Automated Hierarchical Service Level Agreements

Generic Management Principles and Application to Multi-Domain Infrastructure-as-a-Service

Bei der **Fakultät für Informatik**
der
Technische Universität Dortmund
zur Begutachtung eingereichte

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

von

Konstantinos (Costas) Kotsokalis

Oktober 2010

Hauptreferent: Prof. Dr.-Ing. Ramin Yahyapour

Στον πατέρα μου,
Αθανάσιο Κωτσοκάλη

και τη μητέρα μου,
Κωνσταντίνα Σακελλαράκη-Κωτσοκάλη

για το θεμέλιο
και το παράδειγμα.

Preface

The research presented in this dissertation concerns the area of Service Computing; more specifically, it contributes to the topic of enabling IT service stacks with dependability, such that they can be used even further in pragmatic business environments and applications. The instrument used for this purpose is that of a *Service Level Agreement* (SLA).

The main focus is on *SLA Hierarchies*, which reflect corresponding *Service Hierarchies*. SLAs may be established manually, or automatically among software agents; it is mainly the latter case that is considered here. The thesis contributes by means of a formal problem definition for the construction of SLA hierarchies using a *translation* process, a management architecture, a formal model for defining penalties, and a representation that facilitates the processing of SLAs.

Using these tools it is shown that automated SLA management in hierarchical setups is possible, through an application to *Multi-Domain Infrastructure as a Service*. Within this specific technical area, different SLA-based resource capacity planning approaches are examined via simulation – both for online and offline planning. The former case concerns normal runtime operations, and the thesis examines two greedy algorithms with regard to their energy-savings efficiency and their performance. In the latter case, a resource-scarce environment is simulated with the purpose of minimizing penalties from already established SLAs. This is achieved via formally-defined combinatorial models, which are solved and compared to two greedy algorithms.

The research that lead to this thesis was conducted from late 2004 to early 2008 in Athens, Greece; from then on and until late 2010 in Dortmund, Germany, as part of the SLA@SOI EU/FP7 Integrated Project (contract No. 216556) [1]. It is presented at Technische Universität Dortmund (TU-Dortmund), Department of Computer Science, under the supervision of Prof. Dr.-Ing. Ramin Yahyapour.

Acknowledgements

There are many people one must usually thank in this section of a doctoral dissertation. No exception to this rule is to be found here.

First of all, I would like to thank my advisors and supervisors during these years of research; namely, Prof. Dr.-Ing. Ramin Yahyapour, who supervised my work at the Dortmund University of Technology, Germany, and Prof. Dr. Panayiotis Tsanakas who was my supervisor at the National Technical University of Athens, Greece. I owe them a lot for their help and guidance during this process.

I am grateful to my colleagues at ITMC/TU-Dortmund, who contributed towards a stimulating research environment, helped me enormously whenever speaking German was necessary, but also stood by me as good friends during the last and most difficult part of my work. They are all nothing short of amazing. Special thanks go to Philipp Wieder; his tireless support during my time in Germany made a huge difference.

It would be hard to achieve this without the constant support from the many good friends I have been blessed with, outside my working environment. I hope they will understand that I cannot be too verbose here and mention each one separately, but they must all know that their encouragement was precious.

I cannot be thankful enough to my family. This thesis is dedicated to my parents, Athanasios and Konstantina, for all the reasons that words can hardly express. My sisters, Vassiliki and Dimitra, provided support, motivation and a voice of reason that was much needed. They played a crucial role, and for that I thank them wholeheartedly. The same applies to my brother in law, Makis Lazos, and my beloved nephews George and Thanos.

Last, but not at all least, I would like to thank my partner in life, Nadia Nikolopoulou. There is no possible way to describe, even to the tiniest bit, all the reasons why this thesis would have never been written if it was not for her. For all the patience, encouragement, support, inspiration and motivation, Nadia, I thank you. You were right in everything.

Contents

List of Figures

Part I

Introduction

Chapter 1

Context

The vision of the present dissertation is to further enable IT service hierarchies with dependability, in a scalable manner. As a quick introduction to this vision and the overall context, it is beneficial to provide the background of the respective research. This initial chapter briefly discusses the scientific motivation for the dissertation. Then, it continues to provide information about what is within the thesis' scope, and what is not.

1.1 Motivation

The research presented in this doctoral dissertation is primarily motivated by the wish to enable Service Oriented Infrastructures (SOIs) with dependability, for the purpose of *automated* service binding. More specifically, service computing envisages diverse service ecosystems, where software agents automatically discover and bind each other, to achieve reusability via composition. If this vision is to be achieved, some measure of certainty needs to be integral to the discovery and binding process. The other option, *best effort* usage, is considered suboptimal and difficult to apply to real-world businesses, as it offers no guarantees and therefore could pose risks to the viability and sustainability of a business.

Service Level Agreements (SLAs) are an instrument that can be used to approach such goals of increased certainty. They represent the electronic equivalent of business contracts between service providers and service customers. As such, they can describe what exactly is the service, what the service provider guarantees, what are the obligations of the customer, what are the penalties if the guarantees are not honored, and so on. The complex relationships between different services as they are forming service chains and hierarchies can be reflected onto SLAs, forming equally complex SLA hierarchies. This way, it will become pos-

sible to enhance complete service stacks, make them dependable, and advance service computing overall. SLAs are meant to be negotiated, just like normal contracts, and be used during the service's lifetime as an official means to describe expectations and guide the service provisioning and monitoring processes.

The information contained within SLAs includes details regarding the implementation of the service, for which the provider knows how to map these onto resource requirements. Thus, SLAs *can be used as a tool to enable automated decision-making for purposes of online resource planning*. Adaptive resource capacity management is not a new area, yet it has had limited success so far in achieving integration with business objectives and the way businesses operate. SLAs provide necessary formalisms to enable this connection to business objectives, as they typically include provisions for pricing, penalty definitions, etc.

When the service refers to infrastructure resources that are offered over the network (*"Infrastructure as a Service"* - IaaS), the link between SLAs and resource management becomes further prominent. Resources in this case could be many different things, such as computing nodes, storage, network circuits, sensors, scientific instruments, and so on. IaaS is a trend that has seen new heights recently with advances in resource virtualization and the prevalence of *Cloud Computing*. This area is one of the targets for the present thesis.

1.1.1 The SLA@SOI EU Project

Most of the research presented within this thesis took place as part of the SLA@SOI EU-funded Integrated Project (grant agreement: 216556). SLA@SOI is a three-year project running from 2008 to 2011, aiming to enable service hierarchies with dependability, transparent SLA management, and automation. The project takes a holistic view on service management, using a three-layer approach: An infrastructure layer, consisting of finite, countable physical and virtual infrastructure resources; the software layer, consisting of software executing on this infrastructure, possibly also constrained by limitations such as licensing options; and a business layer, that augments software and infrastructure services with business concepts and exposes them as complete products ready to be sold to customers. Each layer represents distinct providers, or distinct departments within the same provider. Eventually, for a product (business-enabled service) sold under guarantees foreseen by some SLA, all involved services from the two lower layers should be provisioned under appropriate SLAs, so that the top-level one is fulfilled (Figure 1.1). Each arrow in this figure represents the establishment and runtime manage-

ment of SLAs, including (re-)negotiation, monitoring and alerting, arbitration, etc.

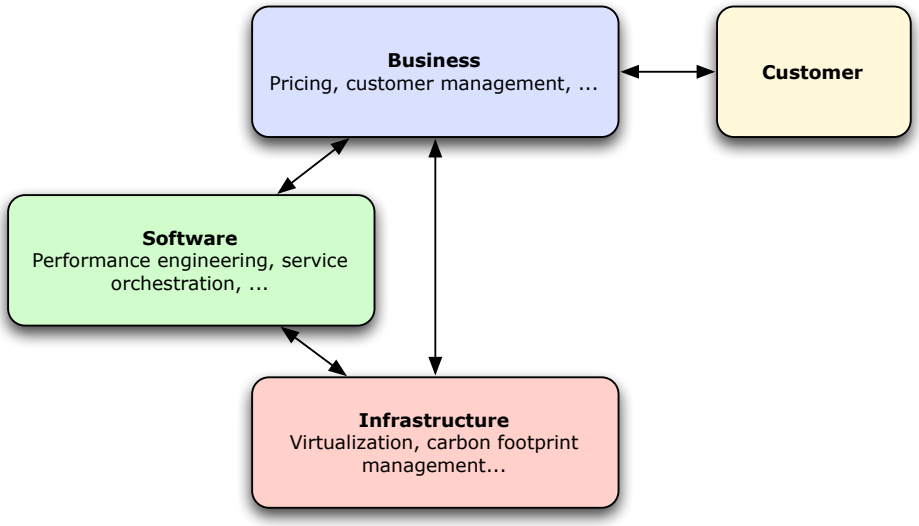

Figure 1.1: Layered SLA establishment of SLA@SOI

The work presented herein is only a fraction of the work completed (and still ongoing as of September 2010) within the project. For instance, this dissertation does not touch upon business concepts (with the exception of cost and penalties), software service orchestration, performance engineering, or infrastructure monitoring.

1.2 Scope and Scientific Contribution

The thesis has two different purposes. First, it tries to create some new formalisms and methods, that apply to SLAs independent of the domain and can be useful in combination with other ongoing research. Second, focusing on infrastructure services, these formalisms / methods are used and extended with algorithms for negotiation-time resource planning, and for resource-planning during exceptional situations when resources become scarce.

Looking at the topic informally and from a very high level, Figure 1.2 illustrates the context of this work, and the questions it addresses.

The scope and the respective scientific contributions of the thesis can be summarized in the following list:

SLA Hierarchies: A concrete definition of the SLA hierarchies and the

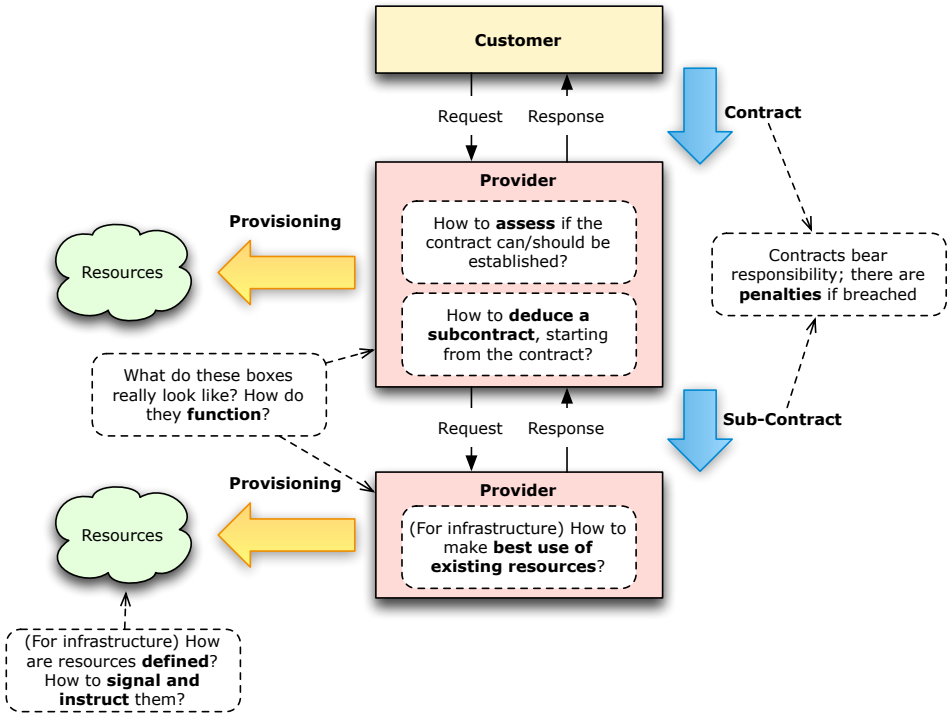

Figure 1.2: High-level scenario

process of "translating" from one SLA to others, is presented. This text is available in Chapter 5.

Architecture: A complete, self-contained management architecture for SLAs is presented in Chapter 6. This architecture takes into account requirements for multi-level SLAs, and can be extended to fit any domain that requires SLA management.

Penalty Model: A penalty model supporting SLA term interdependencies, fairness, and business value declarations, is presented in Chapter 7.

In-memory SLA Model: In contrast to SLA models for on-the-wire representation, a generic and flexible model is necessary for in-memory management of highly-complex SLAs. Such a model should allow the efficient handling of their complexity. A proposal in this area is presented in Chapter 8.

Resource Model and Reservation Primitives: For this work there was required a formalism to express resource groupings and time relevance, so that it can be used for co-allocation and advance reser-

vation. Additionally, such a model can be directly used within SLAs to express guarantee terms for the provisioning of resources. Chapter 9 presents a standards-compatible model for these purposes.

IaaS SLA Planning: The planning process differs significantly in different domains. Here, the focus is on methods that can be applied to the problem of SLA planning for infrastructure services. The topic is researched considering two different use cases: First, normal operations negotiation-time planning, where existing SLAs are never violated on purpose; and the focus is on energy consumption and planning performance. Second, planning under an extreme situation where resources have become scarce (e.g. massive resource failure, or submission of non-negotiated SLAs to the system). In this latter case, some SLAs may be discarded partially or in full, so that eventual penalties are minimized. This is an *offline* process, and for its solution this work proposes a combinatorial (Integer Linear Programming - ILP) approach. Chapters 10 and 11 elaborate on these issues.

Relevant, but *out of scope* for this dissertation, are:

- Negotiation protocols;
- Models for on-the-wire or on-the-screen SLA representation;
- Management necessities for service discovery, composition, binding, provisioning and monitoring.
- Technology and approaches for *implementing* infrastructure adjustment capabilities such as scaling, virtual machine migration, etc.

The next chapter includes a motivating example, that will be used in Part II for illustration purposes. In Part III a different example will be used, to fit the narrower scope of infrastructure services.

What comes next

In the upcoming Chapter 2, the essentials of service and cloud computing will be discussed. The remainder of Part I then continues to discuss SLAs in general in Chapter 3, and provide a more detailed description of the scientific problems that the thesis contributes to – also in the context of various example and more realistic use cases – in Chapter 4.

Chapter 2

Service and Cloud Computing

This Chapter serves as a more elaborate introduction to service orientation and service computing. Fundamental definitions are discussed and relevant concepts such as workflows and composition are provided. The Chapter then proceeds to explain the essentials of cloud computing and how it relates to service computing. Finally, outsourcing of infrastructure services is introduced.

2.1 Service-Oriented Computing

Service-Oriented Computing (SOC) is the computing paradigm that utilizes services as fundamental elements for developing applications [2]. The question naturally arises, how is a *service* defined. In [3], Zhang et al. start their book by reviewing a number of definitions from prior art, and eventually define *Information Technology* (IT) services as follows:

> Services represent a type of relationships-based interactions (activities) between at least one service provider and one service consumer to achieve a certain business goal or solution objective.

Service providers offer and implement the service, while service consumers utilize it ("consume" it).

In general, one could informally say that SOC is the IT discipline that concerns with exposing functionality for others to utilize, possibly in the context of some larger application. As such, any activity offered by one entity to another, is a service. In recent years, SOC has changed to a large extent the way business is conducted. The main reason is the fact

that it is possible to offer complete applications by combining services in a proper way, using the functionality offered by one to cover for functionality missing from another. This makes it easier for companies to focus on their area of expertise and integrate necessary additional functionality by means of reusing existing tools offered by others. It also introduces the notion of a *service dependency*; that is, the fact that a service requires some other service in order to execute as designed. If this concept is recursively applied to the services within an application, the result is a *service chain*. Such formations are very typical nowadays. The term is an analogy to *supply chains*, but instead of referring to traded products, it refers to services.

2.2 An Example Scenario

Before further discussing service dependencies, we introduce an example scenario to be used throughout the thesis for reasons of demonstration. This example consists of an infrastructure service provider, a software service provider, and a customer.

The software service provider offers a geo-positioning facility, consisting of the following services:

1. *Geo-coding (GC):* Given an address it returns the respective longitude and latitude, and vice-versa.

2. *Route planning (RP):* Given two sets of coordinates representing two different geographical points, it returns the shortest route from the first point to the second.

3. *Address-based route planning (AB):* Performs the same functionality with coordinates-based route planning, but accepts full addresses instead of coordinates.

Address-based route planning makes use of the other two services: First, it invokes the geo-coding to convert addresses to geographical coordinates; then, uses these coordinates to invoke the route planning service. The latter two rely on a database server, which they must invoke to retrieve necessary information.

On the same time, the three geo-positioning software services execute within a web application server, which they never “invoke”. The application server, in turn, executes within a physical or virtual server – and the same applies to the database server. We assume that they run on two distinct servers, and that these servers are both offered by the

infrastructure provider. Figure 2.1 shows the different actors involved in this scenario.

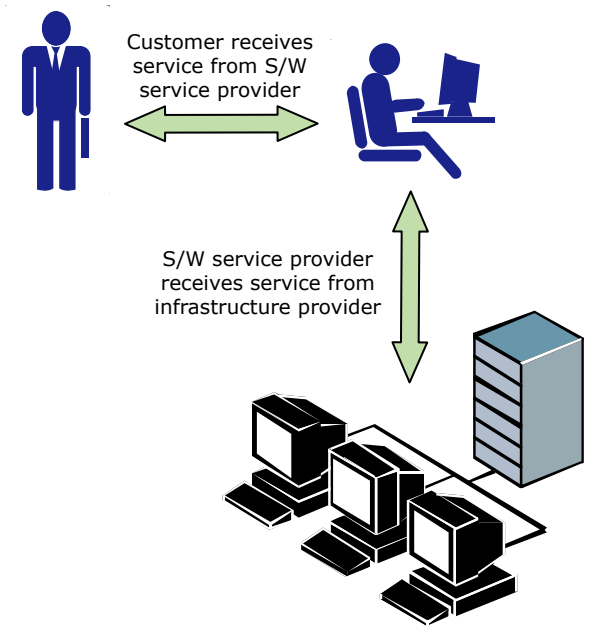

Figure 2.1: Actors of the example scenario

In the upcoming Section, this example will be used to illustrate the various types of dependencies between different services.

2.3 Service Dependencies

Service dependencies can be *explicit*, or *implicit*. Explicit dependencies play a central role in service-oriented computing. The assumption here is that a service's logic (the functionality it delivers) depends on some other service by means of composition. That is, there are *service invocations* involved. It is expected that each service is *invokable* over some interface expressed in well-defined languages (e.g. the *Web Services Definition Language* - WSDL [4]). With a pipeline of input and output of such services, it becomes possible to create a *software service composition*, in the form of a workflow. Oftentimes such compositions are referred to as *Business Processes*, and are in general *orchestrated* by means of a workflow engine. In recent years, the *Web Services Business Process Execution Language* [5] (WSBPEL) has been the most widely used specification for this purpose.

In the example from Section 2.2, service *AB* invokes service *GC* with

the given addresses, and expects to receive two sets of coordinates (corresponding to the two addresses). If one, or both the addresses cannot be found and mapped to geographical coordinates, an error is returned. Should both addresses be properly resolved, service *RP* is invoked and a route between the two is returned. Figure 2.2 illustrates this workflow of invocations and checkpoints, in the form of a flowchart.

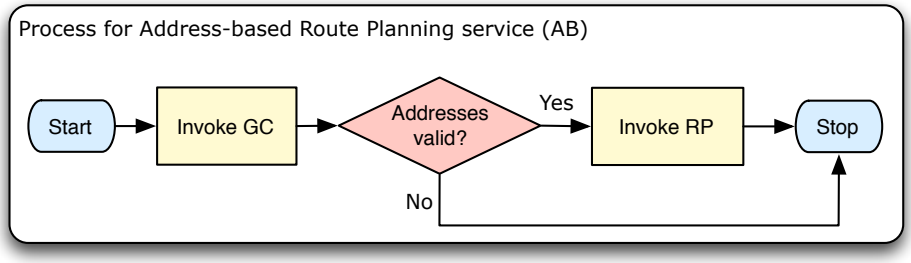

Figure 2.2: Flowchart of address-based route planning service process

Implicit dependencies are, on the other hand, typically related to middleware and infrastructure services (without which a higher-layer service cannot operate at all), and also services used for reasons of redundancy. For instance, web site development agencies may do the site hosting themselves, or outsource it to specialized data centers. Any software relies on some hardware on which it can execute. When it comes to redundancy, the standby services set up for this reason are normally not used, but their existence affects the service that depends on them, in the case of a failure.

In what follows, the services that depend on others will be referred to as *dependents*, and services on which others depend will be referred to as *antecedents*.

Although it is possible to perform this rough classification of explicit and implicit dependencies, it is not possible to refine it further and try to identify in greater granularity the relations between dependent services. Rather, the notion of *dependency* itself is enough to construct a *(Service) Dependency Graph* [6, 7]. Figure 2.3 illustrates the dependency graph for the example scenario elaborated earlier. Relationships "*invokes*" and "*queries*" denote explicit dependencies, whereas "*executes in*" is an implicit relationship.

Throughout this thesis SDGs will also be referred to as *Service Hierarchies*, to denote the layered structure that one can also see in Figure 2.3.

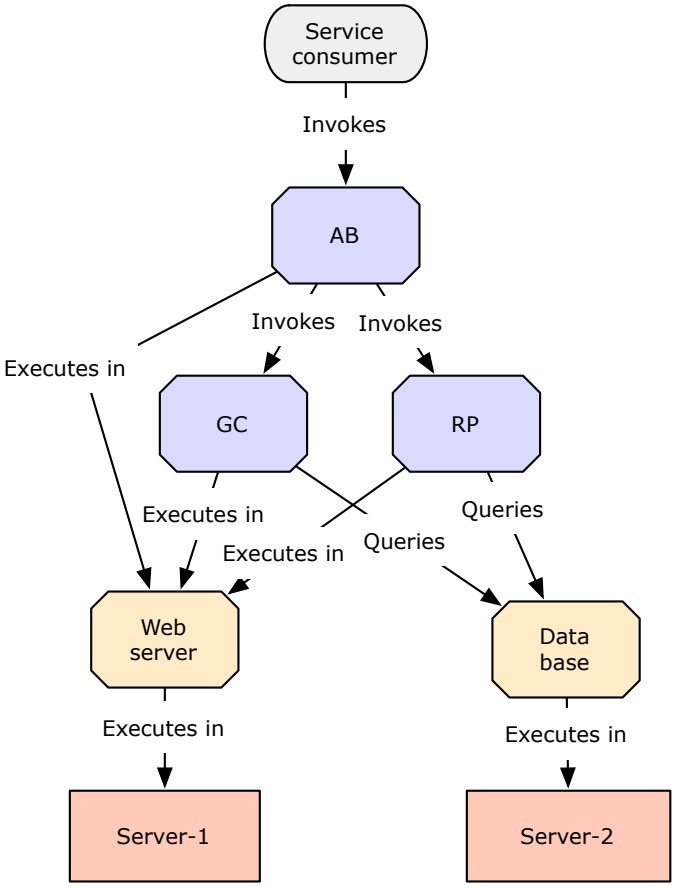

Figure 2.3: Service dependency graph for example scenario

2.4 Cloud Computing

Cloud Computing can be seen as an extension of the concepts of Service-Oriented Computing, enriching them with notions of *virtualization* and *massive scalability*. It is difficult to provide an exact definition of the "cloud computing" term, also due to the fact that at this time it is the subject of extreme marketing, therefore the term is typically overloaded by many of its users [8]. Although it started referring only to computing infrastructure, when Amazon™released *Elastic Compute Cloud*™ (EC2) [9] and *Simple Storage Service*™ (S3) [10], currently the term is also being used for highly scalable software services offered under similar conditions.

Cloud computing is frequently also used as a term interchangeably with *Utility Computing* [11], which essentially refers to "pay as you go" usage models. With utility computing, the user initially leases a minimal

amount of resources, and then she can dynamically add more to the utilized resource pool. In a similar manner, dynamically removing resources from the pool is also supported. As such, the user eventually pays only for the resources that she uses.

Finally, cloud computing has been often related to *Grid Computing* [12], especially during its early stages. The latter refers to a resource sharing model, which is closer to *Platform as a Service* (Paas). Grid computing was very popular for the last decade, but eventually never managed to become a successful model in the business world. Conversely, cloud computing appears to be very suitable for business exploitation, and as such the respective uptake is very significant at the time. Grid computing remains, nevertheless, the preferred usage paradigm for large-scale scientific computing.

A complete definition of cloud computing is not within the scope of the work presented in this thesis. The focus of Part III is on *Infrastructure as a Service* (IaaS), where the service always refers to infrastructure resources. These may consist of computing power in the form of countable *CPU cores* represented by *Virtual Machines* (VMs), storage measured in well-defined units such as MegaByte (MB), or even sensors, virtualized scientific instruments, etc. In general, the addressed field is that of finite, countable, tangible resources which are required as an infrastructure layer on which software executes.

A key concept for IaaS is virtualization. The infrastructure that the consumer is using may be anywhere in the world (assuming no locality constraints have been agreed with the provider). Additionally, it may well be the case that the real hardware resource is shared between many consumers, although this is something that the users cannot see directly. This technology has been instrumental to the uptake of cloud computing and IaaS, as it offers important management capabilities. Providers can resize their infrastructure dynamically, with fine granularity. Enforcing usage limits to users (quota) is not depending on the operating system that the user executes her applications on, but rather on the software that implements the virtualization. Thus, it is possible to do things such as changing the amount of volatile memory of a virtual machine, to scale up and down when needed without restarting the system or performing any manual (hardware-related) operations. Additionally, it is possible to suspend VMs for implementing check-pointing mechanisms, clone and migrate VMs to other physical infrastructure for load-balancing, etc. In general, such virtual environments offer significant flexibility and management convenience, hence also allow new techniques to be implemented for advancing the ways people use IT today.

A common use case of IaaS is that of a company which needs computing resources, but either the usage patterns present sudden scaling requirements (e.g. simulations for a design that was just concluded), or the requirements in the near future are simply unknown. Startup companies are a good example; when operations commence, typically it is not known how prospective users will react, and if the offered product will be successful. Thus, it is economical and makes good business sense to start with small infrastructure, and then scale it if needed. Nevertheless, such infrastructure scaling normally takes a lot of time to design, and the procurement itself is also time-consuming. On the other hand, using IaaS, it is possible for a company to size up its infrastructure in a very short time, by leasing resources from an IaaS provider. For instance, it can start load-balancing among 10 VMs, and then initiate another 10 following an agreement with the provider. It is merely a configuration change in the load-balancer, after which it is possible to distribute requests to double the machines, therefore decreasing by 50% on average the workload on each of the servers previously responding to incoming requests.

Outsourcing infrastructure requirements is often a good business idea, also due to the avoidance of administration costs (and the costs associated with initial equipment purchase). Economies of scale ensure that a provider of infrastructure services can achieve much lower management costs per resource, than what a company with a small in-house infrastructure would have to pay for the same purpose. Provided that these savings are also affecting the pricing of IaaS, the knock-on effect has positive results to the pricing of the services of the company who rents the infrastructure, while it may also lead to increased profits.

Finally, infrastructure outsourcing is an attractive option for companies who provide complex software systems as a service. Typically, performance or other requirements differ from one customer to another. This translates to different requirements from the underlying infrastructure, with a typically very large number of different implementation options [13]. Outsourcing infrastructure requirements enables companies to have access to possibly exotic equipment, that otherwise they might purchase at a high cost, to use only rarely.

2.5 Multi-Domain Infrastructure-as-a-Service

The main concept underlying cloud computing (and therefore, IaaS) is one of abstraction. The user does not know (and is not concerned) exactly where her data resides, where it is processed, etc. As long as it is safe from unauthorized access, and ubiquitously present with reasonable per-

formance, the exact location and domain is not relevant. This property allows that different IaaS domains collaborate in groups and exchange workload.

It is therefore assumed that an IaaS domain which is in need of additional resources *can become itself a customer to other IaaS domains*, and request to outsource to them (a "*variable-role*" assumption). It also applies that an entity can perform as a broker of resources from other domains, without owning any itself. Domains can be *administrative* (i.e. different providers), or *management* (i.e. different operational units within the same provider). The two are not differentiated in this work. The term *domain* is used here to imply resource management independence. Within a single domain, the provider (or unit) can allocate, utilize and release resources based merely on "local" knowledge. In addition, providers (or units) from other domains do not have rights over these resources, therefore using them means that they have to request permission to do so.

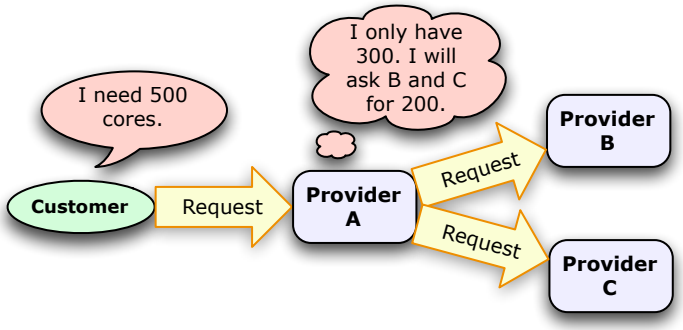

Figure 2.4: Multi-domain resource provisioning

As evident, there is a straightforward analogy with the examples from Section 2.1, where a service was calling out to other services in order to accomplish its tasks. Figure 2.4 illustrates the concept. In this thesis, the term *Multi-Domain IaaS* is used to describe this kind of setup.

