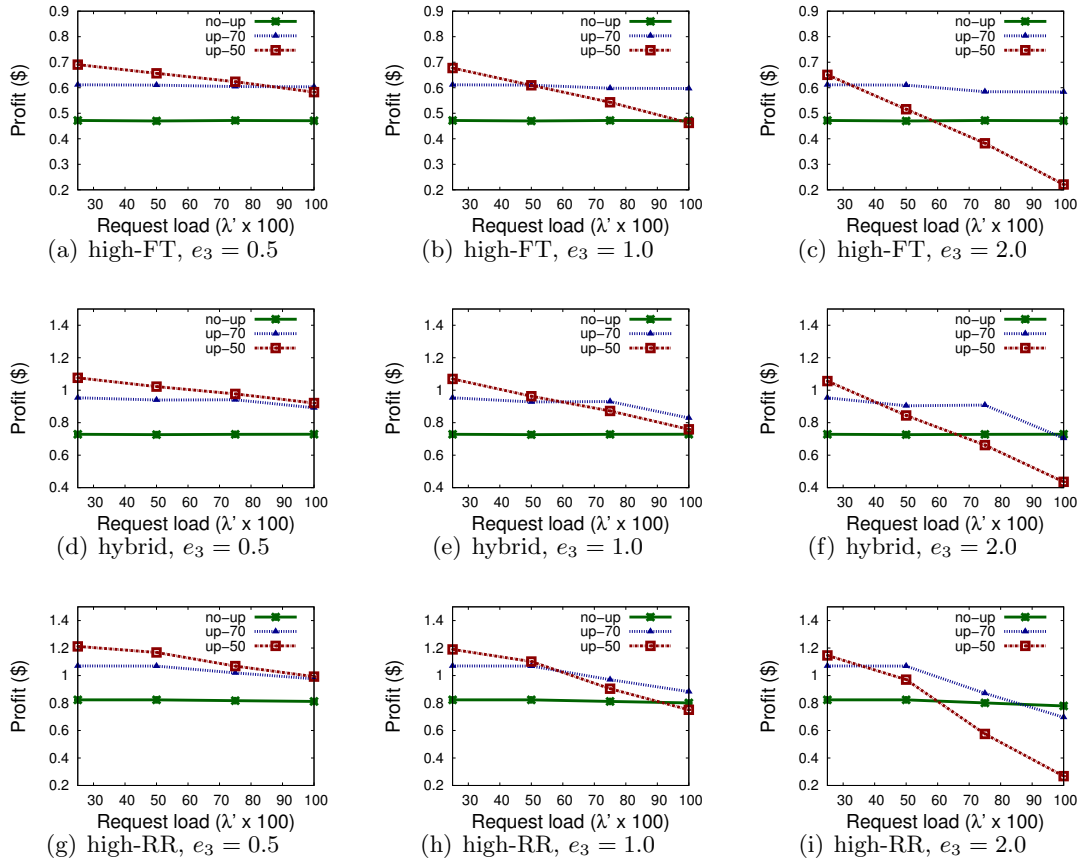
e_3 **or** e_4 **equals to 2.0** means that the fine costs *twice the price* of their commitments.

6.2 Results

The experiments were performed in the Rennes site of the Grid5000 [CCD+05] testbed grid. Specifically, the paradent cluster was used which holds nodes with eight 2.5GHz processors, 32GB RAM memory connected through a Gigabit network. Moreover, all graphics express the net general profit (cf. Equation 3.7) as a function of a given variable in the X-axis. Each dot plotted in the graphics represents a different experiment whose total duration is fifteen minutes.

6.2.1 Scenario A

The evaluation results of the Scenario A are depicted by Figure 6.1. The Scenario A assumes that the infrastructure capacity is enough to satisfy the customer demand profiles. Moreover, it also assumes that there is no job misbehavior. On the other hand, there are three variable parameters. The former variable parameter is the request load λ' which is placed in the X-axis. The second and third variable parameters are drawn as curves: the under-provisioning profiles (`no-UP, UP-70,UP-50`) and the cost of the request abortion fine ψ which depends on different values of the e_3 constant.



(a) high-FT, $e_3 = 0.5$ (b) high-FT, $e_3 = 1.0$ (c) high-FT, $e_3 = 2.0$

(d) hybrid, $e_3 = 0.5$ (e) hybrid, $e_3 = 1.0$ (f) hybrid, $e_3 = 2.0$

(g) high-RR, $e_3 = 0.5$ (h) high-RR, $e_3 = 1.0$ (i) high-RR, $e_3 = 2.0$

Figure 6.1: **Scenario A**: high-FT, hybrid and high-RR customer demand profiles with various values of request abortion fine ψ.

The first general observation regarding the results of the Scenario A is that all the `no-UP` curves (full lines) approximatively are horizontal lines which means that the profit does not vary as the request load increases. This continuous profit happens because there is neither under-provisioning, nor SLA violations which would eventually increase or decrease the profit. Hence all the `no-UP` lines serve as a basis to know which profit is reached when no under-provisioning is applied. Moreover, different customer demand profiles have different continuous profit values as the price of the contract

depends on resource usage which varies according to the customer demand profile.

On the other hand, the `UP-70` and `UP-50` curves make the profit vary given a request load. Theoretically, the lower the under-provisioning factor is, the greater the profit is until the request load begins to be important enough to imply lack of resources owing to parallel request treatments. This behavior is an interesting aspect to observe in Scenario A in order to know in which situation a given under-provisioning factor is worth to be applied.

In general, the use of under-provisioning showed to be effective in increasing the service provider for these experiments. Both up-50 and up-70 under-provisioning profiles allowed increasing the profit if compared to the no-UP under-provisioning profile. However, the up-50 and up-70 under-provisioning profiles decrease as the fine cost increases. Moreover, up-50 decreases quicker than up-70 because up-50 represents a more aggressive under-provisioning profile which is more appropriated for low request loads. Thereby, up-50 reached greater profit when the request load is not high while up-70 enables to increase the profit in the average case. Therefore, it is more prudent to apply the up-70 under-provisioning profile for unknown request loads. In contrast, if a low request loads is expected, the up-50 under-provisioning profile allows increasing even more the profit.