Nevertheless, for the variable-role assumption to hold, there are certain guarantees that should apply. We already mentioned some about security. There can also be others about performance, legal concerns, etc. Hiding the information about who actually delivers the service, as is the case here, means that the primary provider ("A" in the trivial example here) is legally bound to any promises made to the user, even if it is provider "B" or "C" who fails to deliver. This is a more general concept in service-orientation. Such concepts, related to guarantees, obligation, default and penalty, are covered by *Service Level Agreements* as will be

discussed in the next chapter.

Summary and Conclusions

Service computing is the discipline that studies how different services can be exposed, discovered and bound (possibly into larger service processes), so that reusability and expertise are taken full advantage of. Cloud computing is the service computing area where virtualization is utilized to hide from the user the less relevant information, such as equipment and data location. Given sufficiently good reasons (e.g. capacity limitations), a cloud provider or operational unit may choose to outsource part of the service that a customer requires. This concept is essential to the present work, specifically in the context of infrastructure services.

The upcoming chapter discusses Service Level Agreements more formally, and elaborates over relevant concepts of the SLA life cycle.

Chapter 3

Service Level Agreements

The present Chapter explains important concepts relevant to the rest of the document. More specifically, it discusses in detail what Service Level Agreements are, how they are related to service computing and what SLA management includes from a life cycle point of view. In addition, there is provided some discussion regarding a number of problems pertinent to automated negotiation of SLAs.

3.1 Definitions and General Considerations

A *Service Level Agreement* (SLA) is a representation of all those features that a user can expect to receive by a service, plus related information to provide the context unambiguously (such as user's obligations). "Features" here refer not only to the functionality delivered by the service, but also to the quality that the user experiences. As a matter of fact, SLAs are typically associated with *Quality of Service* (QoS), but in a formal representation it is reasonably expected to find the service description as well.

In general, Service Level Agreements define context, obligations and rights for the parties involved, as regards the service consumed. They consist of a set of *facts*, and a set of *rules*. Facts are globally (with respect to the contract) applicable truths, such as parties involved, monetary unit, etc. Rules include:

1. the *conditions* that must hold for a certain *clause* to be in effect;
2. the clause itself, typically describing the expected result that the customer wishes to receive – and which is usually referred to as *Service Level Objective* (SLO); and
3. a fall-back clause in the case that the aforementioned clause is not honored.

As an example, for the condition “time of day is after 08:00”, the clause could be “response time is less than 5 seconds”, and the fall-back clause could be an applicable penalty. This kind of format actually reflects real-life contracts and their *if-then-else* structure, which might apply either as the default or as the exception to such default respectively.

SLAs are typically said to govern service consumption, and guarantee its properties. It was already mentioned that SLAs try to augment services with acceptable levels of determinism. What this means is that, purely on a document level, SLAs also include measures of business value and penalty constructs for those cases that the agreed QoS level cannot be maintained by the service provider.

Given the definition above, SLAs may be represented in various ways; in this work, only their electronic representation is considered, for purposes of *automated* management by means of machine-readable documents. In a direct analogy with contracts as known from traditional business, SLAs are commonly referred to as *electronic contracts*. As a consequence of this analogy, the following important questions come up:

- What are the legal implications of electronic contracts, when it is software that decides whether to establish (sign) them, or not?
- What is the language in which an electronic contract is written?
- How is negotiation of electronic contracts taking place?
- How can the terms of the contract be expressed in uniform ways? That is, how can the semantics of the language be commonly understood?
- How can the software actually make the decisions?

With regard to legal implications, it is clear that machines and software cannot be held accountable in the case of penalties, or more severe breach of contract. Therefore, it is commonly accepted that a formal (non-electronic) contract negotiated between humans needs to be in place, governing all automated interactions between software entities. This contract will have to define what are the purposes of the automated SLAs, the capabilities and administrative boundaries of the negotiating software entities, acceptable terms and penalties, and in general all those business and technical characteristics that apply to the automated interactions. Hence, a framework contract must be in place to reflect accountability and responsibilities of involved business parties.

The next three questions are more technical and have to do with proper representation of messages exchanged between the two parties

(the software entities), as part of the establishment process. In general, it is desirable to model this process after the establishment process of paper contracts, where a first draft is created and then the parties involved perform revisions iteratively. This iteration of drafts forms a *multi-round negotiation*, until both parties are satisfied and agree to sign. A similar approach is commonly used to model negotiation of electronic contracts. However, the question of the contract's content remains. In real life and paper contracts, someone expects that the contract is written in a physical language that both parties can read, e.g. German, English, French, etc. Achieving this universally in software is not trivial, and it is only recently that the first relevant standardisation effort became successful with WS-Agreement [14]. Even with such a common language, though, the problem of describing the obligations and expectations of the involved parties is still open. Let us consider an artificial example, where a contract is negotiated between an administrative manager and a systems administrator. The contract is about backoffice functions invoked by the adminstrative person's desktop spreadsheet. The manager would require that whenever she invokes a specific function from within the spreadsheet, there is *always* a result returned within a few seconds. The administrator may not be able to interpret this to technical terms, for which she would be willing to sign a contract. Such terms would be, for instance, the failover hardware of the backoffice server, load-balancing for performance optimisation, etc. These two people cannot sign a contract, although they may speak the same natural language, because they are mutually unaware of the meaning of the guarantee terms their cosignatory party is willing to accept. The exact same problem applies to electronic contracts, where software entities may support and understand different sets of SLA terms.

The final question has to do with automated e-contracting, which is the latest trend in the area. Service Level Agreements are being used already extensively in various parts of the IT industry, but their first applications were in computer networking by means of facilities such as *Differentiated Services* [15] (DiffServ). SLAs were initially contracts that defined things such as allocated bandwidth, quality of networking circuits, etc. SLA research in the networking area is still very active (e.g. [16, 17, 18]), nevertheless, with the advent of service computing and the "everything as a service" principles, the concept was applied to plenty of different areas and *automated SLAs* [19] are considered in more recent research. Automation here is related to the absence of human intervention to decide what is acceptable and what not, and perform the negotiation and runtime management. Conversely, there are assumed autonomous

agents that can perform all decision-making during negotiation and run-time, given predefined policies and business logic to drive decisions.

3.2 SLA Management

SLA Management entails the actions that must be taken throughout the life cycle of a Service Level Agreement. Figure 3.1 [20] illustrates the definition of the *TeleManagement Forum* (TMF) for this life cycle [21].

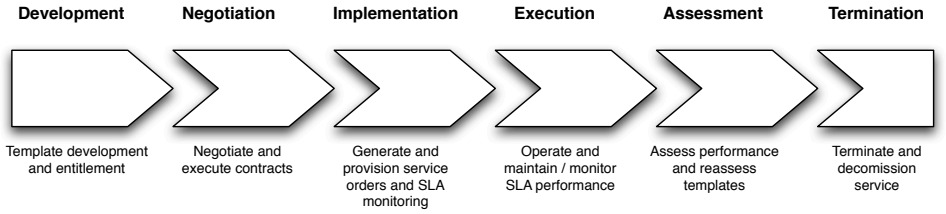

Figure 3.1: The SLA life cycle, according to the TMF

Initially, a *Service Level Agreement Template* must be created. This template abides to some description model, such as WS-Agreement or WSLA [22]. Figure 3.2 illustrates the structure of an agreement, and that of an agreement template, according to WS-Agreement.

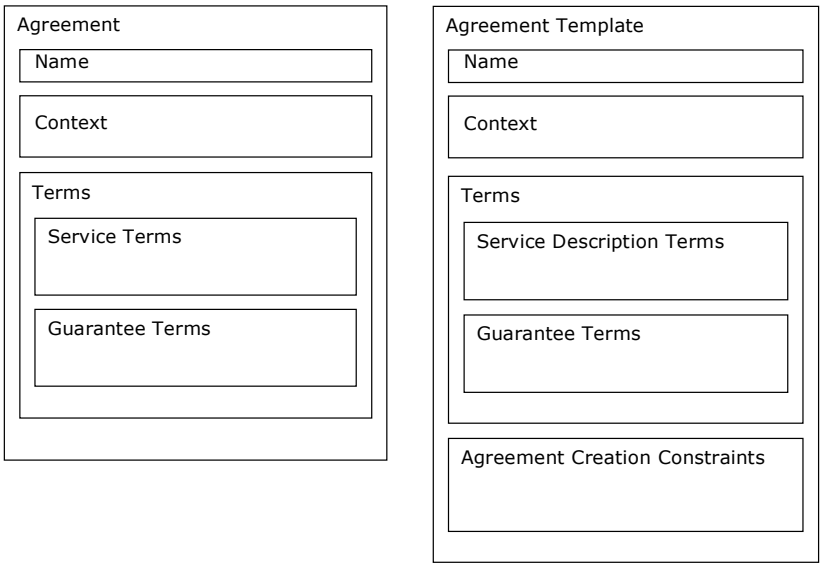

Figure 3.2: An SLA and an SLA template, according to WS-Agreement

A template's purpose is to provide guidelines for the upcoming SLA negotation, by providing a description of the service, a list of negotiable

properties and the applicable constraints, placeholders for stating business values and penalties, and possibly information such as trusted 3rd parties etc. The SLA templates need to be used by the *initiator* of a negotiation, to form an initial *Request for Quote* (RfQ), an agreement offer, or other possible messages.

The *SLA negotiation* follows, where the involved parties are expected to hold a series of message exchanges until they either converge to a final agreement, or fail and quit. For the negotiation to take place, a protocol is needed (e.g. Contract Net [23]) so that involved agents can interact in a structured and meaningful way. Additionally to that, the negotiation parties typically have a strategy based on which they decide what is a good and what is a bad SLA, from a *utility* point of view. Different agents with different strategies may have difficulty to converge to a commonly acceptable SLA; In fact, it has been proven that this negotiation may take prohibitively long to converge [24]. This is why, in most cases, it is considered useful to provide hints to each other about preferences with regard to the candidate achieved objectives. For instance, if the negotiation involves an agreement on the cost of the service, and its availability, the customer may inform the provider that lower cost is more important than high availability. Thus, the provider would be in a position to provide counter-offers that have more chances to be acceptable by the user.

A negotiation may trigger another negotiation; for instance, the negotiation of a customer with the geocoding service, may trigger a negotiation between the latter and an infrastructure service. At this point in time, an *SLA Translation* process takes place: it converts the terms of the first negotiation into terms of the second one. This way, it is possible for the geocoding service to have reasonable certainty that it will be able to offer itself under the negotiated terms and conditions, and therefore it can agree and establish the SLAs with its customers. SLA Translation, and the resulting SLA hierarchies, are discussed extensively in Chapter 5.

The *Implementation* of the service, and therefore of the respective SLA, follows negotiation. In this phase the provider must configure the resources allocated during the negotiation phase, so that they are ready to be used during service execution. For example, this would involve installation of software on the allocated hardware, network configuration, etc. Monitoring setup is also included as part of this phase. The provider may prepare the monitoring infrastructure, so that it is readily available during service execution. For example, this would involve installation of software on the allocated hardware, network configuration, etc.

The *SLA Execution* entails the execution of the respective service(s), and their monitoring to confirm compliance to the SLA. Following the col-

lection of performance or other information, the parties involved will evaluate the conformance of the service execution to the SLA. If some violation occurs, or is about to occur, countermeasures may be taken to bring service performance back on track, or to reduce the violation probability. Hence, this *adjustment* process is a mostly dynamic part of the SLA life cycle. In the case of a possible failure, penalties should be assessed. A provider may choose to pay penalties than react to a violation, depending on business policies. Another possibility is the re-negotiation of the SLA, perhaps in parallel with penalty procedures. In all possible tracks of action, SLA dependencies and hierarchies should be taken into account. This contributes to system sanity and efficiency, with the reasons for failure being discovered faster and with greater certainty.

The *SLA Assessment* phase is mostly related to the Execution phase, as it reuses the information collected during that one. More specifically, monitoring data (which shows how the service performed in relation to the guarantees provided as part of the SLA) is assessed in order to evaluate the sensibility of the offered templates. The outcome of the process may be, for instance, that the current templates are not properly calibrated to take into account the QoS level that a service can actually offer. As such, the templates must be adjusted to lower (or higher) limits and constraints. This feedback process is important for the longevity of SLA-based operations of a provider, otherwise there is a risk that penalties will be too high to sustain.

The final phase is the *SLA Termination*, during which the allocated resources are decommissioned, and clean-up takes place. Accounting and billing may also happen at this stage, especially for short-lived services where there is no need to have periodical payments.

In Chapter 6, a management architecture will be presented, which is largely affected by these definitions for the SLA life cycle.

3.3 Related Work

In upcoming Chapters of this thesis, multiple citations and references to related work are provided to illustrate better the relationship between the presented work, and prior art. Nevertheless, there is important work that has been committed in the past in the area of automated SLA management, and is not directly within the scope of this thesis. In this Section, such work is discussed to provide a more complete picture of the area for the interested reader. Certainly the citation list herein cannot be exhaustive, rather there is an effort to provide some characteristic results and give a feeling of the developments timeline in this scientific area.

In [25], Jin et al. make an analysis of SLAs for web services, around the time when such concepts also started being formalized with WSLA. They look at the topic in relation to then-developing technologies, such as WSDL and UDDI [26], and consider various aspects already such as service composition and how it affects SLAs.

In 2002, Sahai et al. presented a paper on automated SLA monitoring for web services [27]. Without a formal representation for SLAs being available, they try to build on process description languages and introduce quality metrics to be monitored. Then, an architecture and necessary mechanisms for a distributed monitoring system are discussed.

In 2003, one of the first articles on SLAs for Grid Computing appears, by Leff et al [28]. Within the context of commercial grids, the authors look at the issue of honoring SLAs and meeting demand via techniques mostly relevant to workload migration. Considerations for the definition, monitoring and enforcing of the SLAs are integral to the article.

Shortly after WS-Agreement first drafts appeared, people already eagerly started using it to implement various services that could offer performance guarantees. In 2004, Zhang et al. used it also in the context of Grid Computing for a data transfer service [29]. Using the WS-Agreement model, the service would define agreements and offer guarantees for the time completion of a data transfer. Decision-making for the feasibility of the SLA was performed using prediction mechanisms; then rate-limiting on the network interface was used to achieve the enforcement of the SLA.

In [30] (2005), Aiello et al. take a critical look to WS-Agreement, and reflect on the semantics of SLAs. Following, the authors propose extensions to the specification, mostly with regard to SLA life cycle and acceptable states, so that agreements become more long-lived and robust as regards forthcoming violations. In the same year, Sauve et al. connect SLA planning to business objectives such as infrastructure costs and financial loss [31]. The authors develop a complete impact model, to reflect SLA failures onto business objectives.

Chawathe is also considering SLA planning in [32], from a theoretical perspective related to the strategy that can/should be followed by negotiating agents. Profit and risk distribution, as well as collaboration feasibility, are discussed. Also in 2006, Oldham et al. extend WS-Agreement with *Semantic Web* technologies, to enhance partner selection during automated service compositions.

In 2007, McKee et al. [33] discuss different strategies for decision-making while negotiating in the context of grid service marketplaces. The authors elaborate on the complexity of the problem, and look at possible consequences such as “self-denial-of-service” in the case of reserving re-

sources as a means of promise during negotiation.

Barbagallo and Comuzzi explored cross-disciplinary aspects of automated SLA negotiation in 2008 [34]. Specifically, they focus on mechanisms to reduce adverse selection and moral hazard risks. They present a framework for the integration of microeconomic and engineering issues, in contract definition and management. Proposed techniques include third-party verification, signaling, screening / monitoring, reputation management, incentives, etc.

Finally, recent and interesting research results include [35] by Wada et al. The authors both look into SLA-aware service composition, focusing on the multi-objective nature of the problem. A genetic algorithm is being used, alongside an optimization framework called E^3. The application of multi-objective optimization on SLA planning is a promising approach for SLA planning, and indeed, something that this thesis also considers within the context of SLA Translation in Chapter 5.

Summary and Conclusions

In this chapter SLAs were discussed in general, initially providing some considerations which apply to them due to their (possibly) legal nature. Then, technical and other questions about SLAs were put forward, to indicate some of the existing, real-world problems that apply to their automated management. Apropos, Section 3.2 elaborated over all the things that their management entails, according to the SLA life cycle definition from the TeleManagement Forum. The final Section offered a timeline of research in the area of automated SLA management, albeit with a reduced but characteristic set of scholarly publications.

The following Chapter is the last one of Part I. Having provided a concrete background for the thesis' work, Chapter 4 defines the respective scope in detail further than within Chapter 1, by means of realistic use cases. The gaps are identified, and scientific contributions are underlined.

Chapter 4

Use Cases and Problem Description

In this chapter two relevant use case scenarios are introduced, which illustrate typical relations between a service stack and its corresponding SLA hierarchy[1]. These use cases are found today in real-life situations, and are inspired to a large degree by the SLA@SOI EU project. They involve software and infrastructure services, which are expected to be provisioned with performance or other guarantees. A service hierarchy is present, implying similar SLA hierarchies. Following the description of the use cases, the existing problems are identified in greater detail than the summary of Chapter 1.

4.1 Hosted Enterprise Resource Planning

This use case concerns a Software-as-a-Service scenario, where an *Enterprise Resource Planning* (ERP) system is made available to some external customer. That is, the customer does not need to install the ERP stack locally, rather she can use it via some thin client, or a web browser. Without loss of generality, this setup is mostly targeting smaller companies, who usually prefer to avoid the hassle and administrative costs of a local installation.

As defined in [36] by Bidgoli:

> *Enterprise Resource Planning (ERP) is an integrated computer-based system used to manage internal and external resources, including tangible assets, financial resources, materials, and*

[1] The concept of SLA hierarchies is formally defined in Chapter 5; here we will follow an informal, via-example approach.

> *human resources. Its purpose is to facilitate the flow of information between all business functions inside the boundaries of the organization and manage the connections to outside stakeholders. Built on a centralized database and normally utilizing a common computing platform, ERP systems consolidate all business operations into a uniform and enterprise-wide system environment.*

Although different ERP implementations will typically entail different subsystems depending on the customer's requirements and the business offering, the following components are considered to be common across the various implementations [37]:

- Transactional Backbone
  - Financials
  - Distribution
  - Human Resources
  - Product life cycle management
- Advanced Applications
  - Customer Relationship Management (CRM)
  - Supply chain management software
    - Purchasing
    - Manufacturing
    - Distribution
  - Warehouse Management System
- Management Portal/Dashboard
  - Decision Support System

In the present use case, such involved subsystems are exposed as services within the ERP provider; and one may invoke the other to complete its various tasks. Other, lower-level components that are never directly used by the customer, are also exposed as services and invoked by these higher-level subsystems.

Considering that this is a business-critical system, and that complete companies usually base their operations on it, it needs to be always available, and perform within acceptable limits. What constitutes "acceptable limits" depends on the customer and the domain, and is the subject of

SLAs between the customer and the provider. For instance, let assume a retail store that uses such a hosted ERP system to manage its inventory. When a customer wishes to buy an article, a store clerk needs to check the availability of this article. The inventory module will typically first look at the warehouse availability; and if the product is not immediately available, it would consult with the purchasing and distribution components, to see if there is an ongoing order, and if so, what is its current state.

In this scenario, the retail store (as a customer of the ERP service) will expect that the performance of these queries is sufficiently good, so that the customer can be informed promptly on the article's availability. Such performance guarantees are the subject of the SLA among the customer and the ERP service provider. For this running example, one guarantee may be:

> *For all inventory queries within 24 hours, 99% should complete within 5 seconds; and the remaining 1% should complete in no more than 10 seconds.*

Another one, may be:

> *Under the precondition that any query which does not return a result within 10 seconds is considered as a failed request to the system; and with "availability" being defined as the ratio of completed requests to failed requests; then availability of the system over 24 hours of measurements should be no less than 99.9%*.

Such requirements are very common in real life, and SLAs need to reflect them. In the case of automatically negotiated SLAs among software agents, the software needs to be able to deduce what any of such guarantees between the customer and the provider means for corresponding SLAs *within* the provider, and among its services. Relevant questions may involve, for this example:

1. What are the acceptable time limits for queries to the warehouse availability component, the purchase component, and the distribution component, so that the total time for the three sequential queries does not exceed 5 seconds (or 10 seconds for up to 1% of all calls)?

2. What is the technical setup for redundancy and failover, so that availability remains higher than 99.9% within a span of 24 hours?

In addition, it is assumed that the infrastructure may be available internally, or rented from an external (infrastructure) provider. Either way,

the SLAs for the software services will have to be reflected on SLAs for the infrastructure service(s).

4.2 Enterprise IT

This use case refers to a provider of infrastructure resources. It is a department of a larger organization, offering its services within that organization, and may use its own resources exclusively or may choose to outsource some of the workload in the case of utilization spikes. The customers are the departments executing management functions (e.g. the company's Chief Technical Officer).

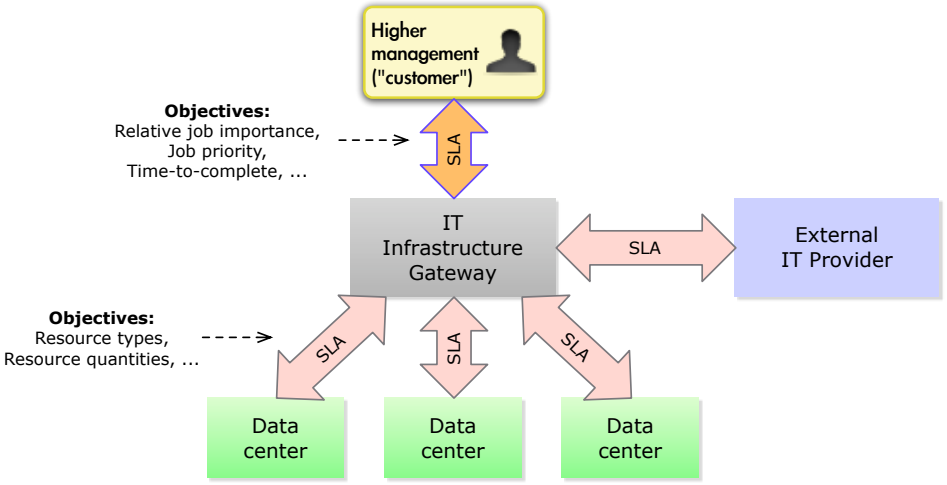

Figure 4.1: Enterprise IT

Figure 4.1 provides a high-level overview of the basic concept. Here, higher management dictates the objectives according to which resource allocation and job prioritization takes place. Such objectives may include things like the expected completion time of a job, the performance requirements of a service, the importance of a task in relation to other similar tasks, etc. As soon as a new SLA is requested to be established, the system must infer resource requirements from such higher-level objectives. Following that, SLAs with data centers (resource pools), and even with external providers if possible / necessary, are requested to be established. These second-level SLAs are produced based on the first-level SLA, as it has been requested by the management department.

In order to stay up-to-date with business environments that may be changing rapidly, objectives and priorities for tasks served by the IT infrastructure may also change often. Examples of such situations include

the sudden, unexpected customer request for a software service, the adjustment of a company's operations to some natural disaster, a rapidly changing financial environment that reflects on the importance of financial models and simulations, etc. It is therefore desirable that the system can handle SLA adjustment requirements. Let assume, for example, the case of two large-scale simulations executed in parallel by two different research groups within the same company. They are both governed by SLAs, which map time-to-complete on resource quantities assigned to each workload. Should one simulation be suddenly considered more important than the other as regards its completion time, it may apply that resources have to be re-assigned accordingly. In such cases, it is desirable that the infrastructure adjusts automatically and takes into account all related consequences for the complete set of SLAs. One possible course of action would be that the *IT Infrastructure Gateway* from Figure 4.1 renegotiates its SLAs with the data centers: It could free up some resources from the less important job, and re-assign them to the more important one. Another option is to request that higher-capacity and performance resources are now used for the more important job, therefore triggering a migration process within the data centers.

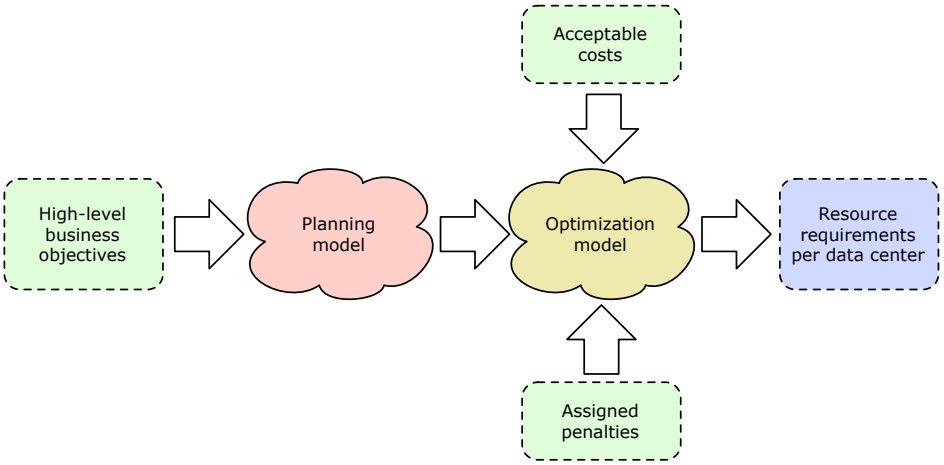

Figure 4.2: Enterprise IT planning and optimization

Independent of the selected course of action, there is apparently required the intelligence to translate objectives from one SLA to the others, and choose one of the various possibilities in accordance to expected utility. Models (typically, domain-specific ones) are therefore required to perform this translation via planning and optimization processes, as illustrated in Figure 4.2.

In this broader SLA planning / optimization and adjustment context,

things like *Total Cost of Ownership* (TCO), data center efficiency, high availability, etc, become very important decision factors. Additionally, resource allocation algorithms need to take into account resources available externally, and deduce whether it would be beneficial to outsource some of the workload. Pricing then comes into the picture and has to be weighed against higher management requirements. Based on such factors, the SLA management framework can deduce whether subcontracting is feasible, and what may be a good or optimal contract.

4.3 Problem Description

The gaps and open issues related to these two use cases (but also to the example scenario from Section 2.2) are quite a few, spanning plenty ICT disciplines (e.g. combinatorial optimization, performance engineering, forecasting, security, capacity engineering, virtualization, etc). In this Section, the ones most relevant to this thesis will be identified.

First and foremost, there needs to be a *formal definition of a hierarchy of SLAs* (Gap 1). Although the concept can be intuitively understood within the Service Computing area, a formal way to describe what it means is still lacking. More specifically, it is necessary to understand *how can SLAs be related*, what is the *nature of dependencies among different SLAs*, and *how those dependencies affect other SLAs established by the same entity*. As an example from the Enterprise IT use case, when the priority of one SLA is increased, and no outsourcing is possible, the priority of other SLAs must be decreased. Even if it is possible to outsource some of the workload, someone must decide which exact workload will be outsourced, and how does that affect costs and performance. Having defined the concept of SLA hierarchies, it is necessary to understand what it means to deduce one from another; that is, *how and when is a hierarchy of SLAs built*.

Following this step, it is needed to have an *SLA management architecture* that takes into account such requirements for the *establishment of SLA hierarchies* (Gap 2). This architecture should be *extensible*, to be applied to any possible use case, *independent of languages, protocols, and legacy systems* already in place. In addition, it would ideally be *autonomous*, so that it is easier to reach (at least) some local optimum when evaluating the feasibility of a contract, or trying to infer a subcontract; both tasks can be extremely expensive from a computational point of view. An architecture of this kind is a necessary milestone in order to achieve hierarchical SLA management, as discussed in the example scenario and the use cases.

The tasks of evaluating SLA feasibility and deducing subcontracts are tightly coupled with the concept of *utility*; that is, some measure of benefit that comes from the respective decisions. More often than not, utility is associated with financial results and some measure of net income. Pricing may take many different forms, and there is already significant prior art in this area. Nevertheless, *a complete model for the definition of penalties* depending on the agreed price, is still missing (Gap 3). Various efforts have taken place in recent years –these are discussed in the respective chapter–, but the author believes there is room for improvement. This is, therefore, an additional identified gap, and a proposed solution is part of this thesis. The solution seeks to be *fair* (thus associated to price, and some objective measure of failure) and unambiguous.

Penalties aside, computing the feasibility of an SLA under its specific terms, or inferring good/optimal subcontracts starting from a top-level contract, may be extremely demanding from a computational point of view. In certain cases (depending on the algorithm at hand and the number of candidates) it may even be impossible to deduce. To the combinatorial nature of resource planning for multiple SLAs, adds the fact that SLAs themselves may be arbitrarily complex. Multiple logical operators such as AND, OR, XOR may be found – for instance, this type of constructs are integral to WS-Agreement. Even within the two simple terms from the ERP use case, multiple terms can be found that are combined with such logical operators. Further to that, "if" clauses are found in the form of pre-conditions for an SLA term to apply, introducing further complexity to the handling of SLAs. Putting everything together, the task of processing incoming (or producing outgoing) SLA offers becomes very hard. Generic recipes to handle this kind of complexity are much needed, and constitute an important gap in the area of SLA negotiation and management in general (Gap 4). This thesis introduces and proposes a methodology using a graph-based structure, known as *Binary Decision Diagrams* (BDDs). Using this methodology, it is possible to simplify SLAs significantly under the assumption that they can be represented as boolean functions. This simplification leads to a *canonical form*, therefore their handling becomes much easier.

The four gaps and respective innovations discussed so far, are generic and applicable to SLA management independent of the domain at hand. Due to the recent explosive growth of IaaS platforms and applications, as well as the recognition by many experts of the need to enhance IaaS with full SLA management, the author decided to focus further on this area, applying the know-how achieved from the generic work outlined within Part II. Within this context, a necessary milestone was to have

an *abstract and flexible syntax to describe resources and operations for reserving them*. Advance resource reservation is a very useful facility during SLA negotiations. Although there is significant amount of prior art in the area of resource description, a sufficient combination of the two features could not be found; thus, a model was developed to describe arbitrary infrastructure resources and primitives for their reservation. The result was loosely based on *existing standards* such as CIM and GLUE (further discussed in Chapter 9), to enable practical use while also applying the respective experience.

This model is a tool to use for some practical problems. The interest of the author was to research the practical use of SLAs in two different scenarios: First, examine the scheduling of IaaS SLAs in the normal case where customers arrive randomly and request some resources for a specific amount of time. Taking into account these incoming requests, the main question to answer was: How to model the scheduling problem, so that the energy footprint of a provider's data centers is minimized? The online nature of the scenario (i.e. for each incoming resource/SLA request, the next ones are unknown) means that some simpler, greedy approach is both convenient and suitable. Indeed, the problem was modeled after the *online bin-packing* problem, where each bin is represented by one server, potentially shared by many customers.

The second scenario is modeling an extreme situation, where the resources are not enough to satisfy a number of already-established SLAs. One such example is the case where a considerable percentage of available resources becomes unavailable, e.g. due to power outage. The question that arises, is how to best manage this case and adjust the use of available resources, so that the least penalties are paid to customers. Various possibilities are examined as regards which parts of the SLAs are honored, and which ones are purposefully withdrawn. A combinatorial model is developed, as a *knapsack problem* variant. To the best of the author's knowledge, this is the first time that this variant appears in literature. The model is evaluated in three different variations, as well as against two greedy algorithms.

Summary and Conclusions

In this Section there were provided two typical, real-world use cases for SLA management based on industrial use cases of the SLA@SOI project. The relevance to SLA hierarchies was discussed and respective gaps were identified. These gaps were analyzed to result in a more complete problem description, so as to provide the full scope of the thesis.

The upcoming Part II includes scientific contributions that apply to SLA management independent of the application domain. Initially, SLA hierarchies will be discussed; then a management architecture; a penalty model; and finally, a flexible, generic model for in-memory SLA representation.

Part II

Hierarchical SLA Management

Chapter 5

SLA Hierarchies

Starting from service chains and business processes, one can deduce dependencies of one service on some other(s). These dependencies stand behind the concept of SLA hierarchies, which is central to this thesis.

The present chapter discusses the respective topic, exploring it in detail. A generic definition of the "SLA translation" problem will be provided, alongside related formalisms that connect it to SLA negotiation.

5.1 SLA Dependencies

Let assume a service hierarchy, where a service "A" relies on two other services "B" and "C". This practically means that, if a customer requests to consume A under some specific guarantees, the service provider can only agree if it has reasonable certainty that services B and C will perform accordingly. Otherwise, best-effort service delivery by the latter can easily affect the delivery of the former. That is, the guarantees offered (and agreed upon) between the provider and the consumer of service A, depend for their compliance or violation on the guarantees agreed upon between A and B, and those between A and C. Hence, it is necessary to have *SLA Hierarchies* that reflect the service hierarchies to which they apply.

For instance, let assume that service A is the address-based route planning service from Section 2.2 (AB), and services B and C are the respective geo-coding (GC) and route-planning (RP) services. It has already been shown that the completion time of an invocation of the former depends on the completion times of invocations of the latter two. Therefore, these non-functional properties are related. Should an SLA be established to guarantee the properties of the dependent service, then an SLA should also be established to guarantee the related properties of the

antecedents. Figure 5.1 illustrates the concept; and Section 5.1.1 elaborates further on it.

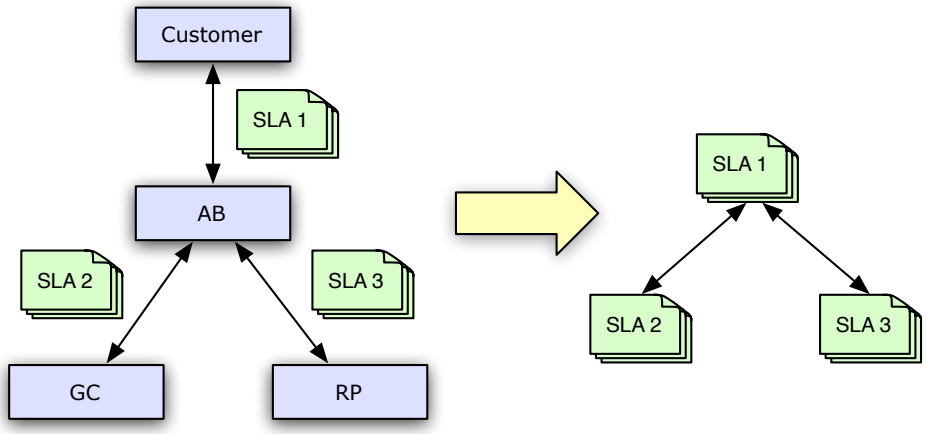

Figure 5.1: SLA dependencies, reflecting service dependencies

The existence of these hierarchies signifies the need to have some method to *deduce one from the others, based on their inter-dependencies*. In the text that follows, this process will be referred to as *SLA Translation*. One could say that SLA Translation is the process that *designs* a hierarchy of SLAs, during negotiation between the various parties involved; and the establishment of the SLAs is what builds and finalizes the hierarchy. Just like with service hierarchies, full distribution of SLA hierarchies is assumed. Each party knows only about the parties (services) depending on it, and the parties (services) on which it depends. Similarly, each party only knows SLAs for which it acts as a provider, SLAs for which it acts as a customer, and the links/dependencies between the two categories. There is no assumption about a centralized data store that holds all SLAs and all dependencies, thereby leading to the possibility to deduce the complete hierarchy/-ies within a service ecosystem.

5.1.1 Service Properties Dependencies

An SLA was described earlier as a set of guarantees over the consumption of a service. Dependency of a service upon others, naturally means that its properties are depending on properties of these other services. Assuming the availability example for a dependent service, it will be affected by the availability of its antecedents: Service AB cannot execute when either GC or RP cannot execute – hence, the availability of the two latter affect that of the former. The same applies for other service prop-

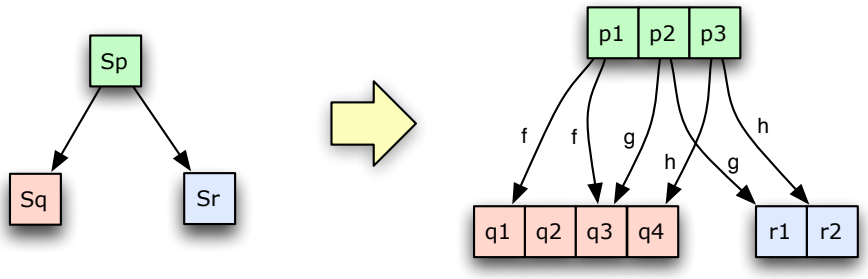

Figure 5.2: Example service Properties Dependency Graph

erties such as invocation completion time, maximum throughput, *Mean-time-to-Failure* (MTTF), *Mean-time-to-Repair* (MTTR), etc.

There may also be cases where a dependent service's property may be related to a completely different property of its antecedents. For instance, the cost of a service is typically affected by the quality of the services that it uses for completing its tasks. Additionally, it is certainly possible to have an SLA which does not refer to all properties of a service, and therefore only a limited number of dependencies must be taken into account: only those which affect the properties mentioned in the SLA.

It is becoming clear that there cannot be a universal classification of service properties, and their dependencies. Certainly, facts that set the context of an SLA cannot apply to all SLAs either, but rather, distinct facts for distinct SLAs should be assumed. It is only possible to define the problem generically based on the abstraction of conditions that need to hold true, when aggregated according to the SLA structure. Use-case-specific knowledge needs to be applied by domain experts to instantiate these conditions, and associate them in the context of different, dependent services.

Starting from the service dependency graph (see Section 2.1), it becomes possible to make a next step towards a service *Properties Dependency Graph* (PDG). If service S_i has properties $P^{S_i} = (p_1^{S_i}, p_2^{S_i}, ..., p_m^{S_i})$; service S_j has properties $P^{S_j} = (p_1^{S_j}, p_2^{S_j}, ..., p_n^{S_j})$; and S_i depends on S_j; then a dependency of S_i's r-th property can be formulated as a function f_r of properties of S_j:

$$p_r^{S_i} = f_r^{S_i}(p_1^{S_j}, p_2^{S_j}, ..., p_q^{S_j}) \tag{5.1}$$

where $1 \leqslant r \leqslant m$, and $1 \leqslant q \leqslant n$.

This definition can be extended for a service S that depends on more than one services. Let assume that S has properties p_1^S, p_2^S, ..., p_N^S. Let us also call the antecedents S_1, S_2, ..., S_m, with property sets $P^{S_i} =$

$(p_1^{S_i}, p_2^{S_i}, ..., p_{N_i}^{S_i}), 1 \leqslant i \leqslant m$. Then, each property p_r^S of service S can be generically expressed as follows:

$$p_r^S = f_r(p_1^{S_1}, ..., p_{N_1}^{S_1}, p_1^{S_2}, ..., p_{N_2}^{S_2}, ..., p_1^{S_m}, ..., p_{N_m}^{S_m}) \tag{5.2}$$

Figure 5.2 illustrates such an example. Here, service *Sp* depends on services *Sq* and *Sr*. Domain experts can then create rules about the dependencies of *Sp*'s properties *p1-p3* on those of *Sq* (*q1-q4*) and *Sr* (*r1-r2*), based on modeling techniques, simulation, or real-world observation. These rules are represented by functions f_1, f_2 and f_3 (named f, g and h for convenience) that implement a mapping of each property on the properties on which it depends.

5.1.2 SLA Translation

Operating under the abstractions from Section 5.1.1, the concept of *SLA Translation* can be discussed. In essence, SLA translation is the process of analyzing an SLA in relation to the PDG (i.e. finding the service properties common to the SLA and the PDG), applying heuristics and pre-existing knowledge, and coming up with one or more subsequent SLAs for antecedents. These subsequent SLAs provide reasonable (but not necessarily complete) certainty that the top-level SLA will not be violated, unless some of them are violated too.

The aforementioned analysis starts with Equation 5.2. One can first define a complete set of such equations, one for each property that the customer of the dependent service requires in the negotiated SLA. Properties that the customer does not require guarantees for, can be excluded. Then, this system must be solved, taking into account existing constraints as provided by the SLA templates. These limits as regards what can be negotiated help customers to reduce the search space for their offers. The search space may otherwise be so large that the problem becomes practically infeasible from a computational point of view.

Optimization of subsequent SLAs is one more requirement for the translation process. Actually, optimization is part of translation, as it affects the final output. The system mentioned above will typically accept plenty of different solutions, and it is up to optimization to select the best of all those candidates. As many properties are simultaneously dealt with, this becomes a multi-criteria optimization problem [38].

Using the example from Figure 5.2, the following relations apply:

$$p1 = f(q1, q3)$$
$$p2 = g(q3, r1)$$
$$p3 = h(q4, r2)$$

If an incoming SLA request/offer for service *Sp* refers to all three properties, an SLA negotiation mechanism which includes translation functions should first find out the dependencies of these three properties based on the PDG. It is assumed, as also mentioned earlier, that these dependencies are known as domain-specific expertise encoded into the system. For instance, if the property is "*availability*", it will typically depend on the availability of all antecedents; this is fairly straightforward to assume. However, if cost is examined, it is not necessary that the cost of invoking service *Sp* is related specifically to cost properties of services *Sq* and *Sr*. On the contrary, it may be the case that there are no cost properties for these two services, but rather that their providers only apply flat-rate pricing. In this case, the cost of *Sp* may rely on properties such as Quality of Service characteristics of *Sq* and *Sr*.

The next action is to find acceptable value spaces ("domains") for all of $(\mathrm{q}1, \mathrm{q}3, \mathrm{q}4, \mathrm{r}1, \mathrm{r}2)$, leading to a feasible solution. "Acceptable" are values which remain within constraints set inside the templates of the two lower-level services, and on the same time satisfy the requested values in the offer for the higher-level SLA.

The last step during the translation process, would be to come up with a sufficiently good solution to the problem of selecting SLA parameters for antecedents, according to an optimization algorithm. The latter typically takes into account all applicable business objectives. An obvious such objective is to maximize profit, nevertheless others are often taken into account as well; for instance, reputation. Associating reputation with (future) profit may be impossible without large historical data sets and very complete, complex models. The same may apply for other possible business objectives. Thus, there may be no one single objective for the optimization, and no single optimal solution. Rather, a multi-criteria approach should be adopted. Without loss of generality, it is here assumed that there may be N applicable criteria, $\mathrm{N} \geqslant 1$. In this case, one of the solutions on the *Pareto front* should be chosen. The Pareto front is a set of all those solutions that are considered to be optimal in multi-criteria optimization. More formally, they are solutions where none of the included criteria can be improved (accept a better value), without some other criteria in the same solution receiving a worse value.

Figure 5.3 shows how this translation process can be implemented according to the previous discussion. Counter-offers are assumed to be possible, as part of the negotiation protocol being used. An entity negotiating over a set of variables may find that small modifications to the negotiating party's requirements may increase significantly the resulting utility. In this case, it may just as well modify the proposed term slightly, and return

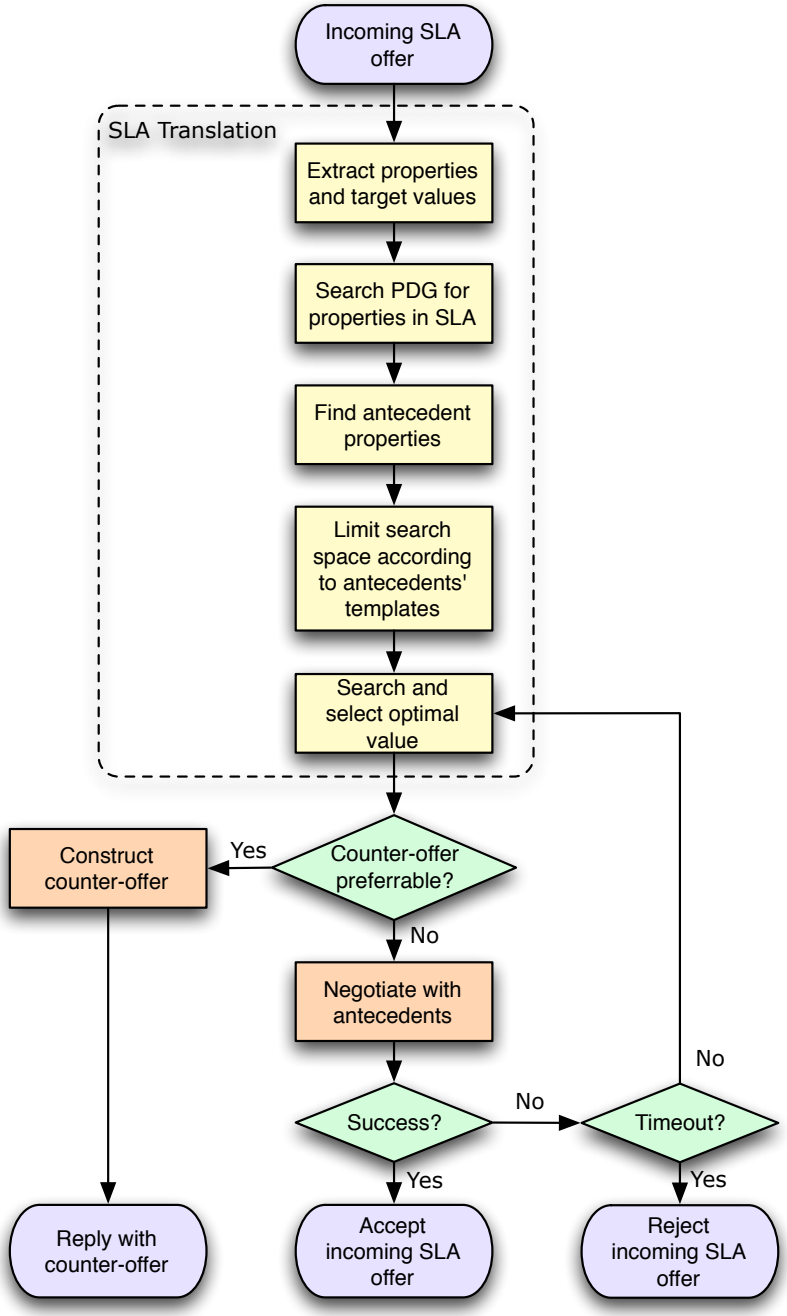

Figure 5.3: Generic flowchart for translation/negotiation of SLAs

a counter-offer which does not match the other entity's requirements, but may provide much better results if accepted. Such negotiation-time risk-taking attitudes can be modeled with *game theory* methods, and indeed there has been important research towards this direction in the past (e.g. [39, 40]) – this topic is, however, outside the scope of the work presented here.

The reason to interleave translation and negotiation in this flowchart, is exactly that they take place in parallel and depend on each other. Negotiation feeds translation with confirmations whether the translation's results are acceptable; and translation provides negotiation with input according to existing domain-specific knowledge, and any further intelligence implemented within the system. It should be clarified, at this point, that the negotiation process and all kinds of decision making remain separate here; that is to say, negotiation concerns only the exchange of messages in such a way that implements correctly a specific protocol, while all planning functionality lives elsewhere and is, in this case, part of the translation process. This is a distinction applied also in the management architecture, as described in Chapter 6. Regarding planning itself, it is anticipated that coping with multiple service level attributes and n-dimensional Pareto fronts might become difficult. In order to be able to implement an efficient and automated SLA translation and negotiation framework, use-case dependent heuristic functions will be required to narrow the search space down to a feasible size. *Scalarization* of the multiple objectives into a single one, may also be needed to achieve automation.

One last thing to mention here, is that the prior discussion tackles the problem of producing lower-level SLAs starting from a higher-level one, in a *top-down* approach. A different scenario is that of a *bottom-up* approach, where the lower-level SLAs are already established and used to decide whether an incoming higher-level offer can be satisfied. The essentials of the previous discussion on producing and using PDGs for SLA translation can be applied just as well on such a bottom-up process, as they are modeling service relationships without being tied to a specific usage scenario.

5.2 Related Work

Although not directly related to SLAs, there is prior art that concerns relationships between different services and/or resources. In [41], Keller et al. describe a *Functional* and a *Structural* dependency model, on which they base their work. They provide a specific classification of dependencies,

into a number of dimensions such as locality, time, type, dependency strength and criticality, etc. Then, building on the two models, they discuss a method for dependency analysis. In [42] the same authors append the two models with an *Operational* one, and further refine a proposed architecture for managing dependencies. The main differences with the present work is that they constrain theirs in the ICT domain where they also provide an architecture to tackle the dependency discovery problem. Additionally, they do not consider SLA management.

In [43], Hasselmeyer discusses the problem in a generic manner, much like this Chapter. However, the work is tied to software services; more specifically, it proposes a scheme where services declare their dependencies on other services based on access patterns (how often are services accessed, in what sequence, etc) with dynamic service selection in mind. Additionally, it looks specifically at functional dependencies. Contrary to that, dependency management in the context of SLAs requires a generic view that handles non-functional dependencies as well.

Bahl et al., in [44], are looking at the same problem as regards network services. The authors describe a model called *Inference Graph* that represents the dependencies. Based on that, they present an algorithm for inferring probabilistically the malfunctioning components of the network, given real-world observations. The main difference with this work is that the Inference Graph describes dependencies based on service states, and probabilities that the state of one depends on the state of another; additionally, the referenced work is explicitly addressing network services.

In [45], Di Nitto et al. introduce an agent-based automated SLA negotiation architecture and discuss efficient search-based algorithms to determine an acceptable Service Level Agreement in a multi-agent environment. The main difference to the present work is the complex and layered multi-tier services landscape that requires additional analysis steps and related data-structures for SLA negotiation. In addition, SLA translation is considered here as an essential part of the SLA negotiation process. Although some of the search-based algorithms analyzed in [45] would be able to cope with multiple objectives that negotiation agents may have, it is not the primary concern of the authors' work.

Summary and Conclusions

This chapter's main contribution is a formalization of *SLA Hierarchies*, and the *SLA Translation* problem. That is, how to deduce an SLA for antecedents when the SLA for dependents is known, and the way that the

latter depend on the former. This computationally difficult exercise can be carried out as an optimization problem, when the properties are different. When the properties are the same (for instance, the number of CPU cores in an IaaS outsourcing request), it is straightforward to go from the first SLA to the second.

The upcoming chapter will present a novel architecture that covers SLA management during its complete life cycle. Alongside components for tasks such as negotiation, provisioning, monitoring, and others, it contains a central component for decision-making (termed “Planning / Optimization / Adjustment”) that is expected to perform the task of SLA translation when required by the specific use case.

Chapter 6

Management Architecture

Managing automated SLAs implies the need for one or more software artifacts that can handle the various stages in the life cycle of an SLA. This Chapter presents a novel architecture for the complete SLA life cycle management, catering for SLA hierarchies. It has been designed with extensibility, domain-independence, and technology-independence in mind. The respective negotiation and runtime scenarios are discussed, followed by technical details about a reference implementation.

6.1 Design

In the past there has been significant effort committed towards implementing different architectures and methodologies for negotiating SLAs, and managing them during their life cycle. Prior art has been concerned with both automated SLA establishment (typically using agent technology), and manual establishment steered by humans. However, existing research results are tied to specific assumptions/use cases, and often specific technologies. In this chapter, a *generic architecture* is proposed, which can be used across different domains and use cases for managing SLAs. This architecture covers all phases of the SLA management life cycle, from negotiation and establishment to termination, also with attention to the existence of SLA inter-dependencies and hierarchies.

The major requirement from this design is that it can be adapted to different use cases, ideally any scenario whatsoever that demands the *establishment* and *management* of SLAs. “Management” refers to provisioning, monitoring, adjusting and terminating SLAs. Therefore, the design has to be adaptable, bear no ties to specific technologies, neither any ties to specific application contexts. In parallel, there must be separation of concerns between service-plane and SLA-plane activities, so that it can

be realistically applied onto existing service infrastructures. Additionally, the design needs to advance the state-of-the-art, by considering hierarchies of inter-dependent SLAs as a result of similar (hierarchical) service compositions.

The *SLA Management Instance* (SAMI) is designed to operate on a level parallel with that of the services, so that is not disruptive to existing service infrastructures. Figure 6.1 shows in an abstract manner the relationship between SAMI and service instances, in the context of the example from Section 2.2. It is reminded that this example involves an *end-customer*, a *geo-location service* offered by a SaaS provider, and the *infrastructure* offered by an IaaS provider. Management of service instances by the respective SAMIs refers to provisioning and adjustment of those instances. The end-customer invokes the geo-location service according to s-SLA, which has been established earlier following negotiation between the end-customer and the geo-location SAMI. The (geo-location) software service executes on the infrastructure provisioned according to i-SLA, established earlier between the geo-location service provider and the infrastructure provider. It should be emphasized, at this point, that this is a simple scenario to be used for illustrative purposes. In a full-fledged use case there may be many more inter-dependent services and SAMIs, in multiple layers.

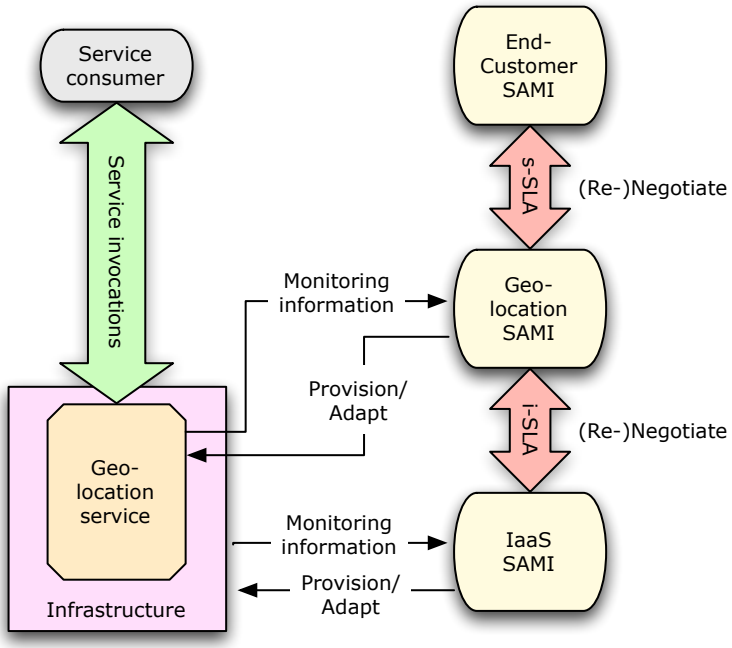

Figure 6.1: SAMI and service instances

It is assumed that the SAMI is autonomous and does not interact with the services in any way other than a) provisioning/adjusting them, and b) receiving monitoring information. Cardinality of services and the SAMI is not predefined either. The architecture is flexible to accommodate different numbers of services. As such, feasible scenarios include a single SAMI for the whole provider's domain, or one SAMI for some services, or even one per service. Figure 6.2 illustrates the SAMI architecture.

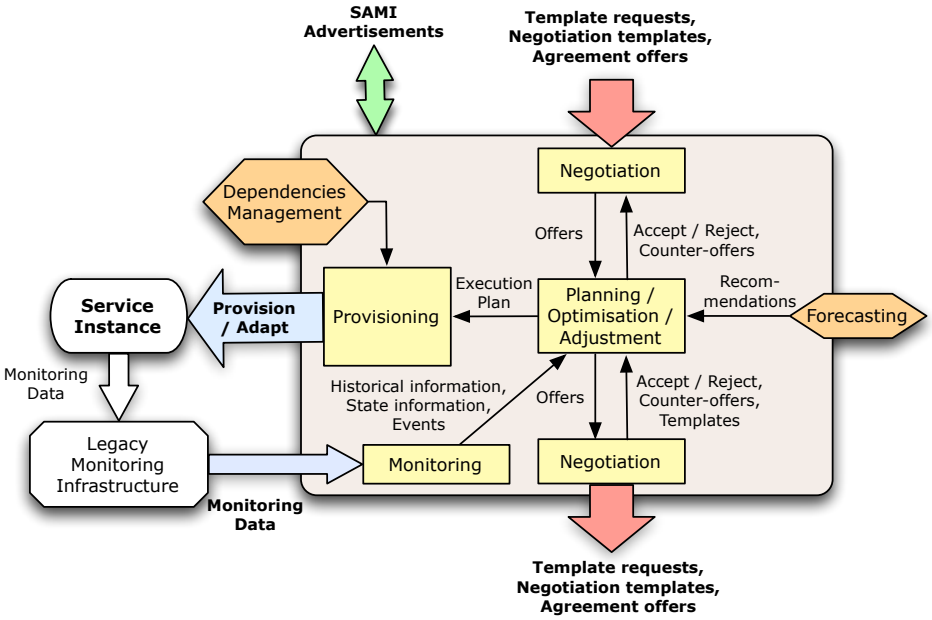

Figure 6.2: The SLA Management Instance (SAMI)

6.1.1 Template Advertisements

Interactions between different SAMI instances may be either based on broadcasting, through the *Advertisements* interface, or point to point, through the *Negotiation* interface. This depends on the type of message exchanged, and the negotiation mechanism. It should be noted that agreement offers may come either by a service customer, or by a service provider.

A *Publish/Subscribe System* (PSS) is being used for broadcasting advertisements to groups of interested parties. The advertisements are SLA templates, published either from customers for purposes of *procurement*, or from providers informing prospective customers about their service offerings. In the case of procurements, a customer broadcasts a template describing the expected service: what it involves from a functional point

of view, what non-functional properties are supported, what can be customized and within what limits, etc. Following, providers interested in signing a contract with such terms can customize the template and return it (either as a template again, or as a final offer) in a point-to-point fashion through the customer's Negotiation interface. In the case of providers that advertise their services, templates are broadcasted as soon as they become available. Then, interested prospective customers can use them to create a modified template for further negotiation, or an agreement offer and submit it through the provider's Negotiation interface. In both cases, the published template should include an endpoint to the publishing SAMI's negotiation interface, so that interested parties can contact it to submit offers.

6.1.2 Negotiation Interface

The Negotiation interface is the endpoint formally used for point-to-point negotiations. It provides a list of available templates for existing service offerings in the specific SAMI's domain of responsibility. This template list may be complete, or customized based on a properly formatted query. Having received and customized a certain template, it is then possible to submit a modified template for further negotiation, or directly an agreement offer. It must be noted, that the second (lower) negotiation interface in Figure 6.2 is there only for purposes of intuitive illustration. In reality, it is sufficient to have a single negotiation interface and implementation. However, this second instance is included in the figure to explicitly indicate the chaining of SAMIs, in the exact same way that services are composed into other services.