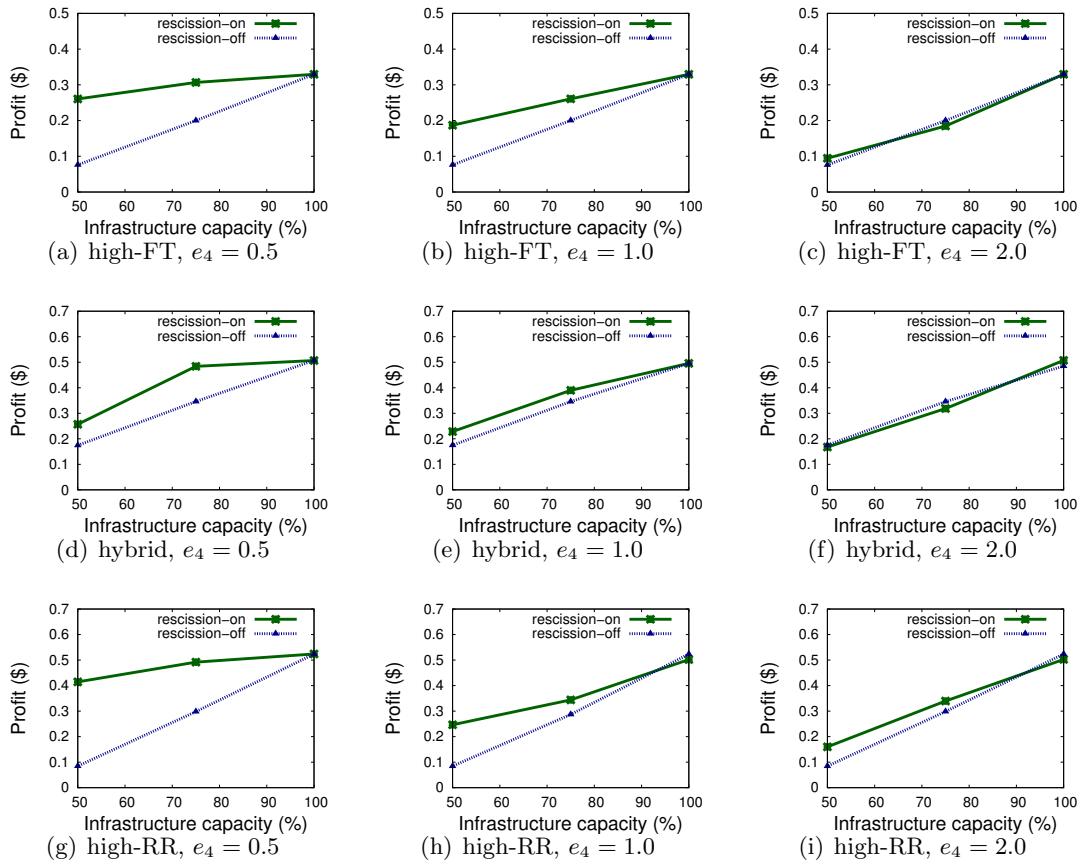
The curve-crossing points indicate the request load which an under-provisioning profile is less effective than another under-provisioning profile. In Scenario A, by doubling the fine costs, i.e. $e_2 = 0.5, 1.0, 2.0$, anticipates the curve-crossing points in a inverse ratio approximatively, i.e., by halving the request loads of the curve-crossing points. Therefore, in order to maximize the provider profit in a situation similar to Scenario A, the up-50 under-provisioning profile should be used for values of e_3 around 0.5 approximately. Moreover, the up-70 under-provisioning profile showed to be an interesting configuration for the average case, i.e, it satisfactorily increased the profit for various request loads.

6.2.2 Scenario B

Figure 6.2 depicts the evaluation results of the Scenario B. The Scenario B assumes that there is no under-provisioning and it does not take into account job misbehaviors. Because rescinding contracts compares contract prices and fines, the Scenario B relies on customer demand profiles with variable contract durations by implying different contract prices even for contracts with the same label. With regard to the variable parameters, the first variable is the infrastructure capacity whose values are exposed in the X-axis. The second variable is the rescission fine (Ψ) cost which varies according to different values of the e_4 constant. The last variable parameter refers to the contract rescission patch which was enabled and disabled for each experiment of Scenario B.

The advantage of this rescinding contracts is to allow the service provider to try to increase its profit even when the infrastructure cannot deliver no more resources. In this sense, the results depicted by Figure 6.2 show the profit being increased for almost all the results, cf. the full lines above the dashed lines.

Moreover, as one can note, the profit grows as the infrastructure capacity increases.



(a) high-FT, $e_4 = 0.5$ (b) high-FT, $e_4 = 1.0$ (c) high-FT, $e_4 = 2.0$

(d) hybrid, $e_4 = 0.5$ (e) hybrid, $e_4 = 1.0$ (f) hybrid, $e_4 = 2.0$

(g) high-RR, $e_4 = 0.5$ (h) high-RR, $e_4 = 1.0$ (i) high-RR, $e_4 = 2.0$

Figure 6.2: **Scenario B**: high-FT, hybrid and high-RR customer demand profiles with various values of rescission fine Ψ.

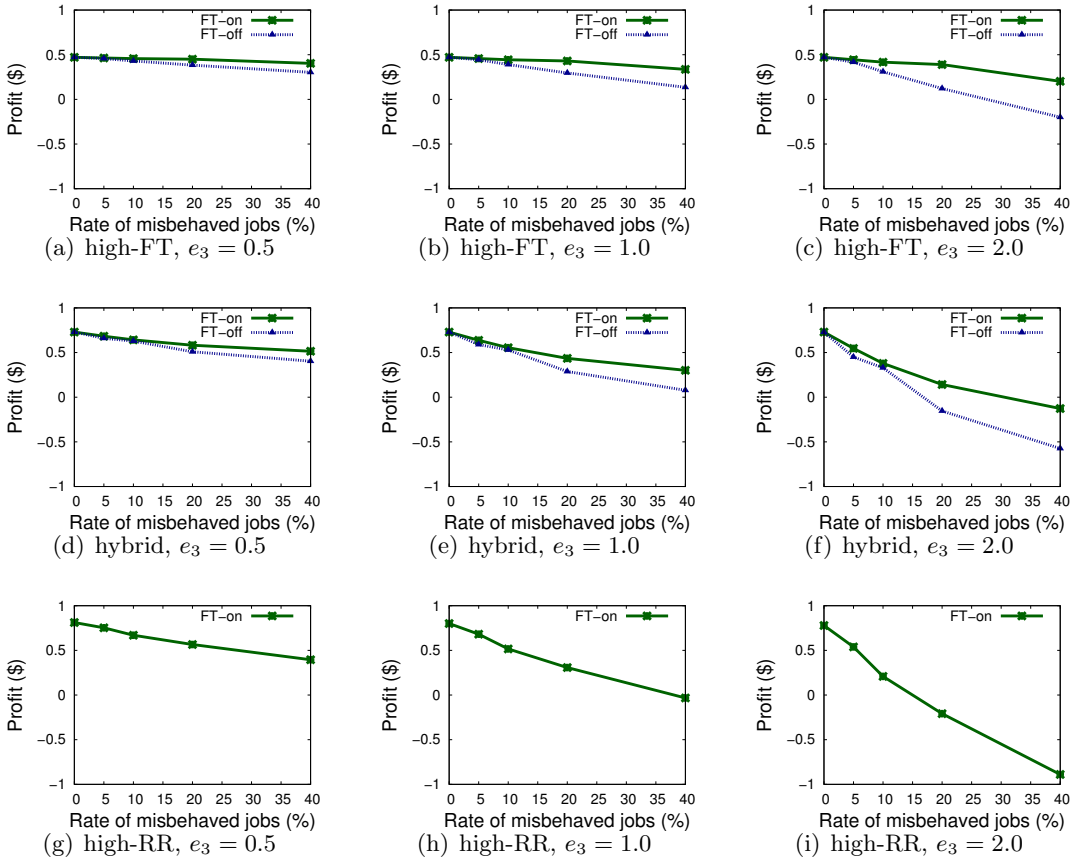
This expected behavior is explained by the fact that the more resources can be acquired, the more contract will be established then increasing the profit.

Lastly, the greater the fine is, the less effective rescinding rescission is. Indeed, the configurations of the Scenario B implied Qu4DS similar behaviors when the rescission fine is the highest. Because high fine costs inhibits rescinding contracts, few contracts are rescinded when $e_4 = 2.0$ for instance. On the other hand, setting lower fine costs allows the rescinding capability to increase the profit, i.e., $e_4 = 0.5, 1.0$.

6.2.3 Scenario C

The Figure 6.3 shows the results from Scenario C. The Scenario C relies on a execution environment where the infrastructure capacity is able to deliver all the required resources. It also assumes that the request load is $\lambda' = 1.0$ based on a fixed contract duration. Moreover, there is no under-provisioning which means that the no-UP under-provisioning profile is applied. On the other hand, since the Scenario C evaluates Qu4DS effectiveness in preventing SLA violations, the job misbehavior rate is variable.

In addition, the FT QoS assurance patch is also variable; being only applied for the hybrid and high-FT customer demand profiles as high-RR does not include contract template labels that require fault tolerance. The last variable parameter is the request abortion fine (ψ) whose costs are defined based on values of the e_4 constant.