6.1.3 Planning and Optimization

It is assumed that the negotiation interface will be able to support different protocols and languages (e.g., WS-Agreement, WSLA), however a single internal model will have to be used for the management of SLAs; a proposal for such a model is discussed later, in Chapter 8. When a template or an offer is received, it is interpreted into this internal model and forwarded to the entity that implements *Planning, Optimization and Adjustment* (POA). This entity represents the central decision system for each specific SAMI, which has the domain-specific knowledge on how to implement SLAs in an optimal or near-optimal manner, and how to enforce them. The decisions of the POA module are related to the existing available resources, and also to the external services that may be needed in order to implement an SLA. Hence, if subcontracts are needed to sat-

isfy the requested functionality and quality objectives, respective offers are created on the fly and forwarded to subcontractors' SAMI instances in consecutive negotiation cycles. A SAMI knows other existing SAMIs through the templates that the latter have published with their advertisements. These templates can therefore also be stored by receiving SAMIs, and used to be aware of existing service offerings. It is this point where the *translation process* discussed in Section 5.1.2 may be required, to convert from the incoming offer to the outgoing offers towards third parties (external to the specific SAMI).

An example of this planning process based on resource availability, and without outsourcing, is the approach discussed in Chapter 10. Chapter 11 provides an example of a planning process that does indeed outsource to other SAMIs, as it has no resources of its own.

6.1.4 Monitoring Infrastructure and Forecasting

Before combining different candidate values for service parameters included in subcontract offers, one has to calculate those candidates first. For this purpose, the POA module depends on two other modules: *Monitoring*, and *Forecasting*. Monitoring will typically be available, as in most cases it is not sensible to establish an SLA without the capability to monitor it[1]. Thus, it is expected that *historical monitoring information* will, in general, be available. Using monitoring data sets, POA is able to decide which external services should be used based on their previous performance and therefore their reputation. Forecasting may be available or not, depending on the use case at hand. In this case, there may be an existing forecasting mechanism used for specific services, to estimate future resource availability, or future performance given predefined conditions. Such mechanisms the SAMI should be able to reuse if they exist, also to make the transition to this new management structure easier.

6.1.5 Provisioning

Following the planning and optimization process, an SLA must be provisioned. This includes preparatory phases that may exist, plus the actual service instance deployment. In this case, the POA module will have to feed an execution plan towards the *Provisioning* component. The latter will also have to take into account information about dependencies of

[1] In a high-risk setup or if dictated by other business policies, it may be the case that a provider establishes an agreement without monitoring it – but rather, based on some rough statistics about the overall system and its performance.

services, for purposes of configuration and resource allocation. The Provisioning subsystem will orchestrate all those actions that are not directly related to Planning and Optimization, but must take place in order to implement a service. That includes, for example, configuration of services according to specific languages and notations, or accounting actions that must come before and after service deployment. Eventually, the Provisioning component will take all necessary actions to enable a service according to the SLA and the consequent execution plan produced by the POA subsystem.

6.1.6 Monitoring and SLA Adjustment

When the service is deployed and enabled, and after consumption starts, monitoring information will have to be collected. As said, there may be an existing (legacy) infrastructure for service monitoring. That will have to be reused, so that service operations are not disrupted and the SAMI platform can be easily applied over existing services. If such a monitoring infrastructure does not already exist, it will have to be created, and eventually monitoring events will need to be sent to the SAMI's *Monitoring* component (or the SAMI should be configured to retrieve such data if needed). First-level processing of raw events takes place there, so as not to flood the POA module with useless information. When a violation of existing SLAs occurs, or is about to occur according to Monitoring's projections, the respective event is sent to POA alongside information that can explain in more detail what went wrong. The POA subsystem may also need to receive current state information about various aspects of the service; that information will also be received from the Monitoring component. Based on current state of affairs and Monitoring's assumptions, the Adjustment part of POA will make a decision on the course of action to take. That may include an adaptation of services through reconfiguration, scaling, etc, or renegotiation. The latter may be directed either towards customers, or towards third parties, depending on whether the (near-)violation is also due to failure of subcontracts.

Business concepts built into POA are used also for negotiation, and their effect is equally profound during renegotiation. Penalties will have to be taken into account during the latter, as well as accounting for the service execution time and consumption up to the moment that renegotiation starts. Signaling of renegotiation takes place by submitting a template, like in negotiation, however it must include a reference to the SLA that is already effective. Whether renegotiation itself is acceptable at all, is something that may be a term in the initial agreement. Even if it is acceptable, the affected party may wish to discard any renegotiation

requests, in which case the violating party will have to pay penalties as defined in the initial agreement.

6.2 Operations

It has already been mentioned that a major design goal for the SAMI was to ensure that operations of existing services would not be affected. This practically means that, ideally, the services will not know anything about the SAMI that manages them, and will not initiate any interaction with it. That said, it is assumed that the processes for interacting with customers, making business decisions and deploying services will have to change as a result of applying the SAMI architecture and its principles within existing service providers.

Looking again at the example from Figure 6.1, in a typical scenario the end-customer would first discover templates describing the geo-location service, indicating which parameters can be negotiated, and providing default values for those. The customer can then form one or more agreement offers based on the templates, or modify some parameters to its likings, and send them back for negotiation. Price could be one such parameter, or left empty as a way to ask for a quote. Figure 6.3 illustrates such an example with a price request, given a throughput of 100 hits/minute and availability higher than 98%. Similarly, the provider could then make small modifications to this document and return it, or could form an agreement offer based on it and send it to the customer. On each round of this negotiation, the SAMIs representing the customer and the SaaS provider would have to evaluate templates, modify them in an effort to optimize the resulting utility, and make their *final* offer (or submit a new template for negotiation). As mentioned previously, it has been proven that this process may take prohibitively long to finish [24], but the practical implications are limited: any business entity can set a timeout, after which it will terminate any negotiation that does not bear results.

From the SaaS provider's side, before it submits an offer or agrees to an offer, it needs to know that it can successfully support it. Therefore, an offer for s-SLA will have to be translated to an offer for i-SLA, according to the domain-specific knowledge implemented within the SaaS SAMI's POA subsystem. Similarly to the case between the end customer and the SaaS SAMIs, there would be a negotiation taking place between the SaaS and the IaaS providers' SAMIs. As soon as i-SLA is established, the offer for s-SLA could be accepted (or submitted). Nevertheless, this design does not exclude riskier strategies, without consecutive (lower-layer) SLAs. Figure 6.4 shows a high-level view of the negotiation process, in

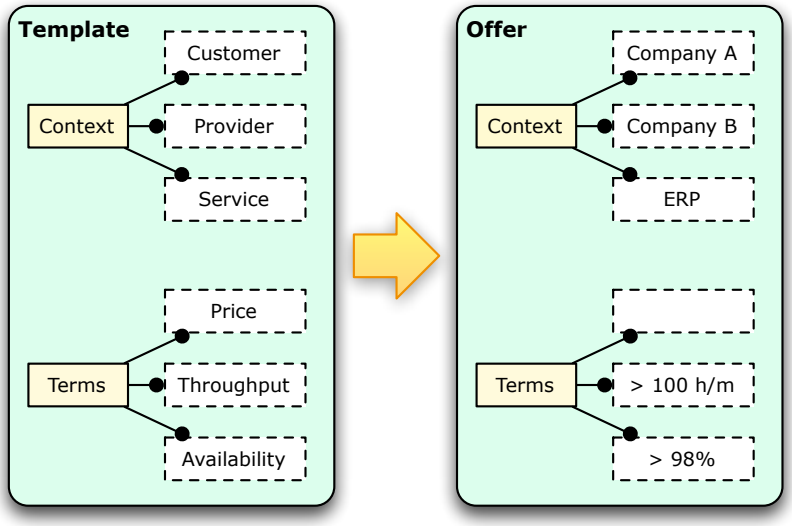

Figure 6.3: A simple Template and Offer example

the form of a UML sequence diagram. A loop of negotiation messages is typically followed by a final, binding offer; and binding offers are ideally synchronized, so that cancellation fees are avoided if a party retracts from negotiations on some layer. This is evident in Figure 6.4, where a failure of step 2.1 leads to failure of step 2.

The author believes that a modeling of the POA system internals that applies to all imaginable use cases is probably impossible. Given the fact that different domains require different vocabularies for the guarantees of the applicable SLAs, and the very distinct utility functions that may be used even by different providers in the same domain/use case, it is only possible to model the interface with generic constructs for assessing SLAs and intervening to the planning/optimization process. Furthermore, it has been proven that no algorithm has a universal advantage over other algorithms, for the set of all possible optimization problems [46, 47]. As such, under the assumption that the Negotiation module looks after correct message exchange, the POA subsystem only needs a method to evaluate incoming offers (either during negotiation or final, binding offers), and a method to interrupt ongoing evaluations.

At the time of SLA enactment, provisioning of i-SLA (i.e. provisioning of the infrastructure resources) must take place before deployment of the software, and the latter should also start before the time that the customer expects to find the service available for invocations. Therefore, this time difference should be taken into account during negotiation and

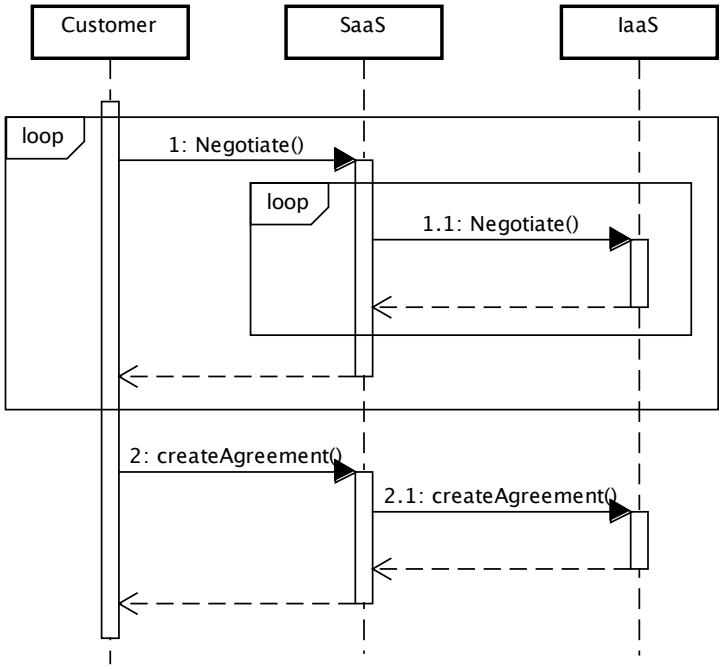

Figure 6.4: High-level overview of the negotiation process

construction of offers. An alternative option is that the customer herself explicitly requests the provisioning of the service when the time comes; and receives confirmation when this happens, after the provisioning of all antecedent services.

Following the provisioning phase according to established SLAs, the IaaS provider's SAMI receives events from the infrastructure monitors, and correlates them to the existing SLAs. If some part of the infrastructure fails, it will try to provision new infrastructure resources as soon as possible. If that is not possible, it will have to start a renegotiation, taking into account any penalties that may apply for doing so. Similarly, the SaaS provider SAMI needs to monitor software service activity. If s-SLA fails, the reason for that should be traced: It could be a customer invoking the service more often than agreed, or it might be software failure, or infrastructure failure (i.e. violation of i-SLA which was not mitigated by IaaS already). Again, the provider needs to solve the problem by either reprovisioning the software onto existing resources, or renegotiating with the IaaS provider, or eventually, renegotiating with the end-customer if all else fails.

6.3 Architecture Applicability

As a proof of concept for the generic application of the SAMI architecture, an additional example from a very different area is used, which includes non-technical services as well. As shown in Figure 6.5, there is a meeting organizer who wishes to arrange hotels and rental cars for the attendees. In such a case, the SAMI would be essentially a system to help decision-making and indicate favorable options. Negotiation would be driven by humans, but the SAMI's negotiation interface/mechanisms could be used. Also, the SAMI's planning and optimization mechanism would monitor existing resources, know their semantics and suggest how to proceed with a negotiation. As soon as the latter concluded, it would book an account manager's agenda, reserve rooms, or reserve cars depending on the service at hand. Monitoring would be implemented through possible complaints by attendees; the account manager would receive them, and either forward them to the hotel/car rental, try to renegotiate, or try to find a different subcontractor.

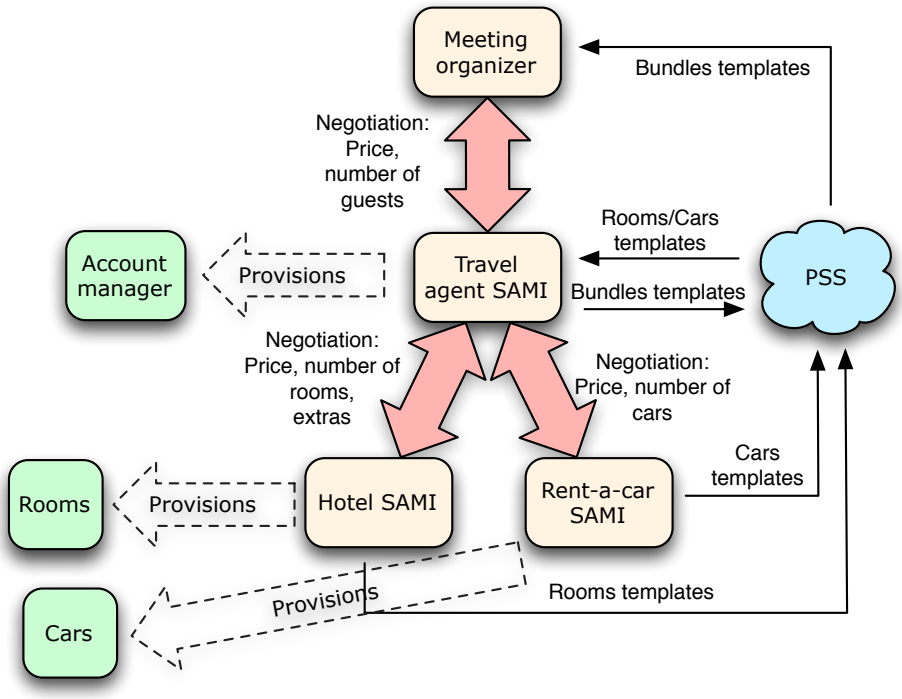

Figure 6.5: Meeting organization outsourcing

This example illustrates the generic applicability of the SAMI architecture, even in domains completely unrelated to IT. Concerns remain sep-

arated throughout the SLA life cycle, and the domain-agnostic approach followed facilitates using the same concepts and architecture in very diverse scenarios.

6.4 Framework Implementation

The nature of the SAMI and the requirement to make it applicable in a wide spectrum of very different use cases, enforce the need to implement it as a reusable *framework*. The SAMI has been implemented as part of the SLA@SOI project, based on technologies that allow domain-specific methodologies to be inserted in the form of *plugins*, interacting via the SAMI kernel. Thanks to this plugin-based pattern, new implementations of the various components can be easily added or even replaced at runtime.

A plugin in SAMI is implementing one of the components of the architecture, for example, the POA module. As such, a plugin provides a specific set of functionalities, within the context of SLA management, based on pre-specified interfaces. The SAMI kernel provides the infrastructure required to support the activation, operation and interaction of each of its components. This infrastructure is based on the SpringDM [48] application framework for the OSGi [49] technology, in which a plugin is known as bundle.

The key feature of this approach is the high degree of flexibility provided for dynamic behavior, and customizable system deployment and reconfiguration. Each SAMI implementation can customize or reuse components, integrate new ones or replace others (even at runtime) with minimal effort. Some of the main features of the plugin-based SAMI architecture are as follows:

- *Simplicity:* The life cycle of each plugin or bundle is handled by the SAMI kernel and supported by an OSGi container;
- *Adaptive behavior:* Given different mechanisms and implementations for different system conditions, it is straightforward to replace one with another when the administrator sees that fit, or automatically according to predefined rules and policies;
- *Easy deployment:* SAMI is able to run under any implementation of an OSGi container. To the best of the author's knowledge, the most complete implementation of the latest OSGi specification (R4) is Equinox [50]; it is therefore the one on which SAMI is being currently tested;

- *Versioning:* Due to the full support for replacement of the various components, when a new version of a component is available it is straightforward to exchange versions and upgrade to the latest one. Nevertheless, rollback is similarly easy if problems appear.

- *Reusability:* The plugin-based architecture also allows the integration of third-party libraries inside each bundle. This encapsulation feature improves code reusability within SAMI implementations. For example, if applied to a *High-Performance Computing* (HPC) domain, one can wrap an existing scheduler with the interfaces foreseen by the POA module, and therefore reuse the scheduler in this new context of SLA planning.

Currently, the plugins implemented or under implementation in the context of the SLA@SOI Project are the following:

- *Negotiation:* The Negotiation module includes a *Syntax Converter*, that converts from on-the-wire representations (e.g. WS-Agreement and other XML renderings) to internal SLA models, and vice-versa; a *Protocol Engine* that takes care of protocol-related formalisms and enforces the respective consistency; a *Publish/Subscribe client* that advertises templates and receives advertisements; a *Template Registry* that stores templates and can be queried *using* templates as input; and an *SLA Registry* that holds established SLAs.

- *POA:* The POA module is custom per domain and/or use case. One implementation already completed is for infrastructure services, partially using the ideas from Chapters 10 and 11. An additional implementation for a demo application with software services is also underway.

- *Provisioning and Monitoring:* The two components have been implemented as one in the project, and comprise two different instances; one for infrastructure services, and one for the aforementioned demonstrator.

At the moment of writing, integration tests are taking place, also converging with parallel ongoing work on the topic of software service prediction services (“Forecasting”) and Dependency Management input. The latter comes in the form of queries to a *Service Manager* implemented within the project.

The SAMI implementation as described so far is a service itself, with its functionality concerning SLA management in complete. Nevertheless, a different implementation approach is to enable the components of the

SAMI architecture as distinct services. It could, therefore, be possible to have services to negotiate, others to plan, to monitor, etc, allowing even further flexibility.

6.5 Related Work

In [51], Ludwig et al. present an architecture for the management of SLAs during negotiation and runtime. Their work is specifically targeting the WS-Agreement standard, and does not discuss renegotiation of SLAs or SLA hierarchy management. The latter is neither discussed in [52] by Padgett et al., which is furthermore tightly coupled to Grid Computing. In [53], Koumoutsos et al. present an abstract architecture for managing SLAs through their complete life cycle, but no information is provided with regards to the interactions of the various components for achieving the purpose, while additionally they concentrate on information management and SLA modelling.

In [54], Keller et al. present the *Web Service Level Agreement* (WSLA) language for defining SLAs, alongside an architecture for monitoring them. Focusing on the architecture part, it only concerns the runtime, without reference to negotiation-time mechanisms. The same limitation to runtime management appears in [55], where an architecture based on the *Common Information Model* (CIM) [56] is presented by Debusmann and Keller. Apart from being limited to monitoring of SLAs, this work is also tightly coupled to specific technologies (CIM and WSLA). In [57], Bartolini et al. elaborate on an SLA management framework, which is very complete from a runtime perspective, but does not touch on the issue of negotiating and re-negotiating SLAs. The case is similar for [58] from Paschke et al., which applies extensively knowledge management techniques into the domain of SLA management. In [59], Buco et al. are also discussing SLA management after establishment, assuming that to be a manual process.

In [60], Chhetri et al. present an SLA negotiation architecture. Although they include a “SLA life cycle management” component, they do not discuss how it is used. In [61], Bartolini et al. present an abstract framework like the one presented in this thesis (including, additionally, negotiation rules), but it is also limited to negotiation time only. Similarly, in [45] Di Nitto et al. focus on negotiation, especially on protocol management and assuming a marketplace approach. In [62] Kim and Segev discuss SLA negotiation only, as part of an abstract framework, which is similar to the present one, in the sense that it also separates concerns between protocol management and decision making. Finally, in [63],

Ayatollahzadeh-Shirazi and Barfouroush present a conceptual framework for negotiating SLAs using generic negotiation principles, but do not discuss runtime management either.

Also close to the work discussed here, is [64] by Dan, Ludwig and Pacifici. The authors present an architecture that includes all those constructs for interacting with customers, provisioning, monitoring and adjusting SLAs. Nevertheless, multi-level (hierarchical) SLAs are not addressed. In addition, their work is described around WSLA and web services in specific, while the SAMI is technology-independent.

Overall, the main novelty introduced by the SAMI architecture can be summarized as follows:

- It is technology-independent: No reliance on web services or other specific technology.
- It brings specific support for SLA hierarchies and chained negotiation, which is necessary for service chains / hierarchies.

In addition, these features are unified with capabilities that were found in prior art, albeit fragmented:

- It covers the complete life cycle of SLAs, including negotiation and execution (monitoring / enforcement).
- It can be customized to different domains; specifically, it innovates by separating protocol-related negotiation functionality from utility and strategy concepts; and by foreseeing a pluggable mechanism for the support of different planning mechanisms.

Summary and Conclusions

SAMI, the SLA Management Instance, was presented in this chapter. SAMI is an architectural design for achieving SLA management, throughout the complete life cycle of SLAs – most notably including negotiation and execution. SAMI's essential principles are extensibility, as well as independence from specific domains and technologies. SLA hierarchies are specifically taken into account, by means of SLA template advertisements, service discovery and composition, SLA Translation interlaced with SLA Negotiation, and hierarchical SLA Provisioning.

In the next chapter, a novel penalty model is formalized and presented. Such a penalty model describes what are the costs that a provider must pay, if an SLA is not honored. It is a necessary consideration both during negotiation (for the Planning and Optimization), but also during

runtime (for deciding the most appropriate course of action when an SLA is violated or about to be violated).

Chapter 7

Penalty Model

One important differentiation of SLAs from best-effort service provisioning requests is the annotation with penalties. That is, all those provisions that define what will happen in the case that a provider fails to deliver the service, as agreed. The consequences of such a failure may be some kind of refunding, additional (free) service points, etc.

The present chapter explores this topic, and proposes a new formalization for penalty definition. This formal model is taking into account requirements for fairness and business value. Following the model's definition, an example is provided that links the model to SLA hierarchies.

7.1 Penalties and Service Level Agreements

Penalties are as an essential part of SLAs as the description of guarantees. The reason for that is the very concept of using SLAs as an instrument to provide some level of determinism in business relations. As such, SLAs also describe what must happen when something goes wrong and the SLA cannot be honored (i.e. it is violated). This section of a Service Level Agreement, describing the penalties, is typically of concern to business and legal departments of companies.

Penalties requested by customers during negotiation indicate the importance of the SLA's compliance, for these customers. In a similar way, the penalties acceptable by the service provider indicate a risk strategy; namely, how far a provider is willing to go to make the customer feel safe, while in the same time reducing the risk for its business. In addition, it may be the case that some penalty is defined for customers who do not respect certain obligations they may have according to the SLA; for instance, exceeding the agreed invocation rate may lead to additional costs.

Penalty *fairness* is important to keep business relationships stable, and to preserve SLAs as a useful and meaningful instrument to define such business relationships. Fairness refers to reasonable and proportional royalties returned when the SLA is violated. This kind of proportionality concerns the interests of both the customer and the provider. As mentioned earlier, the business value of the SLA to the customer is reflected, ideally, in the penalties; while the provider usually does not wish to risk its complete business over a single contract. At the same time, penalties should reflect how far from the agreed QoS level an SLA has drifted, to achieve proportionality. By way of an example, one may consider an SLA where 95% of invocations of some operation are guaranteed to complete within 5 seconds. If 94.9% did so during the accounting period, the SLA is violated – but the penalty will typically be much smaller than if only 80% of the invocations completed within 5 seconds.

Currently, there are various ways to express penalties, some simpler (e.g. flat rate for the whole SLA; or linear proportionality per guarantee) and some more complex – such as the ones discussed in Section 7.4. The various approaches, however, do not satisfy *all* of the following requirements for formulating complex penalty expressions, in a single unambiguous model:

- Capability to describe associative penalties, where the penalty for failing one guarantee of the SLA depends on the state (failed or satisfied) of some other guarantee(s);
- Full flexibility as regards the QoS levels agreed and/or achieved, without constraints such as pre-specified classes of service;
- Openness and applicability to different domains, without dependence of the model on specific languages, taxonomies or technologies.

In this chapter, a formal model of defining penalties within an SLA is provided, taking into account the aforementioned requirements as part of the contract negotiation. The model must be flexible and adaptable to different scenarios, and expressive enough so that complex expressions can be accommodated – e.g. penalties for combinations of different guarantees being violated. In addition, an example of this penalty model is given, applied to the geocoding service scenario from Section 2.2.

7.2 Penalty Model

Let assume service S, and a Service Level Agreement that governs the consumption of this service by a certain customer. Also, that the total cost for the consumption of the service under this SLA is C, and that the agreed Quality of Service is given as a set of guarantees $(Q_1, Q_2, ..., Q_n)$ for the various supported quality metrics and properties. Then, a *set of penalty functions* is defined as

$$P_m(Q_{1_m}, ..., Q_{N_m}) = C \cdot PW_m \cdot \sum_k QW_k \cdot FR_k \tag{7.1}$$

where $m > 0$ and $1_m \leqslant k \leqslant N_m$.

$Q_{1_m}, ..., Q_{N_m}$ represents a combination of guarantees that are depending on each other, and the violation of one may affect, under specific circumstances, the others. It is a way to express correlations of fine granularity. Thus, a customer can express statements such as that a violation of one guarantee is more relevant if another guarantee is also violated.

P_m is a penalty function that corresponds to the above combination $Q_{1_m}, ..., Q_{N_m}$ of guarantees. The sum of all penalty functions during one reporting period represents the total penalties for this SLA, during that period.

PW_m is the *weight* of penalty function P_m. It indicates how important is this function to the total calculated penalty; and is possibly a contributing factor for the service provider to make decisions with regards to the deployment and implementation of the SLA. The sum of all weights is equal to 1.

QW_k is the weight of one specific guarantee being violated, *for this specific combination* of guarantees. *This value may be arbitrarily high*. It offers the capability to the customer, when negotiating, to express the importance of honoring certain guarantees in this penalty function. As an example, one may consider the case where the guarantees concern the availabilities of two load-balancing servers. If the availability guarantee for one of the servers is violated, its weight (and hence, the penalty) is kept small. If, however, on the same time the availability guarantee of the other server is also violated – i.e. the second server becomes unavailable as well –, there may be a very high weight to suggest an equally high penalty for the system becoming unavailable as a whole.

Finally, FR_k is the failure ratio, i.e. the achieved quality vs the planned quality. It indicates how far the offered quality has drifted, when compared to the quality that was agreed to be offered for that specific service parameter. For instance, if availability of a service was agreed to be 100%, but only 90% is achieved, then the failure ratio is 0.1; and if a

response time was agreed to be 5 seconds on average, but 6 seconds is the average response time achieved, then the failure ratio is 0.2. By definition, FR_k may also model possible rewards for performing better than agreed.

Chapter 5 discussed the construction of subcontracts starting from a higher-level contract, in a top-down fashion. In parallel with inferring proper values for the service properties of the SLAs, suitable penalties must be deduced. Therefore, penalty calculation for subcontracts is also part of the SLA Translation process. Due to the domain-specificity of the way that properties depend on each other, it is not possible to provide generic analytical expressions of penalties for antecedent services / SLAs. Rather, the next Section illustrates how this would work, by means of an example.

7.3 Example Application

This section provides an example of using the penalty model, based on the scenario from Section 2.2. It illustrates how the model is applied for the geocoding service GC, and how it is translated to penalties for antecedent services in this context. Let assume that the negotiable properties for the geocoding service are:

- *Availability*, defined as the ratio of service uptime to total monitoring time, over a certain predefined time period (e.g. one month); and
- *Response Time*, defined as the time duration from the moment a message is received, to the moment a response is generated and put on the wire, for a specific percentage of all invocations.

The customer wants to express that availability must be over 99%, and the penalty increases up to the full cost for availability dropping down to 90%. For response time, it is desired that 98% of invocations return in less than 5 seconds. The respective response time penalty increases linearly towards 85% of the calls; at that point, full SLA cost is claimed back. Network delays are not considered in this example. The customer expresses these penalty terms as follows:

$$\mathrm{P}(\mathrm{A}) = \mathrm{C} \cdot 1 \cdot \left(\frac{99}{99-90} \cdot \mathrm{FR}_{\mathrm{A}} \right) \tag{7.2}$$

$$\mathrm{P}(\mathrm{R}) = \mathrm{C} \cdot 1 \cdot \left(\frac{98}{98-85} \cdot \mathrm{FR}_{\mathrm{R}} \right) \tag{7.3}$$

$P(A)$, $P(R)$, are the penalties for availability and response time. C is the total SLA price. For all penalty functions, a maximum penalty equal to the SLA price is requested ($PW_A = PW_R = 1$). Obviously, this may add up to more than the SLA price, should more than one parameters be "sufficiently violated". It may well be the case that the provider does not accept this, but rather sets a maximum aggregate penalty; this is something to be negotiated between the two parties, and its expression is a syntactical matter which is out of scope for this work.

The weight for each guarantee being violated is set such that each unit of additional failure from agreed target contributes linearly to a total failure at the agreed threshold. Thus, $99/(99-90)$ is the necessary number to reach a value of 1 when multiplied with the threshold failure ratio of $(99-90)/99$ for availability (99% being the planned one, 90% being the actual). Similarly, $98/(98-85)$ for the response time threshold of 85% (down from 98%).