(a) high-FT, $e_3 = 0.5$ (b) high-FT, $e_3 = 1.0$ (c) high-FT, $e_3 = 2.0$

(d) hybrid, $e_3 = 0.5$ (e) hybrid, $e_3 = 1.0$ (f) hybrid, $e_3 = 2.0$

(g) high-RR, $e_3 = 0.5$ (h) high-RR, $e_3 = 1.0$ (i) high-RR, $e_3 = 2.0$

Figure 6.3: **Scenario C**: high-FT, hybrid and high-RR customer demand profiles with various values of request abortion fines ψ. The curves with no fault-tolerance are not present in high-RR graphics as this customer demand profile inherently has no customers with fault-tolerant mechanisms enabled.

The first general observation is that the greater the misbehavior rate is, the lower the profit is. This expected behavior happens because misbehaved jobs increases the chances of implying SLA violations. Regarding the customer demand profiles, the more it holds contract template labels which require fault tolerance capabilities, the more moderate the profit drops. For instance, the curves of the high-RR customer demand profile (i.e., no fault tolerance is required) decreases sharper than hybrid and high-FT profiles. Specifically, the high-FT profile was able to keep a moderate decrease of profit curves.

Secondly, it is remarkable the difference between the profits when the FT QoS assurance is disabled. Qu4DS QoS assurance mechanism that addresses fault tolerance

showed to be effective for the configuration of Scenario C. All the full line curves are plotted above the dashed lines which means that disabling the QoS assurance mechanism negatively influences the profit.

Finally, the greater the fine is, the sharper the profit decreases. An interesting aspect to note in the Scenario C is that Qu4DS QoS assurance mechanism showed to be effective even for great job misbehavior rates. For instance, when the job misbehavior rate is 40% (cf. Figure 6.3(c)), Qu4DS prevented the profit to reach negative values.

Part V

Conclusion

Conclusion

This thesis addressed the problem of managing service execution in distributed infrastructures while meeting SLA constraints. SLA management functionalities are proposed which handle service negotiation, instantiation, provisioning and termination in an autonomous fashion. In addition, the thesis employs further management actions in order to increase the profit. These actions include rescinding contracts, reducing infrastructure costs and preventing SLA violations. This thesis also implements the Qu4DS framework which supports the development and management of services in distributed infrastructures. Moreover, experimental results in Grid5000 show Qu4DS effectiveness in maintaining SLAs while increasing the provider profit. The contributions of this thesis are explained in more detail next.

Contract Templates, SLA Translation, Pricing This thesis proposes *a systematic approach for creating contract templates* based on the combination of performance and fault tolerance aspects. Firstly, labels are used to ease the identification of each contract template based on its response time and reliability QoS metrics. Then, high-level QoS metrics constraints are translated to system-level configurations which allow enforcing specific QoS metrics values. Finally, contract templates are integrated with a pricing model which considers aspects concerning both the service provider and its underlying environment. Thereby, the way of how contract templates are created proposed by this thesis: *(i)* enables contract negotiation through contract templates identified by labels; *(ii)* translates SLA quality properties to system configuration; and *(iii)* enables the realization of runtime environment mechanisms driven by SLA objectives.

QoS Assurance Mechanisms This thesis proposes *two mechanisms which ensure the QoS properties described in the SLA*. The first QoS assurance mechanism addresses performance by ensuring the response time QoS metrics. It relies on the translation of the response time QoS metrics to resource requirements able to meet the given response time. Then these resource requirements are used to acquire resources as well as to configure and deploy the service instance. The second QoS assurance mechanism deals with fault tolerance by ensuring the reliability QoS metrics. The fault tolerance QoS assurance mechanism replaces failed and delayed jobs based on the translation of the reliability QoS metrics to replacement thresholds. Therefore, this thesis proposes QoS assurance mechanisms which guarantee that both performance and fault tolerance

QoS are ensured at runtime. As a consequence, SLA violations are prevented which contributes to increasing the provider profit.

Resource Acquisition and Allocation This thesis proposes *algorithms for handling resource acquisition and allocation* which are driven by contract proposals and request arrivals respectively. Contract proposals trigger resource acquisition which is based on the resource requirements translated from response time QoS metrics. In order to increase the provider profit, this thesis applies under-provisioning by reducing the resource requirements. Moreover, if the infrastructure cannot deliver the required resources, current established contracts may be rescinded if this positively influences the general profit. With regard to resource allocation, upon a request arrival, resource availability is checked in order to decide whether the request will be carried on or aborted. If there are not enough resources available, the request to be aborted is chosen by aiming at minimizing the impact on the profit. As a result, this thesis automatically handles resource acquisition and allocation while applying techniques for profit increase in a transparent manner.

The Qu4DS Framework This thesis describes *the design and implementation of an autonomous framework which manages service execution driven by business-level objectives*. The Qu4DS framework aims at the cloud PaaS layer by aiding the development of SaaS services on top of IaaS resource providers. On the one hand, the Qu4DS framework eases the development of Web Services by abstracting over distributed infrastructures. On the other hand, Qu4DS autonomously manages the service execution while increasing the service profit. In other words, Qu4DS deals with service negotiation, instantiation, provision and termination at runtime under business-oriented policies. These policies serve as high-level guidelines for under-provisioning, contract rescission, QoS assurance as well as pricing parameters. Moreover, the Qu4DS design enables extensibility in order to support further policies. In order to evaluate Qu4DS, experiments were carried out in Grid5000 based on different configurations and customer demands. The results confirm the effectiveness of Qu4DS in increasing the service provider profit while meeting SLAs terms.

Future Work: Short Term

Scalability The current implementation of the Qu4DS framework does not aim at scalability. It relies on a centralized design which makes it difficult to support increasing number of requests. When the number of concurrent requests increases, the Qu4DS workload also increases owing to the management of distributed tasks and their event handling. In order to address this issue, request management should be delegated to the deployed service instances in order to let Qu4DS focus on higher-level SLA management tasks. Thereby, customers would directly send requests to service instances which would be in charge of treating requests according to the agreed QoS. In order to implement such a design change, scalable peer-to-peer techniques may be used to address

location and communication of distributed resources. Examples of such techniques include DHTs (Distributed Hash Tables) and unstructured peer-to-peer networks based on epidemic protocols.

Leveraging IaaS Cloud Capabilities Currently, the Qu4DS framework relies on a single resource provider and it only considers horizontal elasticity as means for improving performance. With respect to elasticity, it profiles the service provider with various numbers of resources whose requirements are the same. It would be interesting to also exploit vertical elasticity features from clouds providers by also changing resource requirements (i.e., resource class) and not only the number of resources. Exploiting vertical elasticity allows implementing further QoS assurance mechanisms, and it eases adapting the Qu4DS load according to customer demand.

Regarding different resource providers, future work may consider enabling Qu4DS to support various IaaS clouds by checking prices and negotiating contracts with them. This would allow overcoming resource shortages and the utilization of cheaper resource providers. In order to implement this solution, different IaaS clouds can be abstracted by using the common OCCI IaaS cloud interface [ME11] or by using the Apache Libcloud API [ASF11].

On-The-Fly Profiling The profiling mechanism implemented by the Qu4DS framework profiles the service provider before provisioning. However, profiling data during provisioning is useful for calibrating Qu4DS according to unpredicted environment changes, e.g., network throughput degradation and unusual customer demands. An improvement to this issue is enabling Qu4DS to update the provider profiling while treating customer requests. As a consequence, current values of job execution times and actual request response times would be used to calculate the response time QoS metrics. Therefore, profiling the service provider on-the-fly allows adjusting QoS metrics to the current customer demand load.

Enable Renegotiation The way that Qu4DS negotiates contracts can be performed in a more flexible manner by including renegotiation of established contracts. Supporting renegotiation allows adapting the service provisioning to environment changes without violating SLAs. In case of resource fluctuations, new contract proposals or changes on the provider business model, it may be preferable to try to renegotiate current SLA terms instead of rescinding contracts or aborting requests in order to increase the provider profit. In order to enable contract renegotiation, complete negotiation protocols should be considered as WS-Agreement [ACD+07]. For instance, the WSAG4J [Fra08] WS-Agreement implementation can be used by creating Agreement Template documents along with their Guarantee Terms. In this case, the Business Value List would express how the fines are assessed while the Service Level Objective would express which QoS metrics should be met.

Automatic Under-Provisioning Configuration This thesis allows configuring service execution management based on the under-provisioning factor upf, i.e., a percentage of the resource requirements. The proposed solution relies on a fixed value of upf. A future research direction is to investigate the automatic adjustment of upf at runtime according to changes on infrastructure, service reputation, service business model and customer demand. This may require predicting the customer demand based on historical data which can be done by simple but effective methods such as exponentially weighted moving average (EWMA).

Further Perspectives

Qu4DS as a Service A future work may consider to completely decouple Qu4DS from the service provider. This requires investigating the relationship between Qu4DS and provider SLAs and profits. Regarding the SLA, the interaction between Qu4DS and the SaaS provider (e.g., the flac2ogg service provider) would be defined by another SLA which describes Qu4DS as a self-contained PaaS provider. Qu4DS obligations would include meeting QoS metrics constraints (i.e., strong, medium, weak) and applying the PaaS provider margin of profit (π) and fine costs (ψ, Ψ). Thus Qu4DS would be in charge of QoS translation and profiling by transparently handling lower-level details related to the underlying infrastructure. With respect Qu4DS and provider profits, Qu4DS profit could rely on adjusting the under-provisioning factor based on its own business objectives. On the other hand, the under-provisioning factor could be described in the SLA in order to let it be part of the PaaS provider business objectives. However, this requires that Qu4DS profit come from its profit margin which would increase the price of the final service delivered to customers.

Reputation The way that this thesis handles service profit maximization does not consider the provider reputation. If the service provider business objectives includes long-term provisioning, then reputation plays an important role. Nevertheless, a solution which addresses concurrently profit and reputation interests should decide which aspect will be prioritized. Future work that addresses reputation but prioritizes profit can take advantage of this thesis. For instance, dependability can be improved by enhancing its QoS assurance mechanisms, e.g., supporting availability, resource failures, replication and so forth. Another example refers to the contract rescission policy which may be changed to a less aggressive policy or even be disabled. Moreover, the benefits for enhancing reputation should be quantified in order to define how reputation impacts the profit. Although it is not trivial to coherently map reputation aspects to monetary terms, provider business objectives should serve as guidelines for solutions that address this challenge. For instance, opportunist and intermittent service providers should attribute lower values for reputation when compared to providers which aim at long-term service provisioning.

Support for Composite Services Future work may consider supporting composite services, i.e., services which are composed of other services. This extends the scope of this thesis by enabling to address more complex service relationships in addition to the current SLAs that address customers and the underlying infrastructure. However, this implies dealing with multiple SLAs which increases the complexity of the SLA management. Firstly, future work concerning composite services should add to the negotiation phase further negotiations whose SLAs define the service provider role as provider and consumer. In addition, service instantiation and provisioning should be also driven by these previous SLAs. Secondly, the service provider should add to its pricing model more pricing parameters whose complexity make it harder to increase the provider profit. These challenges are difficult to address and are currently being studied in the context of BPEL (Business Process Execution Language) compositions, where services are composed according to SLA objectives.

Third-Party Insurance Services The way in which this thesis addresses QoS assurance may be enhanced by considering third-party insurance services. The idea is to mimic conventional insurance companies by delegating to them fine payments. Thus, insurance services would cover eventual provider expenses, as fine payments, which imply losses. Insurance services can be added to the whole picture during the negotiation phase. Before booking resources for supporting the proposed contract, the service provider contacts the insurance service and tries to establish a contract which covers the losses for the contract proposal. If the insurance service accepts the contract, then the service provider acquires resources and deploys the service instance. Otherwise, the service provider should decide whether it will accept the contract proposal. In [LZL10], some issues about this subject are discussed.

Support Beyond Master/Worker Services Although the solution proposed by this thesis relies on service instances built according to the Master/Worker style, it can be extended to also support further kind of services. However, such a change may require redesigning QoS assurance mechanisms as well as monitoring metrics. For instance, in order to offer MapReduce support, the Qu4DS interface should add both map and reduce operations and the Qu4DS configuration should include details about the underlying distributed file system. In this context, a Qu4DS extension for MapReduce applications may take advantage of the Apache Hadoop framework and the Hadoop Distributed File System (HDFS). Thereby, QoS assurance would be dealt with by Hadoop built-in capabilities on top of HDFS; on the other hand, Qu4DS would address higher-level tasks regarding the SLA life-cycle management and pricing aspects.

Violation of Resource Requirements Resource providers may fail to meet the resource requirements owing to different reasons, e.g, because of applying over-provisioning in order to increase their own profit. As a consequence, QoS assurance mechanisms at PaaS-level will be compromised since they rely on service provider profiling. A trivial solution for handling the violation of resource requirements consists of relying

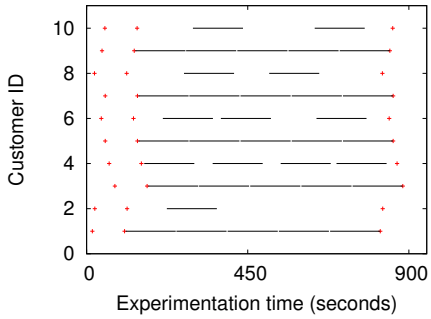
on relaxed QoS metrics instead of QoS metrics with strict constraints. More efficient solutions require adding new assurance mechanisms. For instance, if any resource requirement change is perceived, resources could be replaced in order to try to prevent execution delay. Alternatively, resources could be booked from another IaaS provider, or the contracts could be re negotiated or terminated.

Appendix A

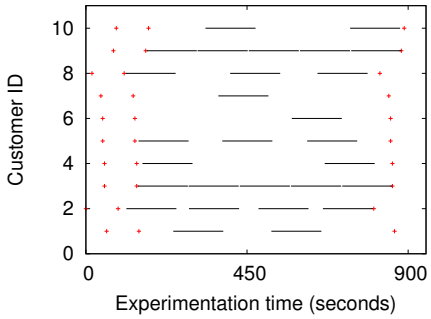
Request Schedules

This appendix depicts the request schedules created by the customer generator for the customer demand profiles *high-FT, hybrid, high-RR* as depicted by Figures A.1, A.2 and A.3 respectively.

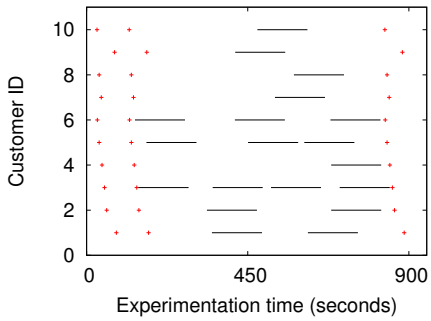
Each customer demand profile holds five request schedules. The first four request schedules depicted by (a), (b), (c), (d) sub-figures represent the different request loads $\lambda' = 1.00, 0.75, 0.50, 0.25$ whose contract durations were set to the whole experimentation duration. The last request schedule is depicted by the (e) sub-figures and represents a fixed request load $\lambda' = 1.00$ with variable contract durations. Moreover, the first cross in the figures means the time in which customers start the negotiation; the second cross means the beginning of contract durations; and the last cross means the end of contract durations.



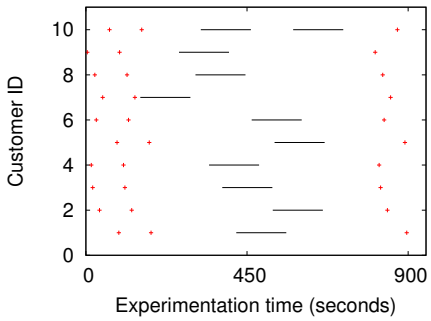
(a) $\lambda' = 1.0$ (37 requests).