The Planning/Optimization component of the geocoding service SAMI knows that the availability of the geocoding service, A_g, depends on the availability of the infrastructure on which the service executes. Its response time, R, depends on utilization of the infrastructure being sufficiently low. Equations 7.4 and 7.5 describe the availability and response time of the geocoding service in relation to the availability and utilization of the supporting VMs. Coefficients k and l are specific to the software used, and are known by the Planning/Optimization component via software modeling, monitoring, or other means. A constant c denotes the response time for near-zero utilization of the VMs. Based on these formulae, the geocoding service provider would deduce the required guaranteed properties and construct an SLA offer towards the infrastructure provider.

$$A_g = \min(A_{VM1}, A_{VM2}) \tag{7.4}$$

$$R = k \cdot U_{VM1} + l \cdot U_{VM2} + c \tag{7.5}$$

$$A_{VM1}, A_{VM2}, U_{VM1}, U_{VM2} \in [0, 1] \tag{7.6}$$

The policy of the geocoding provider is to ensure that full infrastructure SLA costs will be refunded, if the infrastructure provider fails up to an extent such that the *profit* from the geocoding SLA starts being affected. That is; although availability of the geocoding service can become as low as 90% before a full refund is issued; and it relates to VM availability as shown in Equation 7.4; the geocoding provider will ask that it converges sooner to values that constitute "complete failure" to deliver. As such, the failure multipliers must be larger, to reflect this financial difference.

The same applies for response time, and the penalties for infrastructure utilization levels. For the purposes of the example, a naive assumption is made that the geocoding provider has no implementation costs, and that the difference between software SLA price C and infrastructure SLA price C^* is all profit.

Eventually, this penalty can be formulated as in Equations 7.7 and 7.8.

$$P(A_{VM1}, A_{VM2}) = C^* \cdot 1 \cdot \left(\frac{C}{C^*} \cdot \alpha \cdot FR_{A_{VM1}} + \frac{C}{C^*} \cdot \alpha \cdot FR_{A_{VM2}} \right) \quad (7.7)$$

$$P(U_{VM1}, U_{VM2}) = C^* \cdot 1 \cdot \left(\frac{C}{C^*} \cdot \beta \cdot FR_{U_{VM1}} + \frac{C}{C^*} \cdot \beta \cdot FR_{U_{VM2}} \right) \quad (7.8)$$

where $\alpha = 99/(99-90)$ and $\beta = 98/(98-85)$.

One can easily notice that these equations can be simplified as follows:

$$P(A_{VM1}, A_{VM2}) = C \cdot \alpha \cdot (FR_{A_{VM1}} + FR_{A_{VM2}}) \quad (7.9)$$

$$P(U_{VM1}, U_{VM2}) = C \cdot \beta \cdot (FR_{U_{VM1}} + FR_{U_{VM2}}) \quad (7.10)$$

Taking into account Equations 7.2 and 7.3, it holds that:

$$\frac{P(A)}{P(A_{VM1}, A_{VM2})} = \frac{FR_A}{FR_{A_{VM1}} + FR_{A_{VM2}}} \quad (7.11)$$

$$\frac{P(R)}{P(U_{VM1}, U_{VM2})} = \frac{FR_R}{FR_{U_{VM1}} + FR_{U_{VM2}}} \quad (7.12)$$

However, it must be underlined that this applies only because of the assumption that the geocoding service provider has no implementation costs other than the infrastructure for service execution.

7.4 Related Work

In [65], Becker et al. propose a price function over achieved QoS. Subtracting from the agreed price provides the penalty, so it can be considered to be the penalty function as well. Rewards are also possible, using their approach. The main difference with the approach presented in this Chapter, is that in [65] it is not possible to express penalty with more than one QoS properties involved, and these properties being correlated. That is, quality is seen as a unidirectional aggregated measure for a service, while in fact there may be quality characteristics that cannot be aggregated without preference information and also without property inter-dependence information.

In [66], Jurca and Faltings suggest a method to calculate penalties (after using a service and following an SLA) based on a reputation mechanism, where all customers evaluate the quality of the service they received. The authors take measures to avoid false voting for price reduction or other fraud. The approach, however, can only be combined with classes of service.

In [67], Rana et al. discuss monitoring and reputation mechanisms for SLAs. The authors look into EU contract law and take some relevant points into account, then define three broad penalty categories: *All-or-nothing*, *Partial*, *Weighted Partial*. However, a complete mathematical model is not present. The work presented in this paper is then related to WS-Agreement, so the relevant negotiation concepts are also discussed.

Finally, Kosinski et al. present in [68] a mathematical formulation of penalty functions which is fairly similar to what is presented here (although applied specifically in the area of networking). The paper *does* capture the relationships between different properties, and policies are defined depending on such relationships and the respective combinations. These policies are captured within "subcontracts" (the term is used differently than in this thesis; it refers to sections of a single SLA). The main difference to the present work is that in [68] failures are calculated using a pre-specified taxonomy (number of violations, amount of violations, etc), while here the way to calculate the failure ratio is considered domain-specific and left open. In addition, the model from Section 7.2 assigns a weight not only to each violation, but also to each penalty function ("subcontract", using the terminology of [68]).

Summary and Conclusions

In this chapter, a novel, complete mathematical penalty model was introduced and formulated. Expressions compliant to this model can be integrated within SLAs, to define unambiguously the penalties that accompany SLA failures. The model is taking into account concepts of fairness, business value, and quality parameter interdependencies alike.

Via an example, the application of the model to SLA hierarchies was demonstrated. It was shown that it can be applied using domain-specific (and perhaps use-case specific) knowledge, in order to build analytical penalty expressions at negotiation time, dynamically and according to top-down or bottom-up hierarchy construction.

The chapter that follows presents a model for an in-memory SLA representation that can simplify highly complex SLAs, and contribute to making decisions during negotiation and subcontracting.

Chapter 8

System-Internal SLA Representation

Machine-readable SLAs, being a projection of real-world contracts in the IT space, can be arbitrarily complex. They may include conditionals, requirements that must all apply in parallel, optional items that affect final pricing, penalties, rewards, and so on. Processing such complex documents without prior knowledge of their structure (even if the syntax is well-defined and understood) may be a task of significant difficulty. More specifically, this task is equivalent to the *graph isomorphism* problem, which remains open as regards its complexity; its generalization, the *subgraph isomorphism problem*, is known to be NP-Complete [69].

The present chapter proposes a methodology to reduce this complexity, by accepting that SLAs have special properties and can be modeled as boolean functions. Based on this proposition, a structure known as *Reduced Ordered Binary Decision Diagrams* can be used, to result in a canonical representation of SLAs, and allow their processing for reasons of planning and outsourcing. A canonical representation like this can facilitate fast processing of SLAs and decision-making about them during negotiation.

8.1 Basic Concepts

In this chapter it is proposed that a *graph-based* data structure, well-known in the domain of *Computer Aided Design* (CAD) for *Very Large Scale Integrated* (VLSI) circuits, is suitable for modeling SLAs in a way which is both expressive enough, and very efficient. R. Bryant introduced *Reduced Ordered Binary Decision Diagrams* (ROBBDs) in 1986 [70] as an evolution of C.Y. Lee's [71] and S. Akers' [72] work on BDDs. They are

classified as a tool in the area of symbolic model checking, the scientific discipline looking into the problem of verifying that a given system satisfies specific requirements, given any kind of input. The hardware industry race has further contributed to the optimization of the structure itself with a significant amount of relevant research, and a large number of methods already exist for taking advantage of ROBDDs' inherent properties.

It is here reminded, from Section 3.1, that on an abstract level SLAs consist of *facts*, *conditions* and *clauses*; the latter two being here collectively referred to as *rules*, and comprise *if-then-else* structures. The essential reason that ROBDDs are useful for modeling SLAs, is that they are canonical representations generated on the grounds of such *if-then-else* structures. Thus, they can express SLAs unambiguously: equivalent SLAs which are *structurally* different, are eventually represented by the same ROBDD. On the contrary, using formats developed for *on-the-wire* representation (e.g. the various XML-based models that were mentioned in the introductory chapters) does not guarantee this property. It is herein proposed that ROBDDs are used internally in systems which have to manage SLAs, as a representation that facilitates their management. Suitable interpreters should then be developed to convert from standardized, interchangeable formats such as WS-Agreement and WSLA, to this more convenient data structure and vice-versa.

8.2 Binary Decision Diagrams

This section serves as a general, high-level introduction to BDDs and their basic properties. Motivated readers are encouraged to consult with the bibliography for in-depth material. Most of the definitions provided in this section, are summaries of the definitions that can be found in [73] by Ebendt et al.

A BDD is a graph-based representation of one or more boolean functions. This kind of diagram is based on *Shannon's decomposition theorem* [74], which states that, assuming a boolean function $f : X_n \rightarrow X_m$, where $X_n = \{x_1, ..., x_n\}$ and $X_m = \{x_1, ..., x_m\}$, then for any boolean variable x_i, $i \leqslant 1 \leqslant n$:

$$f = x_i \cdot f_{x_i=1} + \overline{x_i} \cdot f_{x_i=0} \tag{8.1}$$

What Equation 8.1 provides, is the *if-then-else* representation sought for: If x_i is *true*, *then* $f_{x_i=1}$ must be evaluated, or *else* $f_{x_i=0}$ must be evaluated. A BDD then, is a *directed acyclic graph* $G = (V, E)$, where V denotes the vertices (nodes) and E the edges. Vertices can be either *terminal* (i.e. their out-degree is equal to zero), or *non-terminal*. The former can carry a value of either **1** (*true*) or **0** (*false*). The latter are

labelled with a variable $x_i \in X_n$; if u is the node, the variable x_i is referred to as $var(u)$. Of the two children nodes, the one followed if x_i evaluates to *true* is referred to as $then(u)$, and the other as $else(u)$.

An illustrative example can be found in [73]. This example is shown in Figure 8.1(a), with a BDD representation of the boolean function $f = x_1 \cdot x_2 + \overline{x_1} \cdot x_3$. Solid lines are typically used for the edge between u and $then(u)$, and dashed lines for the edge between u and $else(u)$. Additionally, non-terminal nodes are denoted as circles, while terminal nodes as squares.

Let π be an ordering of the boolean variables involved in the function to represent. Then, the pair (π, G) is the *Ordered BDD* (OBDD) representation of the function, as long as (additionally to simple BDD definitions) it is true that on each path from the root to a terminal node the variables are encountered at most once and in the same order. Looking into the previous example, Figure 8.1(b) is illustrating exactly this ordering of variables, and how it affects the diagram. A diagram with more than one roots (i.e., representing more than one boolean functions which depend on the same boolean variables) is a *Shared BDD* (SBDD). It must be noted that a root node here does not necessarily imply that the in-degree of this node is equal to zero. For a specific function within a BDD or a SBDD, a *path* is a subset of G which connects the root with a terminal node, without any duplicate occurrences of a node or an edge. The set of all paths for function f is herein denoted as Γ_f.

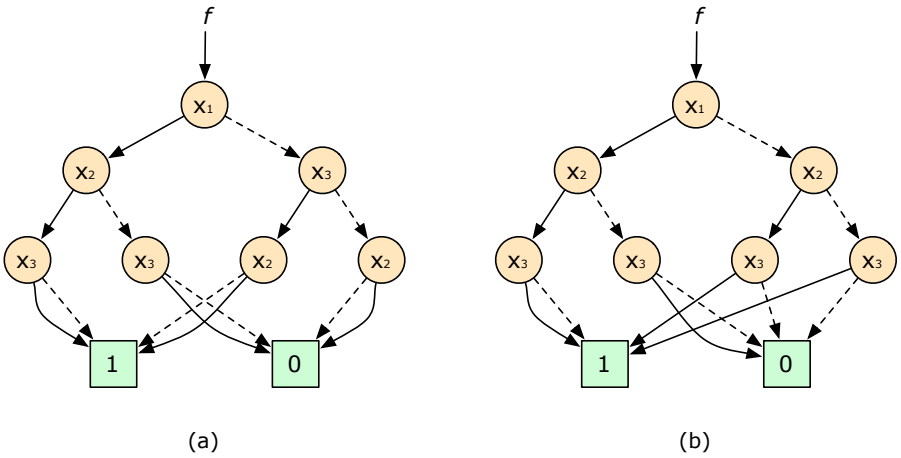

Figure 8.1: Simple/Ordered BDD representations of $f = x_1 \cdot x_2 + \overline{x_1} \cdot x_3$

An ordered BDD can be reduced (resulting in a *Reduced Ordered BDD*), using the following two operations:

1. *Deletion*: If for a non-terminal node u of G holds that $then(u) =$

$else(\mathfrak{u}) = \mathfrak{u}'$, the node can be removed from the graph. All edges pointing to it, if any, must now point to $\mathfrak{u}'$, and if $\mathfrak{u}$ was a root node, then $\mathfrak{u}'$ must be upgraded to a root node.

2. *Merging*: If for two non-terminal nodes $\mathfrak{u}$ and $\mathfrak{u}'$ holds that $var(\mathfrak{u}) = var(\mathfrak{u}')$, $then(\mathfrak{u}) = then(\mathfrak{u}')$ and $else(\mathfrak{u}) = else(\mathfrak{u}')$, then it is possible to remove $\mathfrak{u}$ and have all edges pointing to it redirected to point to $\mathfrak{u}'$. Additionally, if $\mathfrak{u}$ is a root node, then $\mathfrak{u}'$ must be made into a root node.

Remark: In the text that follows, the term *BDD* will be used universally, to refer to Reduced Ordered BDDs. Also, there will be no distinction between single-rooted and shared diagrams. Whenever single-rooted BDDs are *explicitly excluded*, this will be denoted by pre-pending "shared" or just the letter "S".

8.3 SLAs as BDDs

8.3.1 A Usage Example

In order to illustrate more intuitively the usage of this structure for SLAs, the example from Section 2.2 will be used. As discussed, this kind of business scenario involves two SLAs. The first (s-SLA) is established between the end-customer and the SaaS provider, to govern their interactions and apply guarantees. The second (i-SLA) is established between the SaaS and the IaaS providers for the same purpose.

The customer would try to engage in an SLA with the geo-location service provider, which would involve –for instance– metrics for service availability, and service invocations completion time (CT). The SaaS provider would apply its understanding about the software (based on modeling principles or historical monitoring evidence) to derive expected resource requirements, possibly varying throughout a day's, month's or other period. The infrastructure resource requirements, on the other hand, would be the guarantees that the SaaS provider's SLA with the IaaS provider would need to include. The example SLAs are described as follows:

SLA-1: For service "Geo-location", and given that business hours are between 09:00 and 17:00: During business hours, operation "Plan-Route" must complete within 5 seconds, and the service's availability must be more than 99%. Outside business hours, completion time for the same operation can be up to 10 seconds, and the service's availability must be more than 95%.

SLA-2: For service "VMpool", and given that business hours are between 09:00 and 17:00: During business hours, 4 virtual machines must be allocated to this contract. Outside business hours, 2 virtual machines must be allocated.

SLA	Variable	Proposition	Proposition type
SLA-1	x_1	ServiceName = 'Geo-location'	Fact
SLA-1	x_2	BusinessHours = 09:00 - 17:00	Fact
SLA-1	x_3	TimeOfDay in BusinessHours	Condition
SLA-1	x_4	'PlanRoute' CT $<$ 5 sec	Clause
SLA-1	x_5	Geo-location availability $>$ 99%	Clause
SLA-1	$\overline{x_3}$	TimeOfDay not in BusinessHours	Condition
SLA-1	x_6	'PlanRoute' CT $<$ 10 sec	Clause
SLA-1	x_7	Geo-location availability $>$ 95%	Clause
SLA-2	y_1	ServiceName = 'VMpool'	Fact
SLA-2	y_2	BusinessHours = 09:00 - 17:00	Fact
SLA-2	y_3	TimeOfDay in BusinessHours	Condition
SLA-2	y_4	Number of VMs = 4	Clause
SLA-2	$\overline{y_3}$	TimeOfDay not in BusinessHours	Condition
SLA-2	y_5	Number of VMs = 2	Clause

Table 8.1: Example clauses

Table 8.1 illustrates the set of facts and clauses to be used for this example scenario. It is straightforward to see that, given these facts and clauses in the form of boolean variables which evaluate to *true* or *false*, the SLAs can also evaluate correctly if they are modeled according to Equations 8.2 and 8.3 respectively. In the upcoming Section 8.3.2, the problem of expressing SLAs as boolean functions will be formalized. Then in Section 8.3.3 it will be shown how these specific example SLAs map to BDDs.

$$f = x_1 \cdot x_2 \cdot (x_3 \cdot x_4 \cdot x_5 + \overline{x_3} \cdot x_6 \cdot x_7) \tag{8.2}$$

$$g = y_1 \cdot y_2 \cdot (y_3 \cdot y_4 + \overline{y_3} \cdot y_5) \tag{8.3}$$

8.3.2 SLAs and SLA Hierarchies

Here follows a proposed SLA representation: Let Φ^n be the universe of *facts* applicable to contracts as indisputable truth, $\Phi^n = \{\phi_1, ..., \phi_n\}$. Also let Y^m be the universe of *clauses* which can be evaluated to either $true$ or $false$, $Y^m = \{y_1, ..., y_m\}$. A Service Level Agreement is the boolean

function f:

$$f : F^k \cup Z^l \rightarrow \{0, 1\} \tag{8.4}$$

where $F^k \subseteq \Phi^n$, $F^k = \{\phi_1, ..., \phi_k\}$ and $Z^l \subseteq Y^m$, $Z^l = \{z_1, ..., z_l\}$.

Hence, there exists a representation of an SLA as a boolean function, taking advantage of an SLA's binary nature upon evaluation as *possible* / *impossible* to satisfy (at negotiation time) or *honored* / *violated* (at runtime, i.e. while the service is being consumed). The variable terms of an SLA are taking values from Z^l, while pre-agreed understanding (and in general, facts about the world) is encoded in facts accepting values from F^k. This definition is broad enough to encompass various previous definitions, both conceptual (e.g. [75]) and syntactical (e.g. WS-Agreement).

In Chapter 5, service hierarchies and the corresponding SLA hierarchies (contracts and sub-contracts) were discussed. There was also a formal definition of the relationship between one property of a service, and all relevant properties of the services that it depends on (i.e. service dependencies). It is now possible to codify generically *SLA dependencies*, in a manner that allows enough flexibility to describe any such kind. Let:

- $f : F^k \cup Z^l \rightarrow \{0, 1\}$, the dependent SLA
- $f_i : F^{ki} \cup Z^{li} \rightarrow \{0, 1\}$, $i \in \mathbb{N}$, the depending SLAs
- $F^{ki} \subseteq \Phi^n$, $F^{ki} = \{\phi_{1i}, ..., \phi_{ki}\}$
- $Z^{li} \subseteq Y^m$, $Z^{li} = \{z_{1i}, ..., z_{li}\}$

There is defined the dependency of f upon $\{f_i\}$ (and therefore the resulting hierarchy) as a function g:

$$g : Z^l \rightarrow \cup_i (Z^{li}) | F^k \cup_i (F^{ki}) \tag{8.5}$$

Simply put, a function of any number of *variable* terms from SLA f equals a function of any number of *variable* terms from one or more SLAs f_i, under the circumstances defined by the relevant fact sets. Operating under this highly abstract definition allows us the required flexibility to describe contracts with dependencies of any kind, as long as each of them does eventually evaluate to either *true*, or *false*.

8.3.3 BDD Mapping

There now exists a formal representation of SLAs (Equation 8.4) and SLA dependencies (Equation 8.5). The gain in using BDDs lies in *reduction*. Through this process, a BDD becomes a *canonical* representation of the

boolean function it describes, as proven in [70]. Therefore, an SLA described as a boolean function in the form of a BDD takes a unique, well-specified and minimal form, eliminating redundancy and allowing to make the mapping which describes SLA dependencies far more efficient than what it would be if complete graphs were processed. Additionally, the canonical form of the SLAs allows objective evaluation and comparison for maximizing utility.

The exact method to construct a BDD from an SLA depends on the format in which this SLA is originally expressed, and therefore it cannot be algorithmically defined in a universal way. In the case of WS-Agreement, the `Context` and `Service Description Terms` would be used as facts; `Qualifying Conditions` as conditions; `Guarantee Terms` as clauses; and `Term Compositor Terms` could be classified as either conditions or clauses. In fact, WS-Agreement's Term Compositor Terms are essentially boolean operators: `All` (AND), `OneOrMore` (OR), `ExactlyOne` (XOR). Using this predefined knowledge for such a specific SLA language, it is straightforward to implement a parser that can read the documents and construct a (Reduced Ordered) BDD on-the-fly as described in [76] by Bryant with the revised "APPLY" operation.

Using the example also discussed earlier in Section 8.3.1, it is possible to illustrate the reduced form of BDDs representing SLAs. As mentioned, Equations 8.2 and 8.3 represent the two example SLAs as boolean functions of the variables from Table 8.1. Then, assuming an ordering corresponding to the numbering of the variables, the two resulting BDDs would be as in Figure 8.2.

The main deficiency of BDDs is their reliance on the ordering of the variables. The size of a BDD for the same function may vary from linear to exponential, depending on how variables are ordered [70]. Generic algorithms for near-optimal orderings of variables during or after BDD construction have been researched extensively in the past (e.g. [77, 78]). The application to the domain of SLA management and the involvement of *facts* as variables, whose *else* edge always points directly to terminal node **0**, provides already a possibility for optimizing the BDD by pushing all facts to the top of the diagram. Although this kind of ordering does not reduce the total number of nodes, it allows to ensure that indisputable facts are honored by all parts of the SLA, otherwise it will evaluate to false at runtime (i.e. it is violated). Also, at negotiation time, this ordering may speed up the negotiation process significantly, since the first thing to be confirmed as acceptable (or not) is the agreement of the involved parties on the essentials of the contract (for instance, monetary unit). It should be underlined, at this point, that facts are propositions which apply to

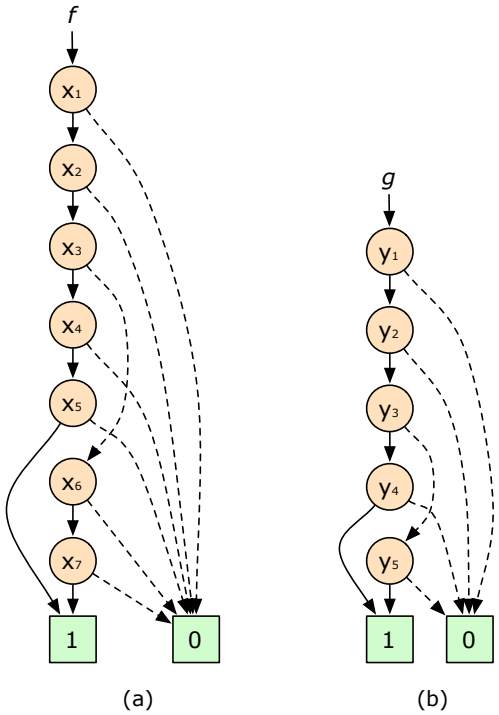

Figure 8.2: The BDDs corresponding to functions from Equations 8.2 and 8.3

the complete contract, and govern all terms included. Therefore, in certain cases, additional attention is required for choosing what is a fact and what is not. Considering, for instance, the case of a two-party contract with two sections describing the obligations of each party, starting each section with an indication as to which party it applies: The statement "section (a) describes the obligations of party (A)" is certainly true for the complete contract. Nevertheless, if reference to the section includes some contract-locality constraint, e.g. "*this* section describes the obligations of party (A)", then this causes ambiguity and cannot apply to the whole contract any more – therefore should be modeled as a *condition*.

After ordering *facts* at the beginning of the diagram, one of the existing BDD methods to optimize the ordering of *conditions* and *clauses* is assumed to be used. Additionally to generic methods described in relevant literature, a kind of structural optimization that takes advantage of the semantics of SLAs and may be applied here is one that considers what is more crucial to the user. Certain SLA representations contain sections on *Business Values*, that may reference specific terms as regards their importance (in Chapter 7 they were used as coefficients to

failure ratios). Given proper formalization of such sections, a constructor of BDDs from SLAs can take them into account and order clause variables from maximum to minimum importance, thus allowing faster evaluation of business-critical terms.

It is now possible to discuss additional principles for the SLA application domain, and for outsourcing parts or all of the contract. Starting from the very semantics of SLAs represented as BDDs, one must distinguish between the meaning of a boolean variable (and the whole diagram) during negotiation time, and during runtime.

8.3.4 Negotiation Time Operations

During negotiation time, the evaluation of a fact variable to *true* or *false* shows whether the fact is recognized as such from the receiving party. For conditions and clauses, it indicates whether there is any reachable state based on assignments of respective variables, so that the condition / clause under examination can eventually evaluate to *true*. Extending this to the complete diagram, at negotiation time it is important to see if there exist, in general, truth assignments for the whole set $F^k \cup Z^l$ which satisfy the diagram and lead to **1**. At this point lies an implicit decision. The party that receives the offer needs to have some certainty that it can honor it after signing. It is a policy issue if this certainty needs to be 100%, or near that, or even much lower (perhaps indicating a high-risk strategy). Whatever the policy, the decision will have to be taken based on some objective criteria. A certainty of 100% would mean that paths of the BDD must be checked for *tautology*, that is, any truth assignment for a path will lead to terminal node **1**. If tautology applies for a single path, that should be enough to accept the offer. Should more than one paths lead to **1**, then the provider must apply additional criteria to make a choice as regards which path should be chosen. Such criteria would typically involve utility, that is, which path would provide the most returns. This is, for instance, directly applicable to the the IaaS use case of Part III. If all paths refer to infrastructure resources that can be provided, then the one that yields the highest profit would naturally be selected.

If there is no path leading with certainty to **1**, it is necessary to make an educated guess whether the offer is acceptable, and whether some part needs to be subcontracted. A simple calculation that can be performed, is the following: Let Γ_f^1 be the set of all paths for f that connect the root to terminal node **1**, and Γ_f^0 the respective set of paths leading to terminal node **0**. It is assumed that by means of historical monitoring information, forecasting, or simply common sense (e.g. time of day) there is assigned to each node u_i in $h \in \Gamma_f^1$ a probability $P'(u_i)$ to evaluate to a

result so that node u_{i+1} is (also) on the same path, and $1-p$ to evaluate otherwise. If the variables of the nodes in the path are *dependent*, then one needs to take this into account and calculate the *conditional probability* of each variable, given the evaluation of all previous variables on this path:

$$P(u_{i+1}) = P'(u_{i+1}|u_1 \cap u_2 \cap ... \cap u_i) \tag{8.6}$$

In somewhat less formal notation, the variables (and the events of them taking a value of *true* or *false*) were represented by the names of their nodes. If the variables are *independent*, then $P(u_i) = P'(u_i)$, $\forall i$. The probability p_h that the complete path evaluates to *true*, is

$$p_h = \prod_u P(u)|u \in h \tag{8.7}$$

Then, the total probability that the SLA can be honored if established, is

$$C = \sum_h (p_h)|h \in \Gamma_f^1 \tag{8.8}$$

Assuming that the acquisition of this probability per node can be performed in constant time, then the complexity of estimating this probability per path is $O(n)$. The consequent requirement to minimize the total number of paths at construction time or variable ordering time, should also be taken into account.

A negotiating party will want C to exceed some threshold, in order to agree to an offer that was received. If this is not the case, then the party (typically, a service provider) will have to either reject the offer, or try to increase C by subcontracting one or more paths and thus increase their contribution to the total success probability. Representing SLAs as BDDs is most useful at this point: The canonical and reduced form of a BDD produces a tractable list of options with regard to what can be assigned to subcontractors. For items in such a list, due to the specific ordering of variables, it is possible to devise unique and unambiguous signatures. The latter may then be associated to different boolean functions, which represent candidate subcontracts. Domain-specific intelligence can be applied by area experts before operation starts, and define the dependencies of certain variables on others for subcontracts. Then, a system based on these principles can make use of this knowledge, and construct proper offers towards third parties. In order to do this, such a system would use the principles discussed in Section 5.1.2. As long as these offers are accepted, and the respective second-level SLAs are established, it should be the case that the corresponding path has increased certainty to complete successfully as regards honoring the first-level SLA. The negotiating party has a choice, according to policies and strategies, to modify the offer and return it with specific values for the variables of that path

(practically suggesting the SLA equivalent of the path), or to accept the complete SLA as long as the increase in C is sufficient.

Coming back to the example scenario from Section 8.3.1, there are two possible ways where this kind of subcontracting is / may be needed. The first, is the explicitly mentioned subcontracting from the SaaS to the IaaS provider. Conceptually, since the SaaS provider has no infrastructure, it cannot offer the service at all unless it subcontracts for infrastructure. Terms x_4, x_5, x_6 and x_7 would always evaluate to *false* unless infrastructure resources are available for the software to execute on. As such, the SaaS provider has to go through this translation process in any case, to calculate infrastructure requirements and make a respective offer to the IaaS provider. If an agreement with the IaaS provider already exists, the contracting system in use should find this automatically after the translation occurs, try to reuse it if possible, otherwise resolve to making a new offer. It must be noted here that, since the outsourcing concerns *paths*, the SaaS provider may just as well make two different offers to two different infrastructure providers (one for each of the two paths in Γ_f^1), or can make a single offer to one infrastructure provider for both paths (this is the working assumption in the example scenario).

The second case, is if it so happens that the IaaS provider cannot satisfy the incoming offer – for instance, does not have the resources to offer the requested performance during business hours. This means that, according to its estimation, y_4 would evaluate to *false* most of the times, and therefore path $y_1 - y_2 - y_3 - y_4 - \mathbf{1}$ would contribute minimally or not at all to the whole agreement's C-value. In this case, the IaaS provider can reject the offer, or – depending on projected utility – try to outsource this path to another IaaS provider.

It should be mentioned that an offer may be for a single SLA, or for multiple SLAs (typically for different services or groups of services) in the form of a Shared BDD. The presented working assumption of an offer for a single SLA does not affect generality.