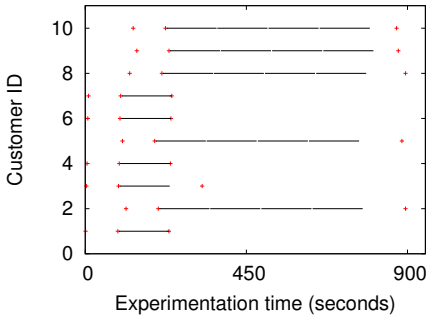
(b) $\lambda' = 0.75$ (28 requests).



(c) $\lambda' = 0.50$ (19 requests).

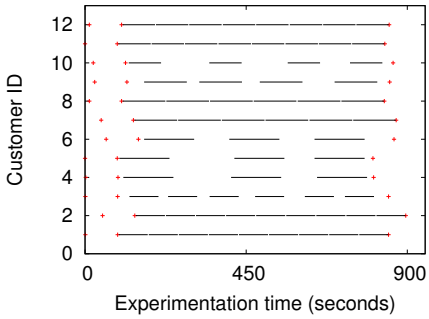


(d) $\lambda' = 0.25$ (11 requests).

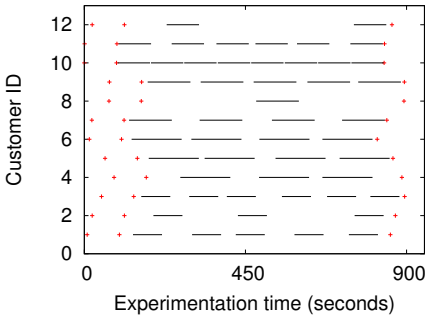


(e) $\lambda' = 1.00$ with variable contract durations.

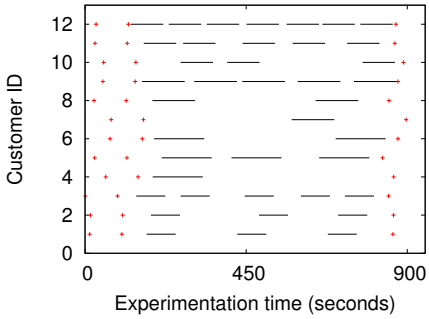
Figure A.1: Request schedules of the **high-FT** customer demand profile.



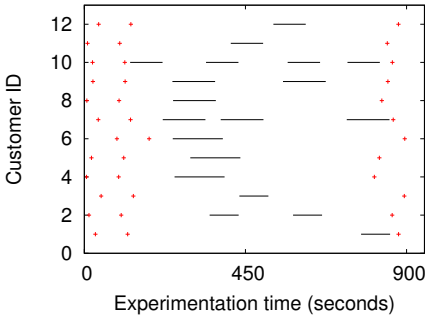
(a) $\lambda' = 1.0$ (69 requests).



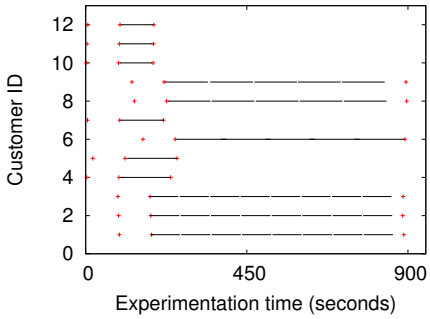
(b) $\lambda' = 0.75$ (51 requests).



(c) $\lambda' = 0.50$ (40 requests).

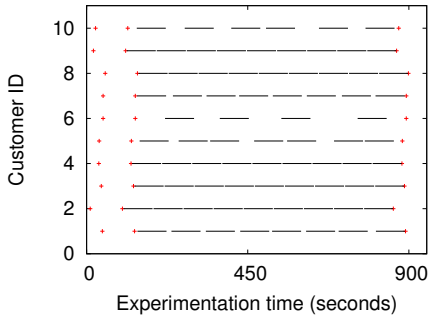


(d) $\lambda' = 0.25$ (19 requests).

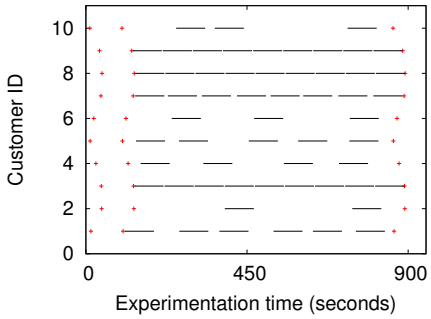


(e) $\lambda' = 1.00$ with variable contract durations.

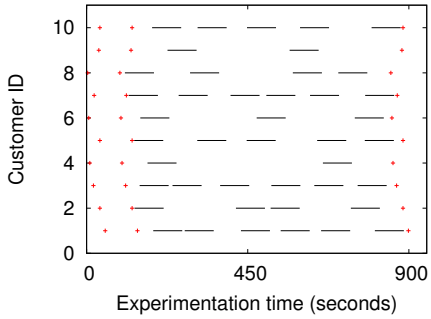
Figure A.2: Request schedules of the **hybrid** customer demand profile.



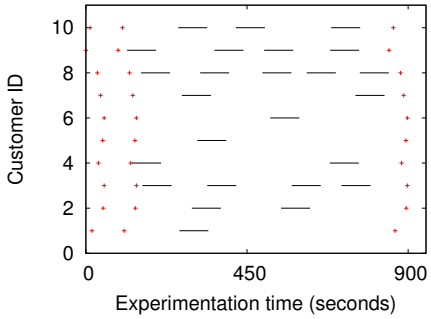
(a) $\lambda' = 1.0$ (78 requests).



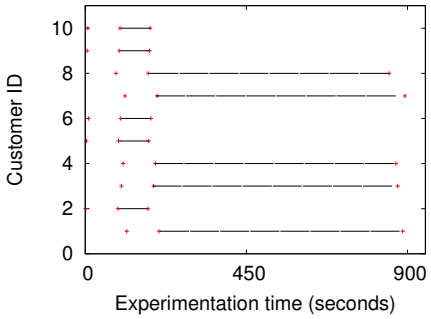
(b) $\lambda' = 0.75$ (58 requests).



(c) $\lambda' = 0.50$ (43 requests).



(d) $\lambda' = 0.25$ (25 requests).



(e) $\lambda' = 1.00$ with variable contract durations.

Figure A.3: Request schedules of the **high-RR** customer demand profile.

Appendix B

Publications and Research Activities

The research activities carried out by this thesis were part of the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE). During this thesis, various interactions with the scientific community were established which includes publications and collaboration. They are summarized next.

- International Conferences: Full Papers
  - [FPP11] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. Cost Reduction Through SLA-driven Self-Management. In *Proceedings of the 9th IEEE European Conference on Web Services (ECOWS)*, September 2011
  - [FPP10b] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. A Self-Adaptable Approach for Easing the Development of Grid-Oriented Services. In *Proceedings of the IEEE International Conference on Computer and Information Technology (CIT)*, Bradford, UK, 06 2010
- International Workshop
  - [FPP10a] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. A QoS Assurance Framework for Distributed Infrastructures. In *Proceedings of The Third International Workshop on Monitoring,Adaptation and Beyond (MONA)*. ACM, 2010
- International Conference: Poster
  - [FPP10c] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. Ensuring QoS for Service Execution on Grids. In *Proceedings of the IEEE 6th World Congress on Services (SERVICES)*, Miami, USA, July 2010
- Technical Reports

  - [IST09] INRIA – Institut National de Recherche en Informatique et Automatique, SZTAKI – The Computer and Automation Research Institute, Hungarian Academy of Sciences, and TUW – Vienna University of Technology. Basic Requirements for Self-Healing Services and Decision Support for Local Adaptation. Technical Report #CD-JRA-2.3.2, S-CUBE Project, 2009
  - [ICF+12] INRIA – Institut National de Recherche en Informatique et Automatique, CNR – Consiglio Nazionale delle Ricerche, FBK – Center for Scientific and Technological Research, UniDue – University of Duisburg-Essen, TUW – Vienna University of Technology, and UOC – University of Crete. Specifications of Policies and Strategies for Distributed and Multi-Level Adaptation. Technical Report #CD-JRA-2.3.8, S-CUBE Project, 2012. (to be published)
  - [UTC+12] UniDue – University of Duisburg-Essen, Tilburg University, CITY – City University London, INRIA – Institut National de Recherche en Informatique et Automatique, Lero – The Irish Software Engineering, Polimi – Politecnico di Milano, SZTAKI – The Computer and Automation Research Institute, Hungarian Academy of Sciences, TUW – Vienna University of Technology, UOC – University of Crete, UPM – Universidad Politécnica de Madrid, USTUTT – University of Stuttgart, and VUA – Vrije University Amsterdam. QoS and SLA Aware Service Euntime Environment. Technical Report #CD-JRA-2.3.9, S-CUBE Project, 2012. (to be published)

- Internship

  - Informatics Center, Federal University of Pernambuco, Brazil. Research activities focused on the dynamic configuration of the QoS assurance mechanisms based on the translation of non-functional requirements.

Bibliography

[AC03] Mehmet Aksit and Zièd Choukair. Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision. In *Proceedings of the 23rd International Conference on DistributedComputing Systems (ICDCSW)*, pages 84–89, Washington, DC, USA, May 2003. 17

[ACD+07] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Global Grid Forum, 2007. 6, 12, 59, 97

[Ama06a] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, January 2006. Accessed in January 2012. 5, 6, 13, 14, 24, 25, 57

[Ama06b] Amazon Web Services LLC. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/, January 2006. Accessed in January 2012. 24

[Ama08] Amazon Web Services LLC. Amazon EC2 Service Level Agreement. http://aws.amazon.com/ec2-sla/, October 2008. Accessed in January 2012. 23

[Ama09] Amazon Web Services LLC. Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/, April 2009. Accessed in January 2012. 29

[Ama12] Amazon Web Services LLC. Amazon DevPay. http://aws.amazon.com/devpay/, January 2012. Accessed in January 2012. 16

[ASF09] The Apache Software Foundation. The Apache CXF Distributed OSGi. http://cxf.apache.org/distributed-osgi.html, May 2009. Accessed in January 2012. 18

[ASF11] The Apache Software Foundation. Apache Libcloud. http://libcloud.apache.org/, May 2011. Accessed in January 2012. 97

[BAP05] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Dynamic adaptation for Grid computing. In *Proceedings of the European Grid Conference (EGC)*, pages 538–547, Amsterdam, June 2005. 17

[Bég08] Marc-Elian Bégin. An EGEE Comparative Study: Grids and Clouds – Evolution or Revolution. Technical report, CERN – Engeneering and Equipment Data Management Service, June 2008. 22, 57

[BLM08] Philip Bianco, Grace A. Lewis, and Paulo 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. 12

[Bui06] Jérémy Buisson. *Adaptation dynamique de programmes et composants parallèles*. PhD thesis, Institut National des Sciences Appliqueés de Rennes, France, 2006. 17

[CBC+04] Walfredo Cirne, Francisco Brasileiro, Lauro Costa, Daniel Paranhos, Elizeu Santos-Neto, Nazareno Andrade, Cesar De Rose, Tiago Ferreto, Miranda Mowbray, Roque Scheer, and Joao Jornada. Scheduling in Bag-of-Task Grids: The PAUÁ Case. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 124–131, Washington, DC, USA, 2004. IEEE Computer Society. 20

[CBP+10] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In Dimiter R. Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, Eliezer Dekel, Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, and Geoffrey 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. 23

[CCD+05] 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. 65, 86

[CD06] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006. 22

[CER04] CERN, LHCb Collaboration. The DIRAC (Distributed Infrastructure with Remote Agent Control) Project. http://diracgrid.org/, 2004. Accessed in January 2012. 21

[CFF+04] Karl Czajkowski, Don Ferguson, Ian Foster, Jeff Frey, Steve Graham, Tom Maguire, David Snelling, and Steve Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. Technical report, Fujitsu Limited and International Business Machines Corporation and The University of Chicago, May 2004. 12, 21

[CFI+09] CITY – City University London, FBK – Center for Scientific and Technological Research, INRIA – Institut National de Recherche en Informatique et Automatique, Polimi – Politecnico di Milano, SZTAKI – The Computer and Automation Research Institute, Hungarian Academy of Sciences, Tilburg University, Université Claude Bernard Lyon, and UniDue – University of Duisburg-Essen. Taxonomy of Adaptation Principles and Mechanisms. Technical Report #CD-JRA-1.2.2, S-CUBE Project, 2009. 17

[CFJ+08] Toni Cortes, Carsten Franke, Yvon Jégou, Thilo Kielmann, Domenico Laforenza, Brian Matthews, Christine Morin, Luis Pablo Prieto, and Alexander Reinefeld. XtreemOS: a Vision for a Grid Operating System. Technical report, XtreemOS Consortium, May 2008. 21, 56, 66

[CGP+10] Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, and the Lhcb Dirac Team. DIRAC pilot framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*, 219(6):062049, 2010. 6, 21, 28

[CH04] Humberto Cervantes and Richard S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the 26th International Conference on SoftwareEngineering (ICSE)*, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society. 18

[CIL+08] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil 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, 26

[Clo11] CloudHarmony Blog. Encoding Performance: Comparing Zencoder, Encoding.com, Sorenson & Panda. http://blog.cloudharmony.com/2011/10/encoding-performance-comparing-zencoder.html, October 2011. Accessed in January 2012. 28

[CS03] Inc. Citrix Systems. Xen. http://xen.org/, 2003. Accessed in January 2012. 23

[Del11] Simon Delamare. Grid5000 libcloud driver. http://graal.ens-lyon.fr/ sdelamar/libcloud_g5k/, April 2011. Accessed in January 2012. 66

[DFL11] Simon Delamare, Gilles Fedak, and Oleg Lodygensky. Hybrid Distributed Computing Infrastructure Experiments in Grid5000: Supporting QoS in Desktop Grids with Cloud Resources. Grid'5000 Spring School, April 2011. Accessed in January 2012. 6, 27

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008. 6, 14, 27, 29, 30, 57

[DG10] Jeffrey Dean and Sanjay Ghemawat. System and method for efficient large-scale data processing, 2010. 27

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982. 11

[DMRTV07] Giuseppe Di Modica, Valerio Regalbuto, Orazio Tomarchio, and Lorenzo Vita. Dynamic Re-Negotiations of SLA in Service Composition Scenarios. In *Proceedings of the 33rd EUROMICRO Conference on SoftwareEngineering and Advanced Applications (EUROMICRO)*, pages 359–366, Washington, DC, USA, 2007. 12

[EHL07] Clement Escoffier, Richard S. Hall, and Philippe Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. *SCC '07: Proceedings of The IEEE International Conference on Services Computing*, pages 474–481, July 2007. 18

[Fac04] Inc. Facebook. Facebook. http://www.facebook.com/, February 2004. Accessed in January 2012. 5

[Fos02] Ian Foster. What is the Grid? - A Three Point Checklist. *GRIDtoday*, 1:22–25, 2002. 19

[Fos06] Ian Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21:513–520, 2006. 21, 56

[FPP10a] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. A QoS Assurance Framework for Distributed Infrastructures. In *Proceedings of The Third International Workshop on Monitoring,Adaptation and Beyond (MONA)*. ACM, 2010. 85, 105

[FPP10b] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. A Self-Adaptable Approach for Easing the Development of Grid-Oriented Services. In *Proceedings of the IEEE International Conference on Computer and Information Technology (CIT)*, Bradford, UK, 06 2010. 105

[FPP10c] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. Ensuring QoS for Service Execution on Grids. In *Proceedings of the IEEE 6th World Congress on Services (SERVICES)*, Miami, USA, July 2010. 105