8.3.5 Runtime Operations

For the runtime scenario a monitoring subsystem is assumed, that can capture service execution-related events from various sources and detect if some SLA term is being violated. The process actually starts much earlier, during negotiation. At that time already, it is beneficial to verify that terms of an agreement *can* actually be monitored (this has already been further discussed in Section 3.2 and [79]). Following this verification step, as part of the negotiation process, an SLA may be formally established, perhaps relying on other SLAs for its existence.

While the service is being consumed, incoming events are processed and terms (in the form of boolean variables of the BDD) are examined to see if a violation has occurred. The ordering of the variables allows the linear-time confirmation, starting from the root and traversing the diagram towards terminal nodes. As each variable evaluates to *true* or *false*, the respective child (*then*/*else*) is followed until a terminal node is reached. If that node is **0**, then there exists a violation, and the reason of failing at that specific part of the SLA should be assessed. Depending on whether this failure happened on a path which was outsourced, or not, there may be a re-negotiation initiated, penalties claimed, or simply adjust the method to estimate success probabilities for different paths. Additionally to corrective actions, such an event must be logged to be reused in next negotiation cycles.

The exact methodology to use in order to avoid unnecessary evaluations of the complete diagram, depends on the monitoring system, the way to evaluate each variable, and the acceptable time thresholds for reaction to violations. A complete definition of such methodologies is out of scope for this work.

8.4 Proof of Concept

As a proof of concept, a simple prototype was built to assess *the approach' efficiency in calculating the feasibility of [complying with] an incoming SLA offer*, if it is established. The prototype accepts an SLA already expressed as a boolean function in *Reverse Polish Notation* (RPN) form, produces a BDD from it and assigns probabilities to the nodes in a semi-random way. Then, it calculates the paths leading to **1**, their probabilities to be followed and the total probability that the SLA can be honored without any subcontracting. Experimentation was with a single SLA offer, which was crafted not to contain dependent variables, according to the following description:

> *The SLA concerns service "Geo-location" (*fact*). Business hours are set to 09:00-17:00 (*fact*). The whole system must run in isolation from other customers of the service provider (*fact*). If the operation invoked is "PlanRoute" (*condition*), and time is within business hours (*condition*), then: completion time should be less than 5 seconds (*clause*); availability should be more than 99% (*clause*); and throughput should be more than 100 operations per minute (*clause*). For times outside business hours: completion time should be less than 10 seconds, availability should be more than 95%, and throughput*

*should be more than 50 operations per second. For operations other than "PlanRoute", invocations should be authenticated (*clause*) and availability should be more than 98%.*

With regard to the assignment of probabilities to the nodes and the paths to follow, there was assigned a probability equal to 1.0 to facts and to the proposition of authenticated invocations (this being a functional requirement that the provider should be aware of). It was then assumed that invocations of "PlanRoute" are one out of three, i.e. a probability of roughly 0.33, and the same for the time of day being within business hours – so it is implied that invocations of the service are equally distributed throughout the day. Finally, for the propositions of completion time, availability and throughput, there was randomly assigned on each node a probability between 0.8 and 1.0 that the provider can satisfy it or will fail (a second random number indicates which of the two applies). In a real scenario, the provider would calculate these probabilities based on monitoring, forecasting or other information. Eventually, this simple scenario was run 10000 times, to see under these semi-random conditions how the SLA success estimations behave. Constructing the BDD for this specific SLA took place in a mere 2.2 seconds. Running the 10000 probability tests took approximately 4 seconds on a 2.4 GHz processor. The diagram contained 16 levels, excluding terminal nodes. Of the 21 paths leading to terminal node **1**, the shortest was 6-nodes long (excluding **1**), and the longest was 13-nodes long.

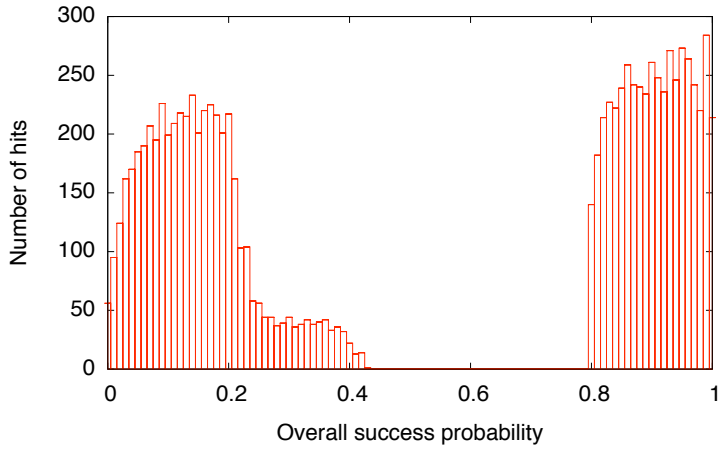

Figure 8.3: Experimental result

Figure 8.3 illustrates *the overall calculated probability that the SLA will be successful if established*, in the form of a histogram for these 10000

runs. For each of the runs, a total success probability for the whole SLA was produced, then it was added to a counter of runs with the same result in order to produce the histogram. Allowing some certainty for individual terms (80%-100% probability of success or failure) results in a clear gap between SLAs projected to fail, and those projected to succeed. This is an indication that, using BDDs in this context and under such circumstances, it is possible to calculate in *only a few milliseconds* and with a *reasonable amount of certainty*, whether the complete SLA can be satisfied or not.

From this experimental evaluation, the feasibility and validity of the approach is exhibited for all SLAs that consist of propositions evaluating to *true* or *false*. As long as all invariable statements of an SLA (e.g. references to other SLAs) can be expressed as facts, and all variable statements can be expressed as conditions and clauses, this assumption is valid for any SLA.

8.5 Related Work

To the best of the author's knowledge, this is the first work that uses BDDs to model and verify SLAs and SLA dependencies. That said, BDDs have been used in service computing before, albeit in very few occasions. In [80], Binder et al. are using a special form of BDDs, called *Zero-Suppressed* BDDs, to create compact digests of service advertisements. Then, the digests are distributed to interested parties which use them for their service composition needs. In [81], Campailla et al. are using BDDs for matching service advertisements in publish-subscribe systems (making use of equivalence checking).

With regard to SLA modeling in general, the most well-known efforts are WS-Agreement and WSLA. As also mentioned in the previous section, the focus of these specifications is on-the-wire representations for enabling interoperability between independent agents. This is an area that is not targeted with the work presented in this Chapter; rather, the focus here is a system-internal representation, that will enable efficient mechanisms for decision making.

As regards applying logic-based approaches to the topic of SLA management, the work which comes closest to the present one, is the one described in [58] by Paschke et al. There, the authors look at the problem in more detail, defining constructs also for things such service description, pricing, QoS, etc. Conversely, in this chapter everything is addressed in an abstract way, and external syntactical definitions / appropriate architectural patterns are assumed for applying these definitions. Additional differences include the explicit focus on managing hierarchies of SLAs

and associations between them as such. The necessary constructs for this kind of functionality also exist in [58], however there is no mention of essential facilities such as equivalence checks and translation between different vocabularies for different layers of a complete IT stack.

One should also not disregard the breadth of tools that exist for building, optimizing and processing BDDs. As part of the SLA@SOI project, a software artifact was prepared to parse SLAs into BDDs, using a readily available BDD construction Java library [82] in only a few days. Libraries like that, and the algorithmic work that has taken place in this area over the last decades, provides significant tooling that can be readily used for fast and efficient SLA processing. This is a very important aspect of the contribution outlined in this Chapter.

Summary and Conclusions

In this Chapter there was presented a novel application of (Shared) Reduced Ordered Binary Decision Diagrams, for representing and managing SLAs, as well as facilitating the construction of SLA hierarchies. BDDs are graph-based structures which have been used for decades in the field of VLSI design and verification, with particular success. They are one of the main tools of the VLSI industry for testing prototypes, and therefore BDDs are a topic under heavy research for decades. The depth and breadth of existing ideas and research can be applied to SLA management for further advancement of this complex service management area. In this particular Chapter there was discussion on the representation through a formal definition of SLAs as boolean functions and from there as BDDs; explained the advantages of this approach; and showed how such kind of use is possible for negotiating SLAs, subcontracting (leading to implicit SLA hierarchies) and detecting SLA violations. Finally, experimental results of applying BDDs to SLA representation were briefly discussed.

In Part III that follows, an application of the principles from Part II will be illustrated in the context of Cloud Computing and, more specifically, Infrastructure as a Service. In addition, there will be an evaluation of using SLAs as a means for infrastructure capacity planning and resource scheduling.

Part III

Infrastructure SLA Planning

Chapter 9

Resource Model

Although SLA models define structurally a general context for describing electronic contracts (an *envelope* as referred to in many cases), they usually do not provide languages and models for the SLA content. This is on purpose, in order to achieve generic use of the various envelope models.

In this Chapter there is provided a *resource description model* that can be used within infrastructure SLAs. This model, called *Resource Element* (RE), is an abstract and standards-compatible construct to describe dynamic groups of resources in a generic manner. Given that resource pools can be described by the RE, it is also used as an abstraction for a *Resource Manager* that is invoked by the respective SAMI's "Provisioning" module from Chapter 6. The relevant invocations are for reasons of *Advance* and *Immediate Reservation*, as a means of resource leasing and provisioning.

9.1 Rationale and Requirements

In certain usage scenarios, infrastructure resources need to be accessed by multiple users concurrently; while in others, exclusive operation is requested. In both cases, the maximum number of users accessing the resources must be controlled – either to enable an acceptable level of service quality, or to comply with exclusive operation requirements. This functionality implies the requirement for advance or immediate reservation of resources, for which an infrastructure service architecture and a set of operations and reservation attributes need to be defined. Not only is reservation required for the management of access concurrency, but also to integrate control tasks such as job submission and data staging into complex workflows. This is further exaggerated when there are specific QoS requirements from such workflows, and therefore some level of

predictable behavior is needed [83, 84]. Therefore, reservation of a resource when shared use is possible, provides a means to support strict guarantees and thus contributes to predictable QoS [85].

The reservation of resources, especially when multiple of them are involved and must be reserved together, implies the need for a model that can describe resources and groups of them. In this chapter a generic model is provided, alongside primitives for declaring resource groups, and for reserving them. Using these, it is possible to formulate relationships between resources, and thus express requirements for resource co-allocation. The resulting data types can be used inside the envelope of a generic negotiation protocol, but can also be directly supported by resource pools attached to a SAMI. In addition to the model and data types, there is provided a set of generic operations that may be supported by these resource pools, and therefore used by the SAMI's provisioning module during the negotiation and provisioning phases.

It is important to note that, although when speaking of IaaS one typically focuses on computing, storage and network resources, there is no particular reason why other types of resources would not be considered. Such resources may include sensors, scientific instruments, robotic and industrial equipment, etc. The presented model takes that into account, and uses this diversity as a requirement.

On a more specific technical level, the following requirements are adopted. Firstly, it is necessary to be able to define groups of similar or identical resources, to which the reservations will correspond. This is for usability reasons: Reserving large amounts of resources by identifying them, one by one, would be impractical. Via the definition of groups which correspond to the inherent details of resources, it is possible to use more complex expressions for resource selection. The reservation of a single group allows the simultaneous booking of all resources in the group. There are assumed *"get"*, *"set"* (which also includes *"update"* semantics) and *"cancel"* operations for the reservations. Additionally, functional and non-functional group properties –which are subject to infrequent changes and only meant for resource selection purposes– are expected to be available.

Based on the rationale elaborated above, the aforementioned requirements are interpreted into the following categories of functionality:

Execution of reservations: Reservation of groups by identification, by resource type and properties.

Provision of reservation metadata: Provision of metadata for a specific reservation and for all reservations.

Handling of existing reservations: Update and cancellation of a reservation.

9.2 Resource Element Design

9.2.1 The Resource Element Model

The *Resource Element* (RE) is defined to be the service representing and virtualizing resources within the IaaS provider's domain, or one of its subdomains (for instance, different data centers of the same provider). It allows access, manipulation and management of a set of resources via the exposure of resource-specific, as well as general-purpose primitives. As a single RE may represent multiple resources of different kinds, use cases can require individual operations to be performed atomically on groups of the real resources represented by this RE. These groups are required to include resources that are supporting semantically and syntactically identical operations, such that the invocation of a given group primitive ensures the operation to be atomically executed on all the elements that are part of the group itself. This model will now be expressed in terms of CIM, and its compliance will be shown.

9.2.2 Relevance to CIM Reference Model

The *Common Information Model* (CIM) [56] provides a common definition of management information for systems, networks, applications and services, although currently it does not include reservation concepts. Nevertheless, the RE definition provided above is compatible with CIM as herein described (in what follows, the *CIM_* prefix of CIM classes is truncated). The RE implements `System`, as it defines a *functional whole*. Resource groups implement the `LogicalDevice` class, which is meant to provide an abstract model of hardware entities. In the case of physical resources that are sensors, the `Sensor` subclass of `LogicalDevice` can be used. Other resources implement `PhysicalElement`, defined in CIM as "any component of a System that has a distinct physical identity".

Through the `SystemDevice` relationship, `LogicalDevice` instances can belong to a `System`. The many-to-many relationship defined between `PhysicalElement` and `LogicalDevice` models the case where one physical resource belongs to multiple groups at a time.

The `LogicalDevice` class exposes properties that can be used for discovery and matchmaking. This allows the construction of complex expressions for resource selection and reservation based on group type, or on group properties (for instance "location", or "CPU clock speed").

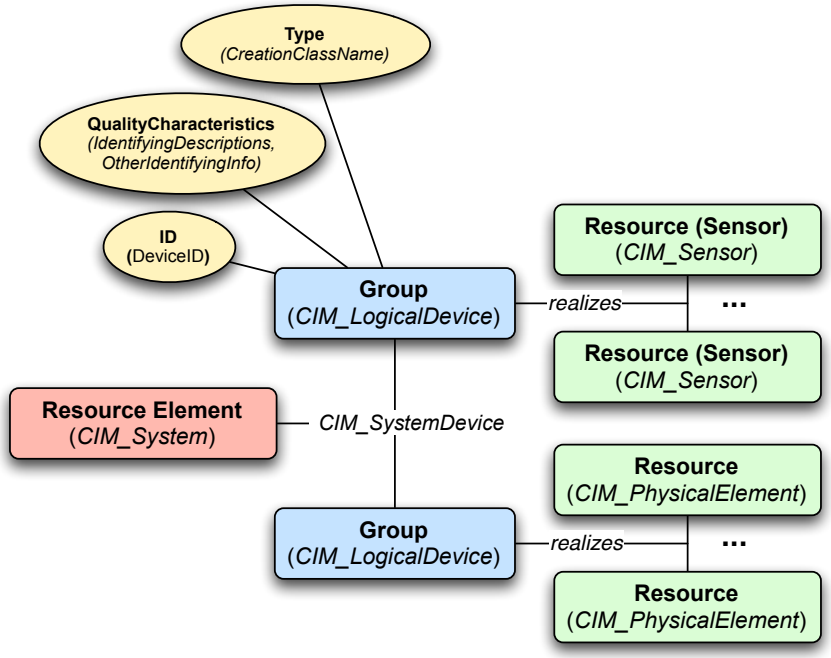

Figure 9.1: Mapping of RE concepts to CIM

The relevant attributes of `LogicalDevice` are `DeviceID`, which identifies groups uniquely, and `IdentifyingDescriptions`/`OtherIdentifyingInfo`, which are free-form and may be used for any kind of such characteristics. The extension of the `LogicalDevice` class and the usage of the `CreationClassName` property offer an additional easy way to implement group types.

9.2.3 Relevance to GLUE Schema

As mentioned earlier, the less frequently changing attributes of resource representations are expected to be available for resource selection and matchmaking. To achieve this, an appropriate *information model* is necessary for representation purposes. The *Grid Laboratory Uniform Environment* (GLUE) Schema by the respective Open Grid Forum working group [86] addresses this requirement for computing and storage resources, as it already provides representations for them. The abstract resource virtualization design presented here is compatible with the essential GLUE concepts, therefore the two can be combined and a normative description in the GLUE framework can be produced straightforwardly. According to the the GLUE 2.0 specification, five core concepts

lie in the heart of the model and can be directly mapped onto the design presented here. The Resource Element itself is the `Service`. Its interface to the SAMI, offering implementation information and normative access methods, is the `Endpoint`. Single resources themselves are the `Resources`; and resource groups are the `Shares`. Finally, GLUE `Activities` refer to control operations taking place on physical resources. This design does not pose any restrictions to these operations, therefore they can be freely described in GLUE terms.

9.3 Resource Element Reservations

Establishment of infrastructure SLAs involves a number of players; using WS-Agreement terminology, they are the *Agreement Initiator* (AI), the *Agreement Responder* (AR) and the RE. Practically, a request to establish an SLA is – in the context of IaaS – primarily a request to provision resources. As such, the reservation of resources will be required at some point, be it during negotiation (tentatively) or after the establishment (permanently for the contract duration). The main scenario addressed in this Part of the thesis can be as illustrated in Figure 9.2. As evident there, a customer SAMI (AI) negotiates with the IaaS provider SAMI (AR) for resources provided by the resource pools (RE). Thus, reservation operations of the RE must be invoked by the AR, via the provisioning module.

The data structures necessary to support RE reservation functionality are presented below.

Time Lease: this type is used to define the temporal creation constraints of reservations. As such, it is required for requesting a reservation, for verifying the availability of a resource group at a certain point in time, or for retrieving reservation metadata. The type consists of a start time expressed as a timestamp, and the guaranteed duration of the reservation. The start time may be zero, indicating an immediate reservation.

Group Types and Characteristics: these are simple structures consisting of an identification number and a textual description. They provide group-specific information that can allow for logical grouping in the process of matchmaking for a reservation request. Such types and characteristics may enable the construction of complex constraint expressions, such as “reserve 10 CPUs where clock speed is 2GHz and location is Germany”.

Resource Group: the group data type includes a numerical identification, a textual description, the group’s type and a list of characteris-

tics associated with it. As any domain-specific group-related data is decoupled from the group identifier itself, the design can be flexibly used in very diverse application environments.

Reservation: this type represents the reservation of one or more groups and includes all related metadata. It contains an identification number, a timestamp indicating the date and time for reservation set-up, its duration, the list of the relevant groups reserved, a boolean attribute to flag the state of the booking operation for all groups, and, finally, the reservation state itself. The state may indicate that the agreement is established but not activated yet; alternatively, that it is active and ready to be used by the authorized consumers; or that is has expired.

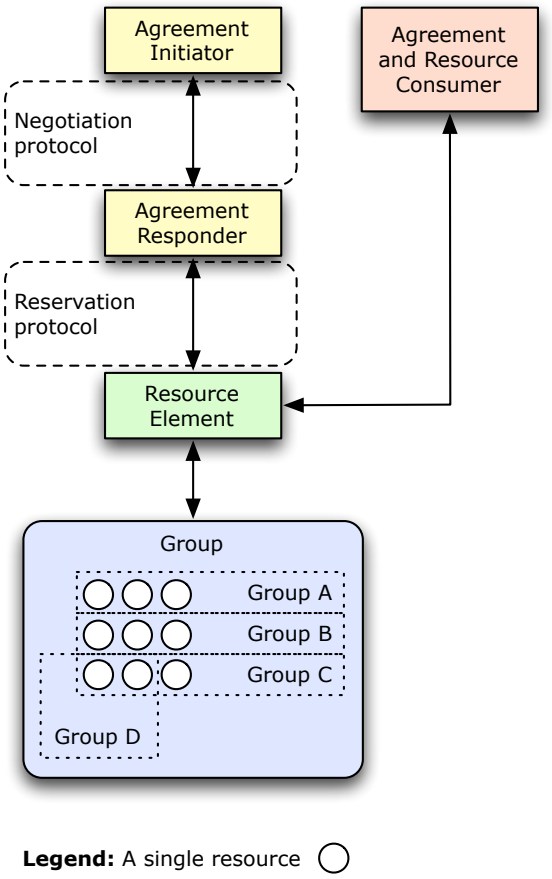

Figure 9.2: Resource Reservation Scenario

It is very important to note that the model allows both static (predefined) and dynamic groups. In the latter case, it is possible that a group

contains other groups recursively, although this is purely a conceptual artifact. What is imposed by the data type is that one single specific resource may belong to more than one groups. Thus, if resources R_1, ..., R_n all belong to a group S_1, and there exists a group S_2 which contains all of R_1, ..., R_n but also R_{n+1}, ..., R_m, then one can conceptually represent S_1 to be contained within S_2.

The following operations are defined for RE reservation management, according to requirements from Section 9.1:

Reserve groups By ID: Assuming that the IDs of the resource groups to reserve are known in advance, the SAMI may use this operation to invoke a reservation request on the receiving RE.

Reserve Groups By Expression: This operation enables the use of expressions including resource quantity, group type attributes and characteristics of a group as reservation constraints. The expression predicates may be combined using unary or binary boolean operators to specify dynamically the groups to be reserved.

Retrieve Reservation By ID: This operation returns metadata about a specific reservation, using its identification number.

Retrieve All Reservations: This operation is meant to return all the reservations for a Resource Element, or, with appropriate factoring of the input argument, all reservations in a specific state (e.g. expired).

Update Reservation: Reservation properties are modified by this operation using information such as time lease, the list of groups to be added or subtracted, new types and qualitative characteristics.

Cancel Reservation: It removes an existing reservation, as long as it has not expired. The policy to adopt in case of active reservations is a service-provider specific issue.

9.4 Application to Real Use Cases

The model and reservation language presented can have practical applications when combined with ontologies which describe the systems (equipment) that will be modeled and reserved. Such ontologies can be built using various sources, depending on the specific resources. Here, for illustration purposes, it will be shown how the model applies to a resource pool including storage, CPUs (processing) and network traffic sensors, as

well as to a real-world scientific instrument. The former could be, for instance, part of the infrastructure supporting the geo-location service discussed so far in the thesis. Additionally, the structure of a reservation request and the corresponding operation in a simple pseudo-language will be illustrated. Storage supports the types found in the *Storage Resource Manager* (SRM) specification [87] and a location characteristic. The instrument is a weather station coming from a SensorML [88] example, and including a thermometer, a barometer, an anemometer and a rain gauge. The thermometer has a resolution of 0.1 degrees Celsius, a range of -45 to +60 degrees, and its accuracy is $\pm$0.5 degrees. In the simplest scenario, the computing resources would be as illustrated in Figure 9.3, while the instrument would be represented as shown in Figure 9.4.

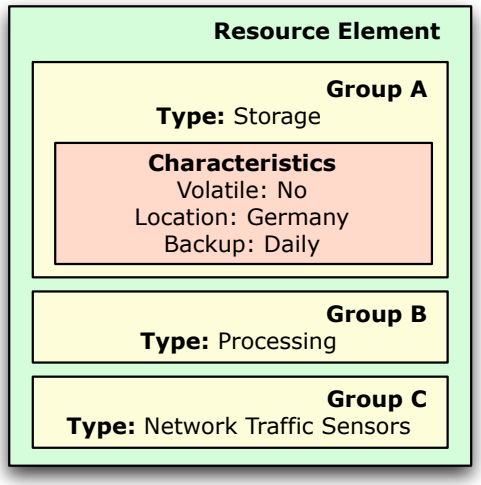

Figure 9.3: Representation of the computing resources

For the computing resources, a sample reservation request could look in pseudocode as follows:

```
Reserve Groups by Expression:
  Time Lease:
    Start: 2010-06-01 13:00:00 GMT
    Guaranteed Duration: 3600 seconds
    Desired Duration: 7200 seconds
  Type: Storage
  Characteristics:
    Storage: Volatile
    Location: Germany
    Quantity: 1TB
```

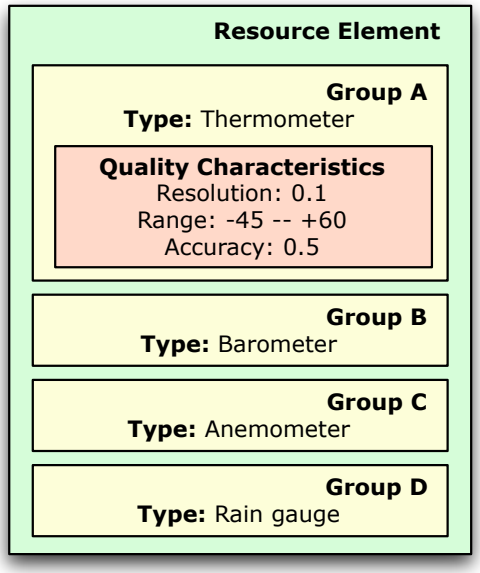

Figure 9.4: Representation of weather station

Making the weather station example slightly more complex, it is assumed that it has two thermometers; one with the characteristics mentioned earlier, and another one with more accurate characteristics, e.g. resolution equal to 0.01 degrees, range -65 to +70, and accuracy 0.05 degrees. A request to reserve this latter sensor for measurements would look in pseudocode as follows:

```
Reserve Groups by Expression:
  Time Lease:
    Start: 2010-06-01 13:00:00 GMT
    Guaranteed Duration: 3600 seconds
    Desired Duration: 7200 seconds
  Type: Thermometer
  Characteristics:
    Accuracy: 0.05
```

In the case where dynamic groups would be used, a group containing the more accurate thermometer would be automatically created and reserved. The initial resource representation would only include the groups originally defined by the resource owner (IaaS provider). Then, as reservation requests would come in and be accepted, additionally created groups could enrich the group ecosystem of the RE for all users, or remain private for the authenticated user who created them, depending on

implementation choices.

Similarly, for more traditional resources such as CPUs or storage, the use of dynamic group creation and deletion is necessary as a means to group resources on-the-fly according to incoming requests on one hand, and existing availability on the other. By way of an example, one can consider a case of a RE with a single group of 100 CPUs, all of the same characteristics. A reservation requests for a group that includes 50 CPUs of those same characteristics, would allow the RE to randomly select 50 out of 100, group them under a single reference endpoint, and return that to be used in all further communication with the requesting SAMI. The remaining 50 CPUs would form a new group to denote available (free) resources, but on the same time the original group would remain to denote the complete resource set.

9.5 Related Work

As discussed in Paragraph 9.2.3, GLUE is a model to describe computational resources – especially in the context of Grid computing. The model presented here is compatible with GLUE, but more generic, so as to allow the modeling of infrastructure resources other than compute units and storage. This abstract approach is necessary to describe, for instance, sensors, or scientific instruments. In this latter case, the types of the resource could be anything: “cameras” for telescopes, “pressure sensors” or “temperature sensors” in meteorological stations, “network interfaces” in active networking equipment, “motors” in robotic equipment, etc. The diversity of the different instrument types that comprise certain apparatus can be too large to express with a reduced dictionary. Similarly, the quality and suitability of a specific instrument can be expressed in many different ways. Examples include error rate, accuracy, response time, availability. However, it must be pointed out that different devices may have completely different metrics to indicate their appropriateness for a purpose, therefore also their suitability to a specific use case: Field of view (lens width), for cameras; Resolution, for visualization devices; Throughput, for a hardware encryption device; and so on.

The same differentiation applies for storage: Advance reservation of storage resources has been specified by the *Storage Resource Management Working Group* (SRM-WG) of the Open Grid Forum. SRM reservation operations are storage-specific as they include a number of functional attributes which are of particular interest to this application area.

In [89], Czajkowski et al. define a protocol for SLA negotiation in the context of Grid computing. Within this definition there exists a resource

description language, which is largely equivalent to the one presented in this Chapter. One difference between the two, is the additional semantics in the present work for resource clustering in virtual, dynamic groups. In addition, here there were presented advance reservation primitives, which were not within the scope of [89].

In [90], Kee et al. describe *vgDL*, the "Virtual Grid Description Language" that serves similar purposes. Nevertheless, the language is specifically targeting the Grid computing area, and as such, it is not clear how easily the language can be applied to different environments. Conversely, the language presented in this Chapter remains abstract on purpose, and therefore application in diverse domains is relatively straightforward.

Some very recent work that took place in parallel with the one described herein, is [91] by Koslovski et al. The authors present VXDL, a language to describe virtual resources and interconnection networks. Targeting a similar area, but not providing similar advance reservation primitives as in this Chapter, the aforementioned paper provides the definition of a more well-specified language (i.e. with specific syntax) that covers both traditional infrastructure resources (compute units, storage, network elements/circuits) but also acquisition and visualization devices, software, etc. In addition it makes possible to define resource groups, as is also the case with the work described here. Compatibility with other efforts and standards is not discussed.

Summary and Conclusions

In this chapter, a resource description model was defined, alongside primitives for advance reservation of such resources. The goal of this work was to complement the SAMI architecture and allow access to resources in parallel to SLA negotiations. The contribution by this work can be summarized as follows:

- The model is extensible and can be used in very diverse use cases, to describe arbitrary kinds of infrastructure;
- It is compatible at large with standards such as CIM and GLUE;
- Given a machine-readable rendering, the model can be used together with envelope negotiation protocols and models (such as, for example, WS-Agreement);
- It is coupled with generic constructs for reserving resources in advance;

- It supports clustering of resources in virtual groups; and
- It is not tied to specific technologies (such as XML), rather it can be adapted through respective renderings.

In the upcoming Chapter, two greedy resource planning algorithms will be examined in the context of a hierarchy of SAMIs. The concepts from Part II, together with the concepts from this Chapter, will be applied to the use case of Infrastructure as a Service as an example of application.

Chapter 10

IaaS SLA Planning

As mentioned earlier in the thesis, this part comprises an *evaluation* of the concepts mentioned thus far to show how hierarchical SLA management can be achieved. In this Chapter and the next one, SLAs and their hierarchies will be used as a means for infrastructure resource capacity management in the context of IaaS.

More specifically, in this chapter a scenario is sketched and the whole setup for IaaS SLA management is illustrated. A simulation is designed as proof of concept based on this scenario, putting the focus on normal, runtime operations. There, SLA requests arrive from customers, they are being negotiated, served, and expire. To this extent, the Chapter serves as proof that the proposed design and models for hierarchical SLA management are feasible.

As discussed earlier, the SLA planning methodologies used during negotiation are considered to be use-case specific. Here, two different greedy scheduling algorithms are used for this purpose. The algorithms are modeled after the *online bin-packing* problem, and they are evaluated (via the aforementioned simulation) with regard to their energy efficiency, performance, and resource utilization efficiency.

10.1 Scenario Description

The scenario assumes customers (which *may* be represented by a SAMI themselves) who contact the provider to ask for the establishment of one or more SLAs, where the service involves infrastructure resources. The provider has a top-level negotiating SAMI, without any attached resources. The latter are distributed among a number of data centers, and resource pools contained in those data centers. For each data center, there is a SAMI that negotiates for the respective resources, and a Re-

source Manager (abstracted by the Resource Element model) that manages them. An SLA of the customer with the provider's SAMI implies there is at least one, or perhaps more, SLAs between the provider's SAMI and the data center SAMIs. For reasons of convenience, but without loss of generality, the assumption here is that each SLA consists of one co-location unit – that is, all resources for that SLA must reside in the same data center. Nevertheless, it is straightforward to have more than one co-location units inside an agreement, and assign each one of them to a single data center. This applies as long as it is possible to associate each co-location unit with a specific price and penalty.

An incoming SLA request, such as the ones used for the simulation discussed later in this Chapter, looks like the example from Figure 10.1. There it is assumed that the constructs offered by the negotiation protocol's envelope are used to assign temporal, price, and penalty parameters. That said, there is nothing that prohibits to use the resource request (RE) structure for the same purpose; the SAMI would then have to remove those group characteristics that cannot be understood by the resource manager. This would then allow to have a single resource request – with multiple resource groups – inside the SLA request.

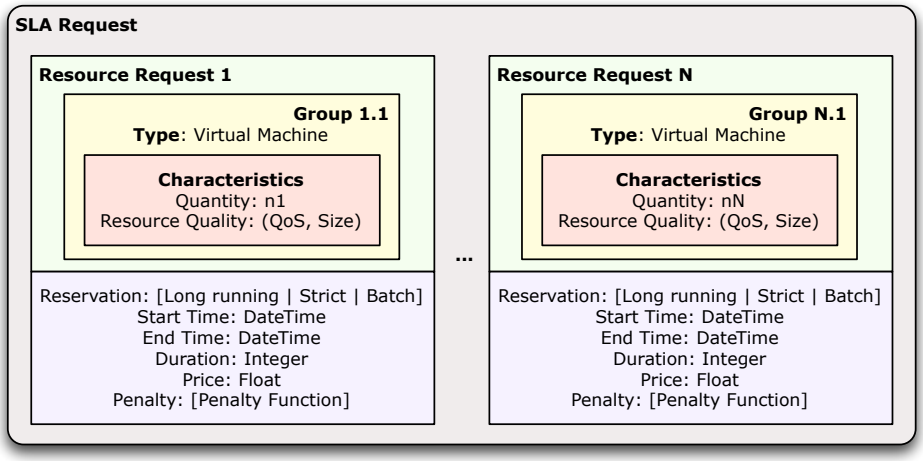

Figure 10.1: An SLA request example

Incoming SLA requests were produced at random for the simulation. It should be noted that these requests contained no boolean operators, and therefore no BDD construction or path selection was necessary. This approach was chosen to reduce less-relevant complexity, but also because the performance of constructing BDDs for random boolean functions is out of scope for the present work (readers interested in the topic can

study [92] by Wegener and [93] by Gröpl). The specific choice does not affect generality; should there be more than one alternatives (paths ending to node **1**), the one with the highest returns could be chosen – and if it could not be satisfied due to lack of resources, the next one(s) would be tested.

The resources involved in each SLA are *Virtual Machines* (VMs), all characterized as "gold", "silver" and "bronze" to illustrate different quality of service (CPU clock speed, amount of memory, amount of storage). Each QoS level also has two categories ("small" and "large"), resulting in the possible requested configurations from Table 10.1. They are assigned in servers of the same QoS class, to reflect the correct clock speed, but also the correspondingly larger total amounts of memory and storage (Table 10.2). All memory and storage amounts are counted in GB. Each resource manager (data center) has three resource pools, one for each server type, each pool featuring 5000 servers. There are three REs in total, each represented by one SAMI. Figure 10.2 illustrates the described setup.

QoS class	Category	Clock speed	Memory	Storage
Bronze	Small	1.6	1	80
Bronze	Large	1.6	2	140
Silver	Small	2.0	2	160
Silver	Large	2.0	4	280
Gold	Small	2.4	4	320
Gold	Large	2.4	8	560

Table 10.1: Virtual machine types

QoS class	Clock speed	Num. Cores	Memory	Storage
Bronze	1.6	4	8	640
Silver	2.0	8	32	2560
Gold	2.4	16	128	10240

Table 10.2: Server types

A customer must provide, as part of the request, the respective details of each resource group that must be allocated. These details include, for each resource group, the following information:

1. *Reservation type:* May be one of `long-running`, `strict`, and `batch`. *Long-running* indicates that the point in time when the allocation should be de-activated (i.e. when the lease will expire) is not known;

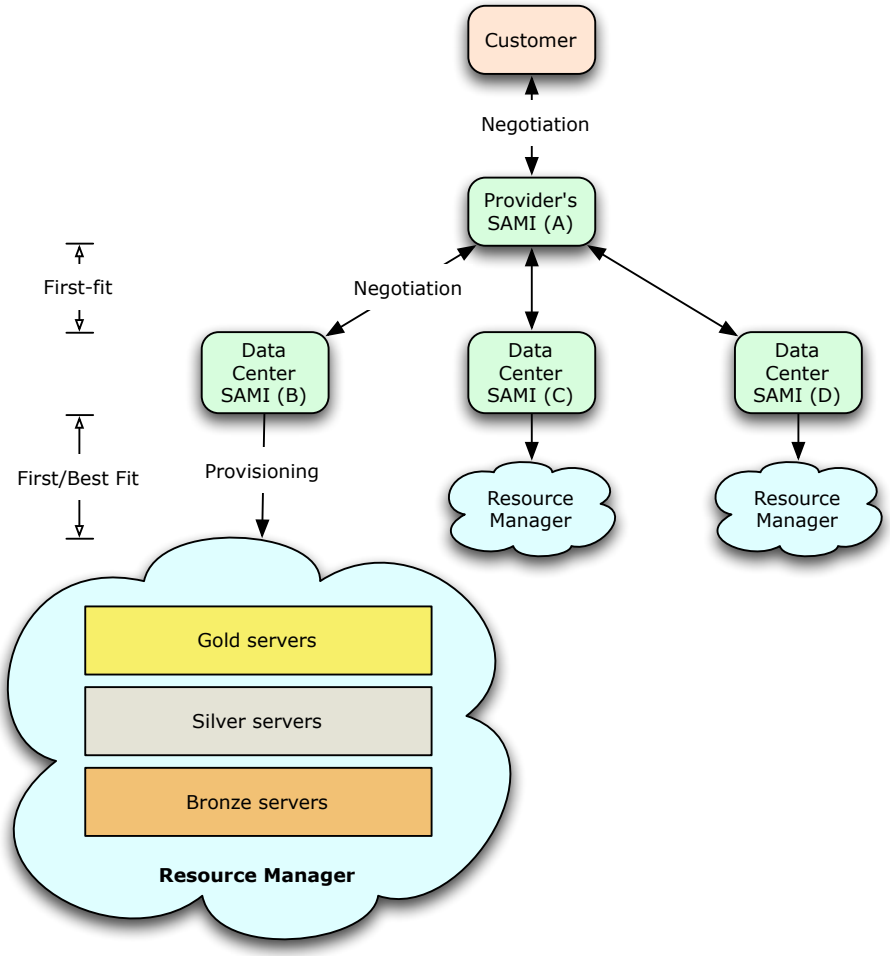

Figure 10.2: The IaaS scenario of Part III

the resources would be constantly needed for the foreseeable future. *Strict* means that the given time limits (as discussed below) are strict and should be honored, otherwise the SLA will be considered to be violated. Finally, *batch* means that the resources must be allocated to the user at some point between the declared time limits, for a duration also declared, with the duration being shorter than the given time frame. This basically means that the provider is free to schedule those resources for some point in time when it will be most convenient; the user does not care exactly when that will be, as long as it is for the given duration. Should the total duration be less than the requested one, penalties will apply.

2. *Resource quantity:* The amount of requested resources, in some standard unit.

3. *Resource quality:* Combination of size category (large, small) and quality (gold, silver, bronze).

4. *Start time:* When the resources should become available (or earliest time limit for batch resource groups).

5. *End time:* When the resource can be de-commissioned (may be a value implying "never" for long-running groups, or a value implying the latest acceptable point in time for batch groups).

6. *Duration:* For how long the resources must be available. May be irrelevant for long-running groups, equal to the given time frame for strict groups, and shorter than the given time frame for batch groups.

7. *Penalty:* A percentage of the group's cost, depending on failure ratio, as discussed in Chapter 7. The failure ratio is proportional to the amount of time that the group's resources are unavailable.

A price per resource group is also assigned to each of them. It is the product of a random value, the group's resource quantity, and its duration.

10.2 Algorithms

As also illustrated in Figure 10.2, the assignment to a data center is always made (in this scenario) on a First-Fit basis. In the next Chapter, different strategies for the negotiation among the provider SAMI and the data center SAMIs will be explored.

The provisioning strategy, however, may take different forms. The scenario is modeled as a *dynamic, multi-dimensional, online bin-packing problem*. A bin-packing problem is the (combinatorial) problem that tries to minimize the number of *bins* into which are placed N given items of different sizes, and all items are known in advance. Its *online* counterpart is the same problem, but the items are not known in advance; rather, every time a new item m is given, $\mathrm{m} < \mathrm{N}$, the remaining items ($\mathrm{m} + 1..\mathrm{N}$) are unknown. The multiple dimensions are related to the different types of resources required per item (CPUs, memory, storage). Finally, the problem is characterized as *dynamic* due to the introduction of a *time dimension*, and the fact that items may also leave a bin – not only join it.

Figure 10.3 illustrates the problem graphically. Each box without a label represents a request for a VM, within the incoming SLA offer. Its horizontal side represents the amount of memory that the VM should feature, and the vertical side represents the respective amount of storage. The color shows what is the required clock speed (and, therefore, in which server pool it should be assigned). The problem faced is how to assign all VM requests and minimize the number of (active) servers that have one or more VMs assigned to them. That means, unused memory and storage should be minimized for the active servers, while in parallel, the number of inactive servers (dark-colored ones) should be maximized. Figure 10.4 illustrates the time dimension of the problem (excluding the "storage" dimension and keeping only the one about memory, to simplify representation). Requests A – E are being agreed to as they come. The height of each box represents the requested memory, while the width represents the duration of the advance lease. Location on the time axis represents the exact point in time when the lease starts. When SLA request F arrives, it cannot be placed into Server 1, as there is not enough memory left available from point in time T1 to point in time T2. Server 2 must then be started, and the request should be assigned on it.

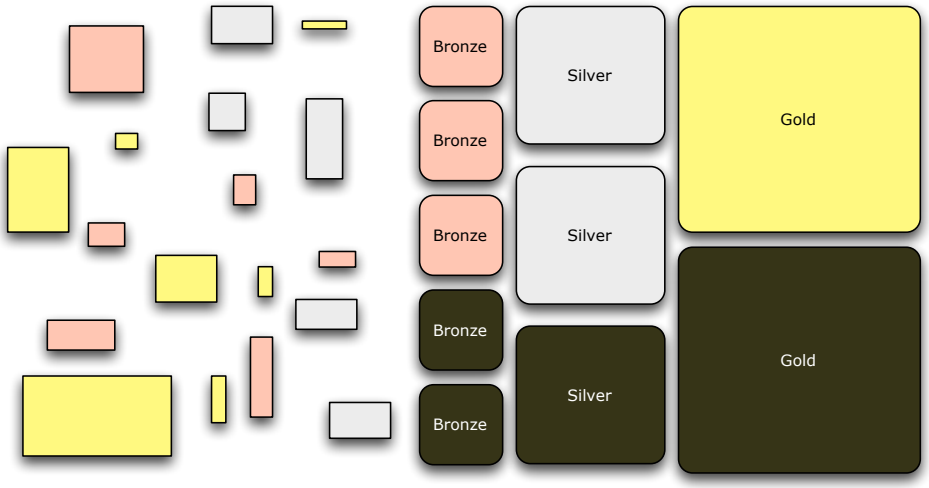

Figure 10.3: Two-dimensional bin (server) packing

The reason that the formalization of bin-packing is chosen, is the wish to minimize the number of *active servers*, and thus explore the efficiency of different approaches with regard to energy savings. Although bin-packing is one of the oldest combinatorial problems, and there exist various different formulations (online/offline, dynamic/static, single-/multi-dimensional), the combination of all three properties has not been stud-

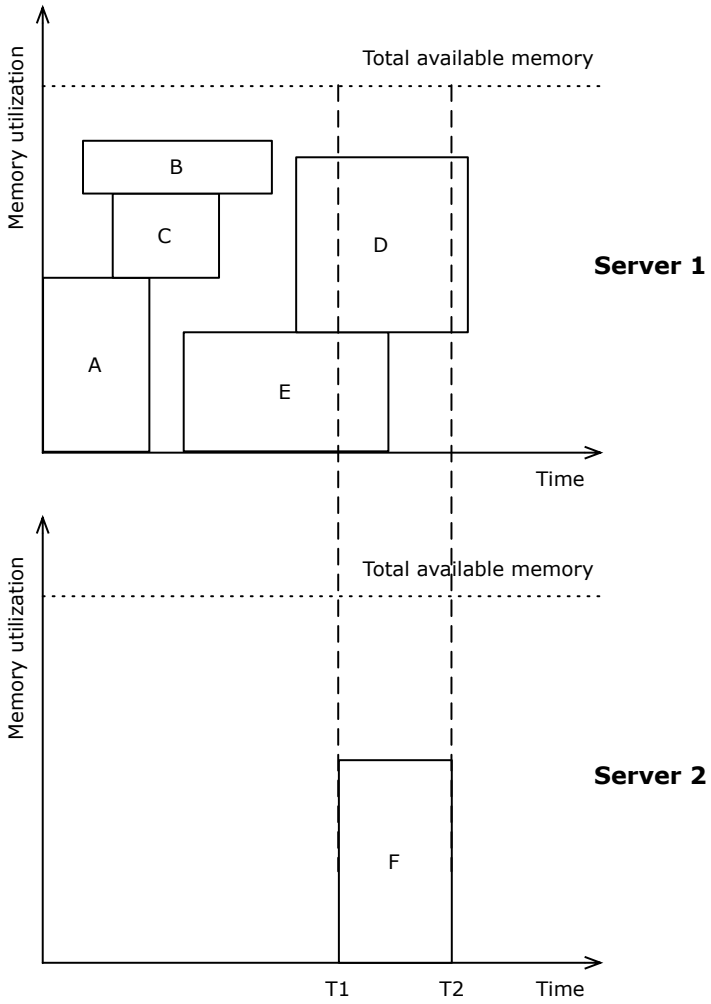

Figure 10.4: Dynamic bin (server) packing

ied until very recently. To the best of the author's knowledge, the only other work that addresses this specific problem is [94] by Epstein and Levy, under publication at the time of writing (September 2010). This paper studies the problem in a more complete manner, and therefore readers interested in its properties are recommended to advise with it. In the present Chapter the requirement is to apply the problem, as an appropriate formulation, and explore the efficiency of two relatively simple solving algorithms. It should also be noted that in the traditional dynamic variant of the online bin-packing problem, the events of an item leaving the bins is not known in advance. In the case studied here, it may be known in advance when an item will leave the bins, as it reflects the du-

ration of the reservation (declared by the customer at negotiation time, in the case of strict and batch reservations).

The strategies explored are a *First-Fit* (FF) and a *Best-Fit* (BF) greedy algorithm. The FF algorithm is going through the list of *open bins* (candidate servers) for each VM group requested in an SLA, until the first server with sufficient resources is found for the time frame of the group and some part of its total quantity; when this happens, the specific VM group's quantity is reduced by the respective amount that was assigned to the server, and the search continues in the same manner until the full group quantity has been assigned. Figure 10.5 illustrates the algorithm executed for every resource group in the request, in the form of a flow chart.

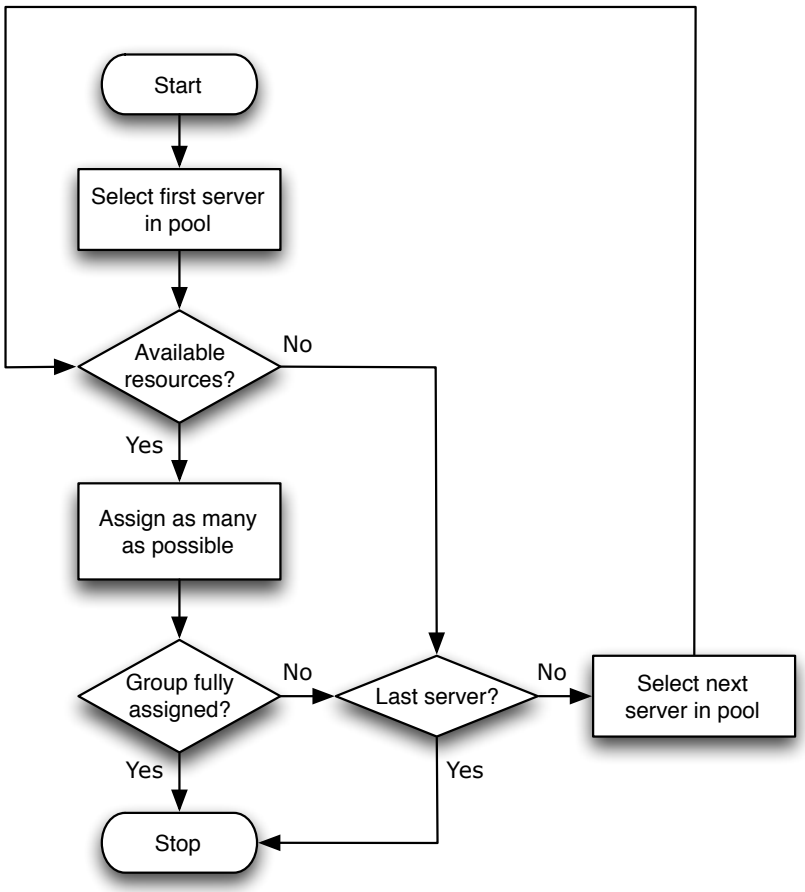

Figure 10.5: First-Fit assignment algorithm for each resource group

The Best-Fit algorithm is similar, but instead of selecting the first server with available resources, it tries to find the one that will minimize resource

waste (in the form of spare cores, or memory/storage residue). Naturally, this means that BF has to run through the server list multiple times – up to s^2, s being the total number of servers in the pool. Therefore, its performance is expected to be much worse than FF; this is confirmed in Section 10.3 via simulation. The Best-Fit algorithm used is illustrated in Figure 10.6.

10.3 Experimental Evaluation

The simulation was executed for three different arrival rates and resource demand scenarios: One where resources were always enough for both algorithms (low consumption), a "borderline" one where some SLA requests would start being dropped, and one where resource demand was very high and the pools could not keep up with it. Results were averaged across the three data centers, for each case.

In all cases, execution speed of the Best-Fit algorithm was predictably much worse than First-Fit. This is illustrated in Figure 10.7. It is immediately evident that BF is significantly slower than FF. Nevertheless, this delay should be assessed in relation to potential advantages in SLA acceptance rate and energy savings.

In the first case, as designed, all SLAs were accepted by both strategies. Looking at energy savings, measured by means of total servers switched on, Figure 10.8 shows that the Best-Fit algorithm performs better to some small extent. Zooming in to the graph, Figure 10.9 shows that the difference is in the scale of 2%. In general, the better performance of the BF strategy is expected: It it tuned to avoid resource waste, and therefore, it always first tries to maximize utilization for the servers that have *some* resources available. Therefore, it fills utilization gaps, and is slower to try switch on the next available server.

Switching to a larger arrival rate, Figure 10.10 shows that BF accepts all SLA requests, while FF already starts dropping some.

In this case, however, energy savings are much more using a BF strategy, and can be seen to approach 5% - 6%. This is illustrated in Figures 10.11 and 10.12.

When moving to the very large arrival rate, where in both cases not all incoming SLA requests can be served, these energy saving difference is evened out as one would expect. All servers must be utilized, and the advantage of the BF strategy now is that it can serve more SLA requests, as it has been making better use of the available resources. Figure 10.13 illustrates this better rate of accepted SLAs, which at times reaches a gap of approximately 10%. Finally, Figure 10.14 shows that all servers

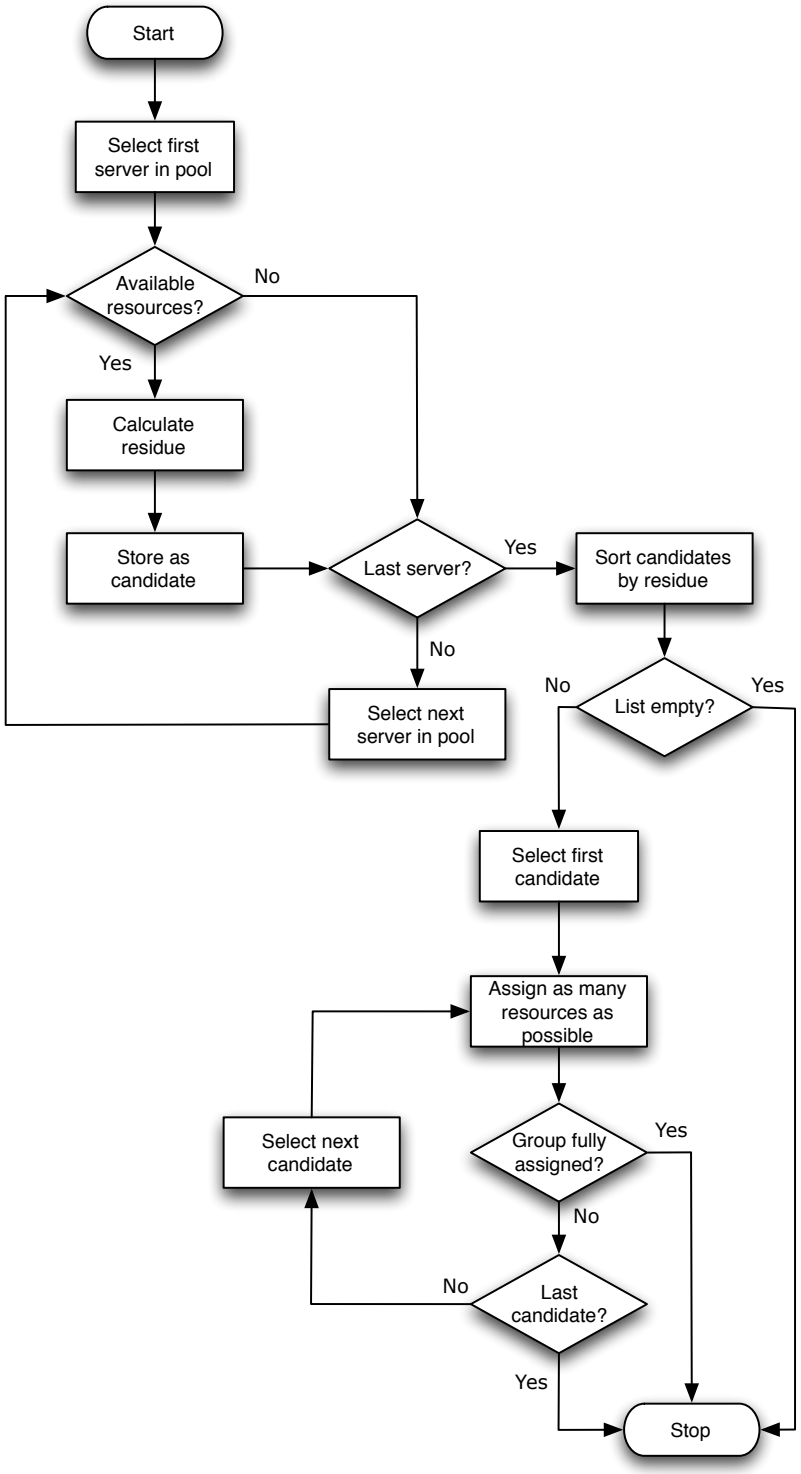

Figure 10.6: Best-Fit assignment algorithm for each resource group

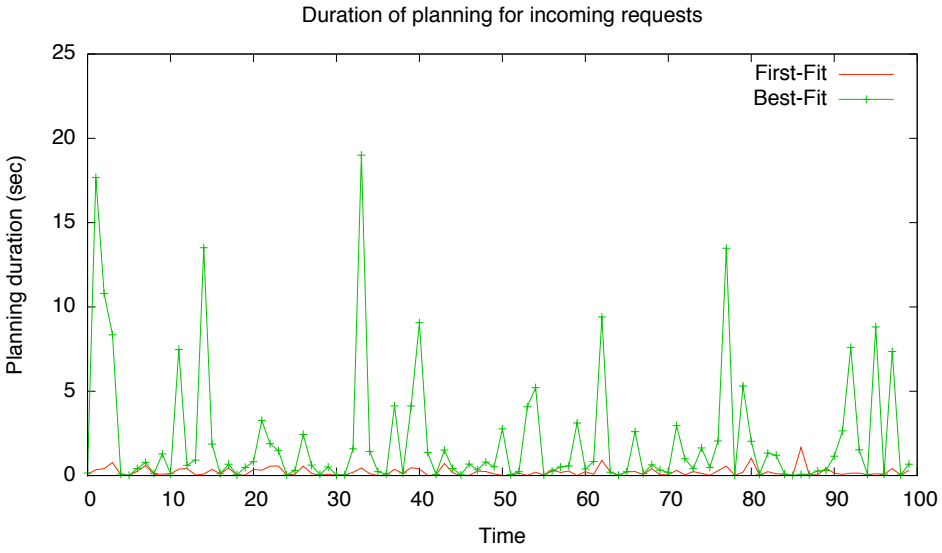

Figure 10.7: Performance comparison of FF and BF

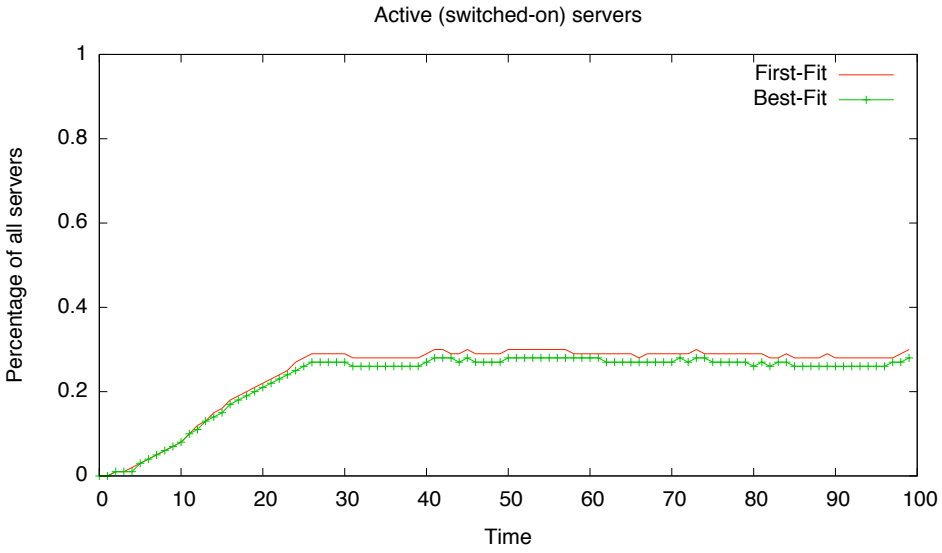

Figure 10.8: Energy savings comparison (low arrival rate)

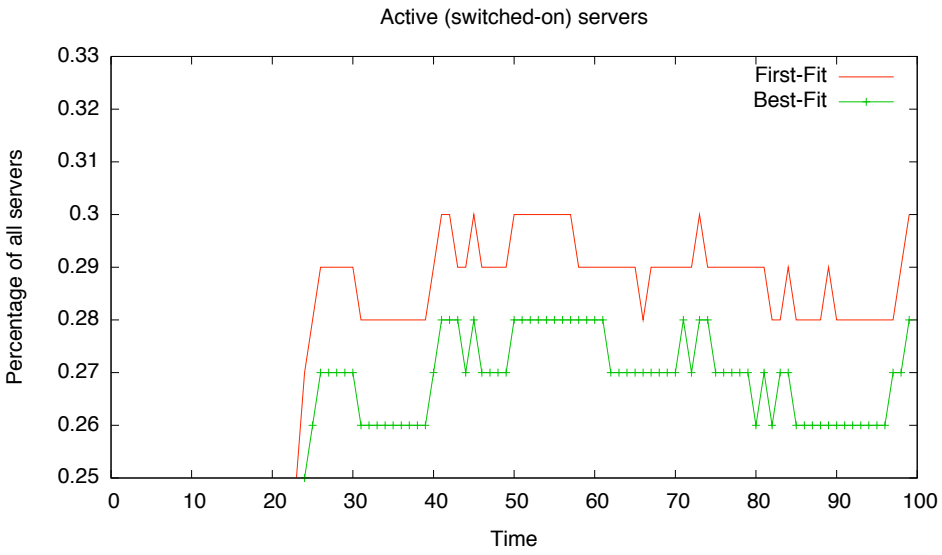

Figure 10.9: Energy savings comparison, zoom in (low arrival rate)

are fully utilized, and the advantage of the BF strategy has vanished as expected.