[FPP11] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. Cost Reduction Through SLA-driven Self-Management. In *Proceedings of the 9th IEEE European Conference on Web Services (ECOWS)*, September 2011. 85, 105

[Fra08] Fraunhofer Institute for Algorithms and Scientific Computing, The. WSAG4J – WS-Agreement for Java. http://packcs-e0.scai.fraunhofer.de/wsag4j/, 2008. Accessed in January 2012. 97

[GFI05] GFI Software. Monitis – IT Monitoring. http://portal.monitis.com/, 2005. Accessed in January 2012. 12

[GG11] Stéphane Genaud and Julien 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. 6, 29, 30

[GJK+05] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John 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), November 2005. 5, 28, 56

[GJK+08] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid Applications (SAGA). Technical Report GFD-R-P.90, Open Grid Forum, 2008. 19, 20, 38

[Goo04] Inc. Google. Google maps. http://maps.google.com/, October 2004. Accessed in January 2012. 5

[Goo08] Google. Google Application Engine. http://code.google.com/appengine/, January 2008. Accessed in January 2012. 13, 14, 15, 24, 25, 57

[HKSW06] Peer Hasselmeyer, Bastian Koller, Lutz Schubert, and Philipp Wieder. Towards SLA-Supported Resource Management. In *Proceedings of the*

International Conference on High Performance Computing and Communications (HPCC), pages 743–752. Springer, 2006. 6, 27, 30

[Hor01] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, 2001. Accessed in January 2012. 17

[Hue04] 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. 6, 14, 28, 30

[HWS+06] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. Taverna: a Tool for Building and Running Workflows of Services. *Nucleic Acids Research*, 34(suppl 2):W729–W732, 1 July 2006. 20

[ICF+12] INRIA – Institut National de Recherche en Informatique et Automatique, CNR – Consiglio Nazionale delle Ricerche, FBK – Center for Scientific and Technological Research, UniDue – University of Duisburg-Essen, TUW – Vienna University of Technology, and UOC – University of Crete. Specifications of Policies and Strategies for Distributed and Multi-Level Adaptation. Technical Report #CD-JRA-2.3.8, S-CUBE Project, 2012. (to be published). 106

[Inf11] Information Sciences Institute of the University of Southern California. Pegasus Workflow Management System. http://pegasus.isi.edu/cloud, July 2011. Accessed in January 2012. 57

[IST09] INRIA – Institut National de Recherche en Informatique et Automatique, SZTAKI – The Computer and Automation Research Institute, Hungarian Academy of Sciences, and TUW – Vienna University of Technology. Basic Requirements for Self-Healing Services and Decision Support for Local Adaptation. Technical Report #CD-JRA-2.3.2, S-CUBE Project, 2009. 106

[JMF09] Shantenu Jha, Andre Merzky, and Geoffrey Fox. Using Clouds to Provide Grids with Higher Levels of Abstraction and Explicit Support for Usage Modes. *Concurrency and Computation: Practice & Experience*, 21:1087–1108, 2009. 22, 56

[Jor09] Jason Jordan. Shntool – A Multi-Purpose WAVE Data Processing and Reporting Utility. http://etree.org/shnutils/shntool/, March 2009. Accessed in January 2012. 76

[Jos08] Jose Antonio Parejo and Pablo Fernandez and Antonio Ruiz-Cortés and José María García. SLAWs: Towards a Conceptual Architecture for SLA

Enforcement. In *Proceedings of the 2008 IEEE Congress on Services – Part I (SERVICES)*, volume 0, pages 322–328. IEEE Computer Society, 2008. 28

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003. 17

[KLH+10] Reto Krummenacher, Jean-Pierre Lorre, Christophe Hamerling, Alistair Duke, Matteo Villa, Francoise Baude, Elton Mathias, Virginie Legrand, Cristian Ruz, Dong Liu, Carlos Pedrinaci, Tomas Pariente Lobo, Marin Dimitrov, and Philippe Merle. Final SOA4All Reference Architecture Specification. Technical Report D1.4.2A, SOA4ALL – Service Oriented Architectures for All, 2010. 26, 30

[KMB+12] Gabor Kecskemeti, Michael Maurer, Ivona Brandic, Attila Kertesz, Zsolt Nemeth, and Schahram Dustdar. Facilitating self-adaptable Inter-Cloud management. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, February 2012. 14, 29

[KP11a] C. Klein and C. Perez. An RMS for Non-predictably Evolving Applications. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 326 –334, September 2011. 28

[KP11b] Cristian Klein and Christian 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. 6, 14, 28, 30

[KTKN11] Gabor Kecskemeti, Gabor Terstyanszky, Peter Kacsuk, and Zsolt Neméth. An Approach for Virtual Appliance Distribution for Service Deployment. *Future Generation Computer Systems*, 27(3):280 – 289, 2011. 6, 22, 23, 29, 30

[KW10] Constantinos Kotsokalis and Ulrich Winkler. Translation of Service Level Agreements: A Generic Problem Definition. In Asit Dan, Frédéric Gittler, and Farouk 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. 6, 26

[Lap85] J. C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *Proceedings of 15th International Symposium on Fault-Tolerant Computing (FTSC)*, pages 2–11, 1985. 13, 14

[LB09] Sonja Lehmann and Peter Buxmann. Pricing Strategies of Software Vendors. *Business & Information Systems Engineering*, 1(6):452–462, 2009. 16

[LDBK10] Sonja Lehmann, Tobias Draisbach, Peter Buxmann, and Corina Koll. Pricing Models of Software as a Service Providers: Usage-Dependent Versus Usage-Independent Pricing Models. In Gurpreet Dhillon, editor, *Proceedings of the 8th Annual Conference on Information Science, Technology & Management (CISTM)*, August 2010. 16

[LKD+03] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web Service Level Agreement (WSLA) Language Specification. Technical report, IBM, 2003. 6, 12, 59

[LLJ10] André Luckow, Lukasz Lacinski, and Shantenu 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. 6, 21, 28, 30, 57

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Survey*, 22(4):321–374, 1990. 19

[LTH09] Hui Li, Wolfgang Theilmann, and Jens Happe. SLA Translation in Multilayered Service Oriented Architectures: Status and Challenges. Technical Report IB 2009-8, University of Karlsruhe, 2009. 6, 27

[LZ10] Young Choon Lee and Albert Y. Zomaya. Rescheduling for Reliable Job Completion with the Support of Clouds. *Future Generation Computer Systems*, 26:1192–1199, October 2010. 6, 14, 27

[LZL10] Min Luo, Liang-Jie Zhang, and Fengyun Lei. An Insuanrance Model for Guranteeing Service Assurance, Integrity and QoS in Cloud Computing. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 584 –591, july 2010. 99

[ME11] Thijs Metsch and Andy Edmonds. Open Cloud Computing Interface – Infrastructure. Technical Report GFD-P-R.184, Open Grid Forum, 2011. 24, 57, 97

[MFG10] Mario Macías, Josep Oriil Fitó, and Jordi 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. 25, 26, 30

[MG10] Mario Macías and Jordi Guitart. Maximising Revenue in Cloud Computing Markets by means of Economically Enhanced SLA Management. Technical Report UPC-DAC-RR-2010-32, Universitat Politècnica

de Catalunya. Departament d'Arquitectura de Computadors, 2010. 25, 26

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology, September 2011. 22, 23

[Mic08] Microsoft, Inc. Windows Azure. http://www.windowsazure.com/, October 2008. Accessed in January 2012. 13, 14

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, 2004. 17

[NMS+05] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Technical Report GFD-R.052, Open Grid Forum, 2005. 22

[Nur09] Nurmi, Daniel and Wolski, Rich and Grzegorczyk, Chris and Obertelli, Graziano and Soman, Sunil and Youseff, Lamia and Zagorodnov, Dmitrii. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 124–131, 2009. 57

[Ope07] Open Service Oriented Architecture. SCA Service Component Architecture – Assembly Model Specification. http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf?version=1, March 2007. Accessed in January 2012. 18

[Ora07] Oracle. VirtualBox. https://www.virtualbox.org/, January 2007. Accessed in January 2012. 23

[OSG07] OSGi Alliance. OSGi Service Platform – Service Compendium (Release 4, Version 4.1). http://www.osgi.org/Download/File?url=/download/r4v41/r4.cmpn.pdf, April 2007. Pages 281–315. Accessed in January 2012. 18

[PAB11] Mike P. Papazoglou, Vasilios Andrikopoulos, and Salima Benbernou. Managing Evolving Services. *IEEE Software*, 28(3):49–55, 2011. 18

[Par72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems Into Modules. *Commun. ACM*, 15:1053–1058, December 1972. 11

[Per05] Paulo Rogério Pereira. Service Level Agreement Enforcement for Differentiated Services. In *Proceedings of the 2nd International Workshop of the EURO-NGI Network of Excellence*, Villa Vigoni, Italy, July 13-15 2005. 28

[PG03] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing, Introduction. *Commun. ACM*, 46(10):24–28, 2003. 18

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40:38–45, 2007. 18

[RFCRJ04] Nelson Rosa, Paulo Freire Cunha, and George Ribeiro Justo. An Approach for Reasoning and Refining Non-Functional Requirements. *Journal of the Brazilian Computer Society*, 10:62–84, 2004. 10.1007/BF03192354. 13, 41

[RH06] Inc. Red Hat. KVM – Kernel-based Virtual Machine. http://www.linux-kvm.org/, 2006. Accessed in January 2012. 23

[Sal99] Inc. Salesforce.com. SalesForce. http://www.salesforce.com/, January 1999. Accessed in January 2012. 6, 16, 24, 25, 57

[Sch10] Schubert, L. Et Al. The Future Of Cloud Computing, Opportunities for European Cloud Computing Beyond 2010. Technical report, European Commission, Information Society & Media, 2010. 22, 23

[SNM+02] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Proceedings of the 3rd International Workshop on Grid Computing (GRID)*, GRID '02, pages 274–278, London, UK, 2002. Springer-Verlag. 19, 20, 22

[SS09] Vladimir Stantchev and Christian 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. 6, 26, 30

[Szy03] Clemens Szyperski. Component Technology: What, Where, and How? In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society. 18

[TBB+08] A Tsaregorodtsev, M Bargiotti, N Brook, A C Ramo, G Castellani, P Charpentier, C Cioffi, J Closier, R G Diaz, G Kuznetsov, Y Y Li, R Nandakumar, S Paterson, R Santinelli, A C Smith, M S Miguelez, and S G Jimenez. DIRAC: a community grid solution. *Journal of Physics: Conference Series*, 119(6):062048, 2008. 6, 21, 28

[TGT+10] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A Tighter Analysis of Work Stealing. In *Proceed-*

ings of the 21st International Symposium on Algorithms and Computation (ISAAC), volume LNCS no XXXX, Jeju Island, Korea, Dec 2010. Springer Verlag. 28

[The11a] The FLAC project. Free Lossless Audio Codec (FLAC). http://flac.sourceforge.net/, 2011. 75

[The11b] The Xith Open Source Community. Ogg Vorbis Audio Format. http://www.vorbis.com/, October 2011. 75

[The11c] The Xith Open Source Community. The Ogg Container Format. http://xiph.org/ogg/, October 2011. 75

[THK+10] 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. 6, 26, 30

[Uni12] University of Wisconsin-Madison. Condor Project. http://research.cs.wisc.edu/condor/, January 2012. 21

[UPF+08] Université Claude Bernard Lyon, Polimi – Politecnico di Milano, FBK – Center for Scientific and Technological Research, SZTAKI – The Computer and Automation Research Institute, Hungarian Academy of Sciences, INRIA – Institut National de Recherche en Informatique et Automatique, CNR – Consiglio Nazionale delle Ricerche, UniDue – University of Duisburg-Essen, and USTUTT – University of Stuttgart. State of the Art Report, Gap Analysis of Knowledge on Principles, Techniques and Methodologies for Monitoring and Adaptation of SBAs. Technical Report #PO-JRA-1.2.1, S-CUBE Project, 2008. 17

[UTC+12] UniDue – University of Duisburg-Essen, Tilburg University, CITY – City University London, INRIA – Institut National de Recherche en Informatique et Automatique, Lero – The Irish Software Engineering, Polimi – Politecnico di Milano, SZTAKI – The Computer and Automation Research Institute, Hungarian Academy of Sciences, TUW – Vienna University of Technology, UOC – University of Crete, UPM – Universidad Politécnica de Madrid, USTUTT – University of Stuttgart, and VUA – Vrije University Amsterdam. QoS and SLA Aware Service Euntime Environment. Technical Report #CD-JRA-2.3.9, S-CUBE Project, 2012. (to be published). 106

[UTU+08] Université Claude Bernard Lyon, TUW – Vienna University of Technology, UPM – Universidad Politécnica de Madrid, Polimi – Politecnico di Milano, INRIA – Institut National de Recherche en Informatique et Automatique, UniDue – University of Duisburg-Essen, FBK – Center for Scientific and Technological Research, SZTAKI – The Computer and Automation Research Institute, Hungarian Academy of Sciences, UOC –

University of Crete, CNR – Consiglio Nazionale delle Ricerche, and USTUTT – University of Stuttgart. Survey of Quality Related Aspects Relevant for Service-based Applications. Technical Report #PO-JRA-1.3.1, S-CUBE Project, 2008. 12

[Vie09] John Viega. Cloud Computing and the Common Man. *Computer*, 42:106–108, August 2009. 22

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39:50–55, December 2008. 22

[W3C10] W3C Working Group. Web Services Architecture. http://www.w3.org/TR/ws-arch/, 2010. 18

[WB12] Linlin Wu and Rajkumar Buyya. *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, chapter Service Level Agreement (SLA) in Utility Computing Systems, pages 1–25. IGI Global, 2012. 12, 13

[Xip11] Xiph.Org Foundation. Ogg Vorbis Tools. http://www.vorbis.com/, November 2011. 77

[Yan03] Jian Yang. Web Service Componentization. *Communications of the ACM*, 46(10):35–40, 2003. 18

[Zen10] Zencoder, Inc. Zencoder – Cloud Video Encoding/Transcoding Software as a Service. http://zencoder.com/, May 2010. Accessed in January 2012. 5, 6, 13, 14, 25, 28

List of Figures

List of Tables

Abstract

Services enable building loosely-coupled and dynamic applications in distributed environments. Service-Level Agreements (SLAs) are used to define service relationships by describing how services should behave. Moreover, SLAs include the Quality of Service (QoS) that should be delivered along with the service. However, managing service executions on top of distributed infrastructure while meeting agreed QoS is challenging. Firstly, QoS metrics should be mapped to low-level system configurations in order to enable building QoS assurance mechanisms. Secondly, service execution should deal with failures and load variations. In addition to these issues, service execution should be driven by pricing aspects since profit increase is an important concern for service providers. Current approaches that handle service execution support neither the whole SLA life-cycle nor profit augmentation.

This thesis proposes an autonomous solution for managing service execution in distributed infrastructures by aiming at increasing the provider profit. In particular, this thesis supports the full SLA life-cycle based on: *(i)* SLA translation under pricing constraints; *(ii)* mechanisms which ensure fault-tolerant and performance QoS; and *(iii)* resource acquisition and allocation driven by contract proposals and requests. In order to realize this solution, this thesis describes the design and implementation of the Qu4DS (Quality Assurance for Distributed Services) framework. Qu4DS includes a rich set of SLA management functionalities that provides a higher-level support for service developers. Moreover, Qu4DS is evaluated on top of Grid5000 and the results show that Qu4DS is able to increase the provider profit while meeting SLAs in different scenarios.