Via all the experiments run, it is evident that the use of a Best-Fit strategy that tries to minimize resource waste within a single data center, has significant advantages when aggregated over the whole infrastructure of a provider. Energy savings varied and reached a difference of approximately 6%; and accepted SLA requests (which can be directly reflected on additional customers and therefore additional revenue) was also significantly higher in the case of saturated infrastructure. These advantages should be balanced against the significantly higher planning time. The gap among the two varied, but was consistently high, in some cases more than an order of magnitude. That said, the total time wasn't prohibitively large, and in all cases remained under 20 seconds for our setup of three data centers, nine resource pools, and 5000 servers in each pool. Should this planning duration be acceptable, then the gains in using a Best-Fit strategy should not be overlooked.

10.4 Related Work

Variants of bin-packing have been studied extensively in the past. As mentioned in Section 10.2, Epstein and Levy very recently studied the complete variant we use here (albeit with unknown departure times for the items involved) in [94]. An overview of related *approximation* algo-

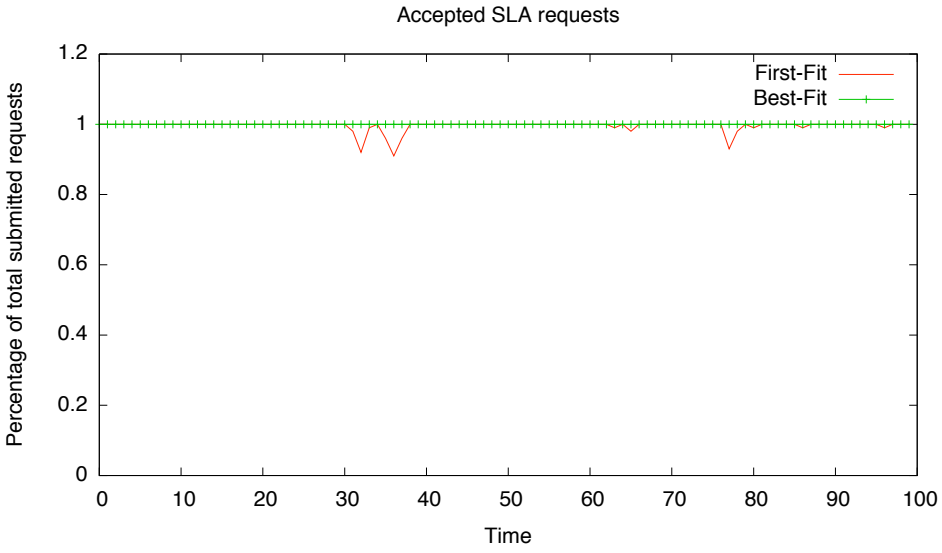

Figure 10.10: Accepted SLAs comparison (medium arrival rate)

Figure 10.11: Energy savings comparison (medium arrival rate)

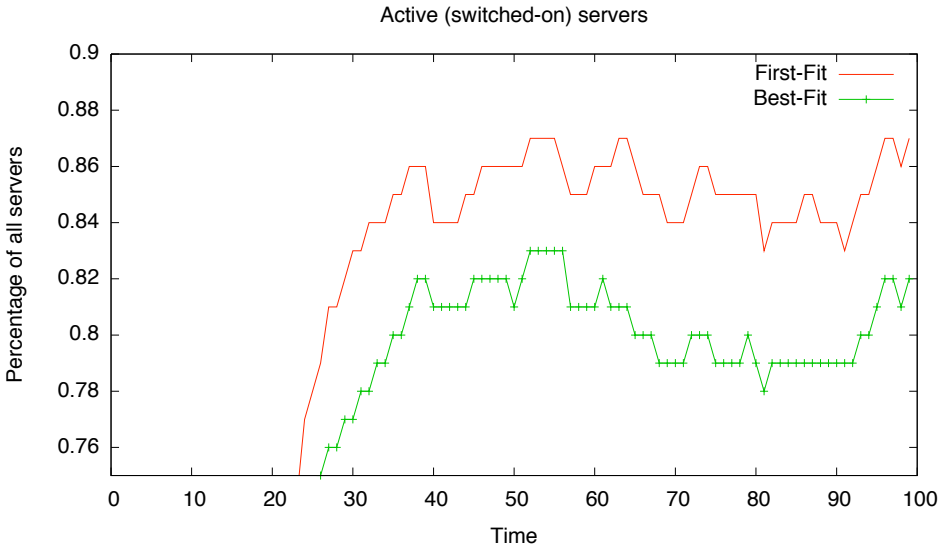

Figure 10.12: Energy savings comparison, zoom in (medium arrival rate)

Figure 10.13: Accepted SLAs comparison (large arrival rate)

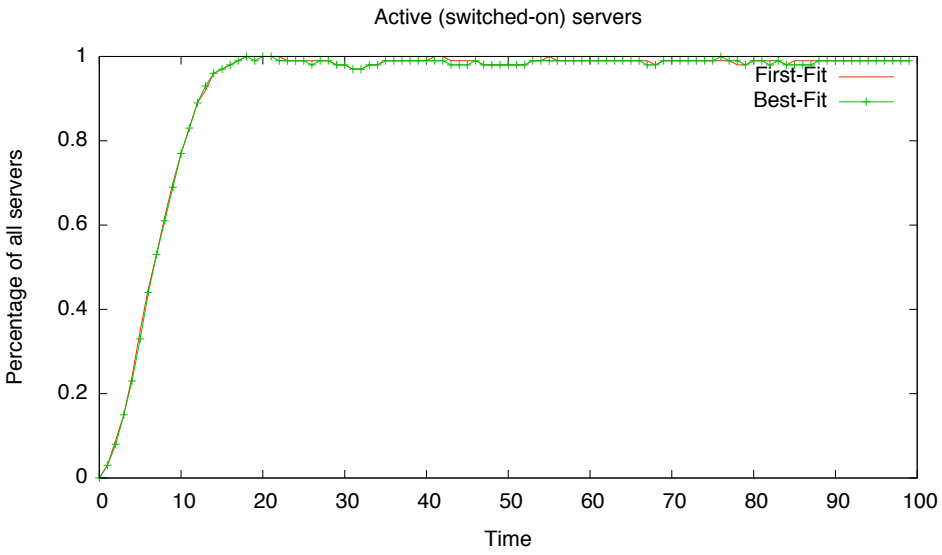

Figure 10.14: Energy savings comparison (large arrival rate)

rithms for other *online* variants can be found in [95] by T. Gonzalez (ed.) and [96] by Coffman et al.

In [97], Van et al. also address a VM packing problem aiming at minimizing active servers, but they approach it as an offline, combinatorial problem that they solve periodically. Also, VM migration is possible in their model, something not allowed in the scenario presented here. A similar approach is presented by Hermenier et al. in [98]. The authors discuss task placement in cluster systems, but assume a VM per task, making the problem largely equivalent.

In [99], Nakada et al. apply a genetic algorithm to plan efficient VM packing vectors, as an alternative to best-effort and VM migrations. Their work, however, is currently at the first stages and therefore it is difficult to extract conclusions with regard to its sufficiency or purpose fitness.

Ajiro and Tanaka present some similar work for server consolidation in [100]. They perform measurements using two different algorithms, a *First-Fit Decreasing* (FFD) one (which is equivalent to BF from this Chapter), and a *Least-Loaded* (LL) one where the load was assigned to the server with the most available resources. By definition for this Chapter's energy-saving objective, LL was inappropriate. Also, the authors do not address the time perspective.

As regards SLA-based resource scheduling, there has been quite some work in the area during the last few years. Most existing work, however, looks into application-specific SLAs (which may, for instance, address ap-

plication performance guarantees), while the present work considers only SLAs for infrastructure services that are application-agnostic.

In [101], Netto et al. concern mostly with advance reservation of resources. Similarly to the work presented in this chapter, SLAs may have some flexibility with regard to their start and end times. Although Netto et al. do not include long-running SLA requests, they have a type that has flexible start time and strict end time. The authors also assume that the user provides (as part of the SLA) a function that correlates execution time with number of resources; and that there is, as such, flexibility with regard to the amount of resources assigned to the user. Conversely, here it is assumed that the user requests a specific amount of resources for a specific duration, and the main concern is which resources to use so that the energy footprint is reduced.

In [102], Goiri et al. look at SLA-driven resource management in the context of Grid computing. As such, SLAs are related to specific tasks also in this case. The paper is related mostly to SLA adjustment issues: It presents a method to reschedule SLAs that are failing, by scaling either vertically or horizontally. These concepts and the architecture presented may be useful for the purpose of renegotiation and dynamic workloads. A very similar system is presented in [103] by Roy and Mukherjee. Using management agents in a Grid environment, the authors monitor tasks in relation to the respective SLAs, and perform adjustments when necessary.

Finally, in [104] Ardagna et al. present a solution to the problem of assigning multi-tier applications to servers within a data center. The multi-tier nature of those applications makes the problem quite similar to the topic of hierarchical SLAs, nevertheless the authors address it exclusively in the context of a single data center. Ardagna et al. construct a Mixed-Integer Non-linear Program, based on which they try to maximize revenue and minimize costs; the latter, in the form of reduced energy footprint. The problem is solved using a custom heuristic.

Summary and Conclusions

In this Chapter, there was presented a scenario that applies the main SLA management concepts from Part II on the domain of Infrastructure as a Service. Multiple SAMIs are assumed, in different management domains, serving incoming SLA requests. Based on those requests, which were dispatched from a central broker to the data centers, two different VM placement algorithms were evaluated as regards their performance and energy efficiency, in a simulated environment. The algorithms, First-Fit (FF) and

Best-Fit (BF), could be used as the POC implementations of the data center (2nd level) SAMIs. They exhibit a large difference in performance, with FF being faster. However, BF is significantly more efficient with regard to resource utilization, and energy savings by means of switched-off servers (no VMs assigned to them).

In the Chapter that follows, a different planning problem (yet, within the same setup) will be tackled. More specifically, the problem is to decide on the best course of action when suddenly resources become less than those needed to serve the already established SLAs. This may happen, for instance, in the case of massive resource failure. The objective of upcoming work is to minimize resulting penalties, and the approach used is to solve a knapsack-based combinatorial problem.

Chapter 11

Offline Planning with Limited Resources

After an SLA has been established, it must be executed and monitored. Failure to provide the necessary resources and implement the service will result in a penalty, according to the SLA's provisions.

In this Chapter, it is assumed that at some point in time the resources become drastically less than needed to serve the SLAs already established. This may be, for example, due to some massive resource failure. The provider at that point must make a choice, which SLAs to violate purposefully, and to what extent. This is different from the previous Chapter not only because of the extraordinary circumstances involved, but also because the planning involved is now *offline*: the provider knows all SLAs in advance, and can use the complete SLA set during this planning process.

11.1 Scenario Description

As an example of the situation to be examined in this Chapter, it will be assumed that at some point in time the normal operations (as described in Chapter 10) are disrupted because of one data center failing completely, e.g. due to power outage, network disruption, etc (Figure 11.1). It must be stressed that this is just an example; the approach is valid in any case where the existing available resources are less than those required by established SLAs. Another example of such an occasion could be the case where SLAs are not automatically negotiated among SAMIs, but rather inserted into the system by an administrator without prior confirmation of the existing resource capacity. Should resources become less than those necessary for the complete provisioning of all SLAs, some of them

will have to be withdrawn partially or completely.

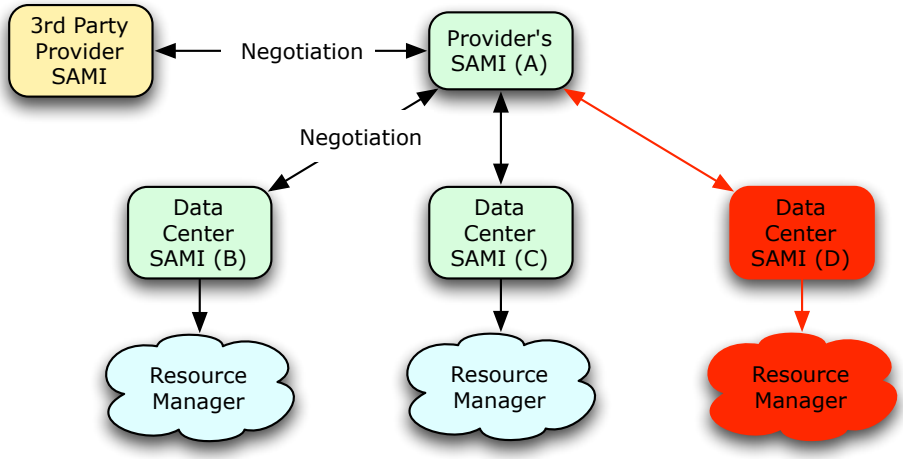

Figure 11.1: Resource failure scenario

This time, the problem is not to fill up as few servers as possible, like in Chapter 10. Rather, it is a question how to make the best possible use of all available resources – where "best" is associated to the least penalty. Figure 11.2 illustrates the concept in a simple manner. If the resource pool is as shown in this Figure, where the horizontal dimension represents time (lease start times and durations), the vertical axis represents the SLA(s) requested resource quantity; and the number in each SLA representation is the associated penalty if it is discarded; then the problem is which SLAs to choose and assign resources to, so that the penalties of the remaining SLAs (the ones "outside" the pool) are minimized.

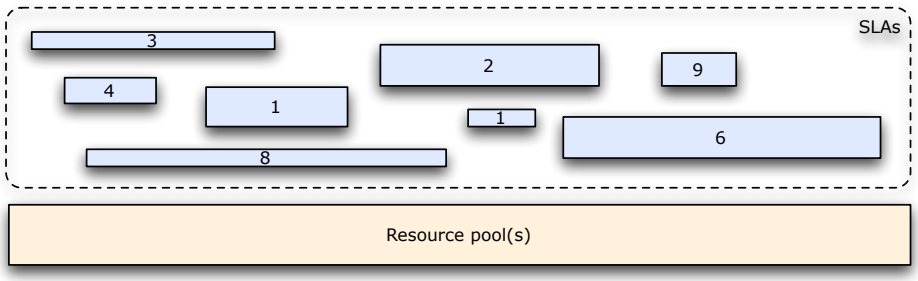

Figure 11.2: Problem illustration

In this case, we are evaluating five different planning strategies, for the interaction among the main provider's SAMI (A) and the data center SAMIs (B, C) or possibly a 3rd-party provider's one. It is important to note

that 3rd-party SAMIs and data center SAMIs are essentially treated in the same way. For the provisioning phase (i.e. the communication between data center SAMIs and REs), a simpler model has been chosen here to improve simulation speed: It is assumed that there are large resource pools with CPUs, memory and storage, instead of separate servers offering them as it was examined in Chapter 10. This does not affect the results, due to the fact that within a data center there is simply a check whether an SLA fits or not; and the result is the same, irrelevant of the strategy used each time. Thus, this simplification causes no loss of generality when it comes to the comparison of the five examined strategies.

The structure of an SLA is as discussed in the previous Chapter. It includes a single co-location unit, that is, a group of resources that must end up in the same data center. Each group contains subgroups (*resource requests*) of *the same* resource types, for instance "CPU cores of clock speed 1.6 GHz", and a requested quantity. For each resource request, there is also mentioned the type of the lease (long-running, strict, batch) and the respective time limits.

The planning strategies that will be evaluated are the following:

Non-flexible: The SLA is included/ scheduled in full, or not at all. That is, there is either no penalty, or a penalty equal to the full price of the SLA. Apparently this strategy is the least flexible, being almost "binary" in nature: All resources have to be provided, otherwise the full penalty applies. It represents resource requests where the various groups inside them are tightly coupled, and it is not useful to have some resources available, while others are not. The only element of flexibility is with regard to start / end time, for "batch" reservation types as discussed in Chapter 10. Even in that case, however, the total duration of the lease is strictly defined; if the resources are provided for less time than agreed, the full penalty applies. One can intuitively see that this strategy will be the fastest, as there are far less options to evaluate during the optimization phase.

Flexible-items: Some of the resource requests in the SLA may be accepted, while others not. The total penalty is the sum of penalties for each resource request that was not accepted. This strategy is similar to the previous one, with regard to the lack of flexibility regarding time. It is not possible to modify the start or end time of the associated resource leases, except for batch reservations where the duration must still remain the agreed one, or the full penalty will occur. Looking at Figure 11.3, it may be the case that resource request 1 is accepted and assigned, while resource request 2 is not. This could happen if, for instance, after calculating penalties for the two

requests according to the respective penalty functions, it turned out that the penalty for request 1 is relatively smaller than the one for 2, and therefore can be tolerated.

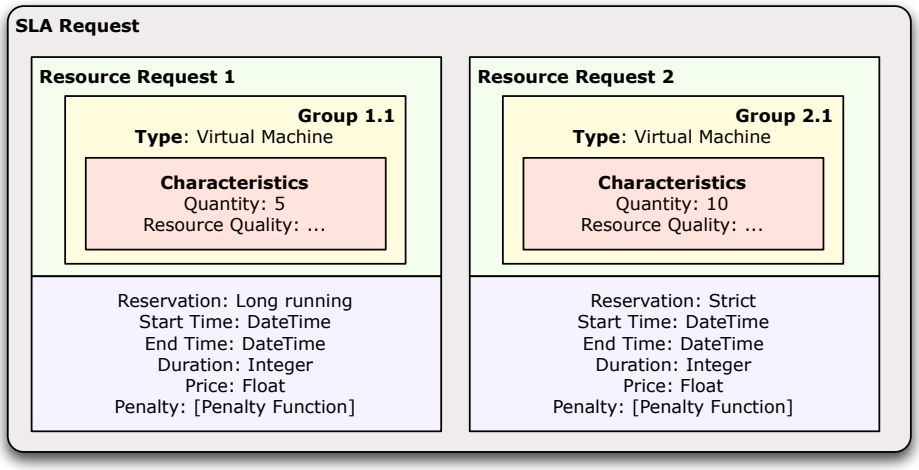

Figure 11.3: Example for “flexible items” strategy

Flexible-time: In an effort to maximize the number of accepted SLAs (i.e. minimize the number of customers that receive no service whatsoever), in this strategy there is given complete flexibility: Not only some resource requests of an SLA may be served while others not, but those that were chosen may be scheduled to start later than planned, end earlier (for “strict” leases) or have a duration shorter than requested (for “batch” leases). In the example from Figure 11.3, it may be the case that request 1 (featuring a long-running reservation) may start later than agreed; and 2 may start later, end earlier, or be rejected altogether.

First-Fit: This is a *greedy* algorithm, that sorts SLAs according to their penalties in decreasing order and tries to assign (serve) as many of them as possible. It follows an all-or-nothing approach, like the “Non-flexible” strategy. Each SLA is committed to the first data center in a row that will have enough resources (i.e., the first SAMI to accept it).

Best-Fit: Like First-Fit, but the SLA is committed to the SAMI that will report back the least available (free) resources after acceptance. As it cannot be realistically expected that 3rd-party SAMIs will report resource availability, this algorithm can be applied only within the same administrative domain – that is, within the same IaaS provider.

In all strategies, the objective is to pay the least penalties possible. The evaluation in Section 11.3 will assess them with regard to execution time, best achieved objective, how far customers are affected, etc.

11.2 Problem Models

The three combinatorial strategies (Non-flexible, Flexible-items, Flexible-time) will be formalized as binary (0-1) *Integer Linear Programming* (ILP) models. It is also possible, with some modifications, to formalize them as *Mixed-Integer Programs* (MIP) of quadratic form. Such models are simpler to express and more flexible with time. However, they are also known to be extremely difficult even for state-of-the art solvers and relatively small instances, therefore an ILP approach was preferred. As it will be shown later, even like this, the problems are fairly hard to solve for larger instances. In order to achieve the ILP formulation, time was quantized. The exact analogy of each simulated time slot to a real time duration is left open; it may be one minute, one hour, or anything else.

The problem is formulated as a knapsack problem [105] variant. More specifically, this is a *multi-period*, *multi-dimensional*, *multiple-knapsack* problem. In the traditional knapsack problem, the question is the following: If there is one knapsack of known (weight) capacity; a number of items with their total weight to be larger than the knapsack's capacity; and each item has a *value* associated with it: What is the combination of items that can be put in the knapsack, so that its capacity is respected, and the total value of items is maximized.

Here, the "multi-period" part relates to the fact that items may come and go depending on time. "Multi-dimensional" is related to the fact that each SLA includes resource requests of different types (reflecting the different resource types), and therefore there are more than one knapsack capacity dimensions to be considered. Finally, there are "multiple knapsacks" representing the multiple data centers that may be used.

Variants of the knapsack problem have been extensively studied in the past, and there have been various combinations of such variants examined. To the best of the author's knowledge, *this is the first time that the specific combination of the three variants is formalized and studied*.

11.2.1 Non-flexible

The integer variable x_{ij} indicates whether SLA i (out of I SLAs in total) is assigned to data center j, or not. Its values are 1 and 0, respectively:

$$x_{ij} \in \{0, 1\} \tag{11.1}$$

Each SLA may be assigned only to one out of the existing J data centers, at most:

$$\sum_{j=1}^{J} x_{ij} \leqslant 1 \tag{11.2}$$

The integer variable y_{ijklm} indicates whether resource request m of SLA i is assigned to data center j, from time slot k to time slot l (inclusive). It also takes values 1 or 0, to indicate an assignment or not:

$$y_{ijklm} \in \{0, 1\} \tag{11.3}$$

$$k \leqslant l \tag{11.4}$$

Each of these resource requests may be assigned to one data center at most. In addition, it can only be assigned once – i.e., only one value for k and one for l are acceptable:

$$\sum_{j=1}^{J} \sum_{k=1}^{H} \sum_{l=1}^{H} y_{ijklm} \leqslant 1 \tag{11.5}$$

As mentioned earlier, in this strategy the target is to assign the complete SLA, or no part of it at all. This means that, if the SLA is accepted (i.e. assigned to a data center), all of its resource requests should be accepted as well; and if the SLA is not accepted, no resource requests should be accepted either. This constraint is formulated as follows:

$$\sum_{k=1}^{H} \sum_{l=1}^{H} \sum_{m=1}^{M_i} y_{ijklm} = M_i \cdot x_{ij} \tag{11.6}$$

M_i is the number of resource requests in SLA i.

Each resource request m of SLA i has a lease type LT_m, which takes values from the set (LR, ST, BA) – for Long Running, STrict and BAtch. It also has a declared duration, DUR_{im}, a start time T^s_m and an end time T^e_m. If a start time is in the past (i.e. the SLA is already executing), it is substituted by the current time:

$$T^s_{im} = max(\lceil NOW \rceil, T^s_{im}) \tag{11.7}$$

It applies that:

$$\left.\begin{array}{r} (LT_m = LR) \wedge (k \neq T^s_{im} \vee l \neq H) \\ (LT_m = ST) \wedge (k \neq T^s_{im} \vee l \neq T^e_{im}) \\ (LT_m = BA) \wedge (k < T^s_{im} \vee l > T^e_{im} \vee l - k + 1 \neq DUR_{im}) \end{array}\right\} \Rightarrow y_{ijklm} = 0 \tag{11.8}$$

That is, for long-running leases, candidate start time k must be equal to the declared start time T^s_m and candidate end time l must be equal to the end of the planning horizon. For strict leases, k and l must be equal to declared start and end times; and for batch leases, k and l must be within the declared limits T^s_m and T^e_m, while duration should be equal to the duration declared (requested) by the user. In all other cases, the assignment variable y_{ijklm} must be 0.

For any given time slot t within the planning horizon, the 0-1 variable z_{imt} indicates whether resource request m of SLA i *may* be active at that moment:

$$z_{imt} = \begin{cases} 0, & t < T^s_{im} \vee t > T^e_{im} \\ y_{ijklm}, & T^s_{im} \leqslant t \leqslant T^e_{im} \end{cases} \tag{11.9}$$

The weight of a resource request m of SLA i in a specific dimension d is w_{imd}. As also mentioned earlier, dimensions are different types of resources, such as CPU cores, memory, storage. The sum of all weights of active resource requests at any given point in time and for any given resource type (dimension), should not exceed the amount of available resources of that type:

$$\sum_{i=1}^{I} \sum_{m=1}^{M_i} w_{imd} \cdot z_{imt} \leqslant c_{jd} \tag{11.10}$$

Finally, PEN_{iklm} is the penalty of an resource request m of SLA i, if it is active during time slots k to l. *Penalty definition is external to the model*. Here, for reasons of convenience it is given as a percentage of the resource request's price P_{im}, proportionate to its time-related violation:

$$PEN_{iklm} = P_{im} \cdot \frac{DUR_{im} - (l - k + 1)}{DUR_{im}} \tag{11.11}$$

Given Equation 11.8, the penalty in the Non-flexible strategy can be either equal to the full price P_{im}, or 0. Nevertheless, it is straightforward to apply a more complete penalty scheme, like the one elaborated in Chapter 7.

The objective is to *maximize the total penalties for resource requests assigned to data centers*; that is, to *minimize* penalties for resource requests that will be left unassigned, and therefore, will have to be refunded.

$$max \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{H} \sum_{l=1}^{H} \sum_{m=1}^{M_i} PEN_{iklm} \cdot y_{ijklm} \tag{11.12}$$

11.2.2 Flexible-items

The only difference between strategies Non-flexible and Flexible-items, is that in the latter it is possible to select only *some* of the resource requests to assign in one of the data centers. Therefore, the only part of the model to change is Equation 11.6, which becomes:

$$\sum_{k=1}^{H}\sum_{l=1}^{H}\sum_{m=1}^{M_i} y_{ijklm} \leqslant M_i \cdot x_{ij} \tag{11.13}$$

11.2.3 Flexible-time

Apart from the modification of Equation 11.6 into 11.13, in the Flexible-time strategy it is also possible to start serving some SLA resource requests later than agreed, or end them earlier than agreed. This is reflected on the relaxation of constraints from Equation 11.8. More specifically, this Equation now becomes as follows:

$$k < T^s_{im} \vee l > T^e_{im} \Rightarrow y_{ijklm} = 0 \tag{11.14}$$

It is assumed that starting to serve a resource request *earlier* than agreed, or continuing to serve it *after* its agreed expiration time, offers no value either to the customer, or to the provider.

11.3 Experimental Evaluation

11.3.1 Setup

The models described in Section 11.2, plus the two greedy algorithms (Fist-Fit and Best-Fit) were evaluated and compared with random data for incoming SLA requests. Randomness was related to the reservation type, the duration of the reservations (i.e. the durations of the SLAs), their start and end time, the amount of requested resources, and the costs (with penalties being calculated as a percentage of those costs). Normal distributions were used across the board. ILP models were built with the PuLP modeler [106] and executed using the GUROBI™ [107] state of the art commercial solver. In various benchmarks against CPLEX™ and other solvers, GUROBI™proves to be as fast or even faster in most cases (e.g. [108, 109]. All simulations were run on a 4-way server running the Linux SMP kernel v2.6.32. Each of the four processors was a 1 GHz AMD Opteron with 1 MB of attached cache memory. The total RAM of the server was 16 GB. Except for standard operating system processes, and the Secure Shell (SSH) daemon, no other processes were executing at the times of the experiments.

The experiments executed included the following setups:

- Standard configuration: 100 SLAs, 4 different resource types, 3 data centers, a planning horizon of 10 time units, and 3 resource requests per SLA;
- Increasing number of SLAs, from 100 to 1000 in steps of 100;
- Increasing number of resource types, from 1 to 5;
- Increasing number of data centers, from 1 to 5;
- Increasing planning horizon, from 5 to 30 units in steps of 5;
- Increasing number of resource requests per SLA, from 1 to 5; and
- All parameters increasing in parallel, in four steps: SLAs from 200 to 800 in steps of 200, resource types/data centers/number of resource requests per SLA from 1 to 4, and planning horizon from 5 to 20 in steps of 5.

Each setup was tested with all five strategies, and for the combinatorial ones there was a timeout of 15 minutes. It should be noted that GUROBI^{TM}is highly optimized for parallelism, and indeed it was possible to confirm that all CPUs were at top load during problem solving. Therefore, the 15 minutes of real time correspond to 1 CPU-hour due to the four processors of the server used. For the last type of experiments (variables increasing in parallel), there was initially tested a 5th configuration with 1000 SLAs, 5 resource types/data centers/SLA resource requests and 25 time units in the planning horizon, but even the pre-solving phase (leading to a bound value) was not finishing in the 15 minutes allocated; therefore, this setup was dropped from the experiments. The complete set of experiments was run *five times*, and results were averaged when suitable (e.g. for normalized values).

In all experiments, the amount of available resources was crafted during modeling time to be significantly less than the amount required. Without loss of generality, it was assumed that all scheduled SLAs start in the future; a pseudo-random number generator following a gaussian distribution was used to return start/end times, required resources per SLA resource request, resource type, etc. Figure 11.4 illustrates the required resources over time, as a percentage of the available resources. This graph concerns only experiments with a planning time horizon of 30 units, but for other experiments the resource requirements are similar.

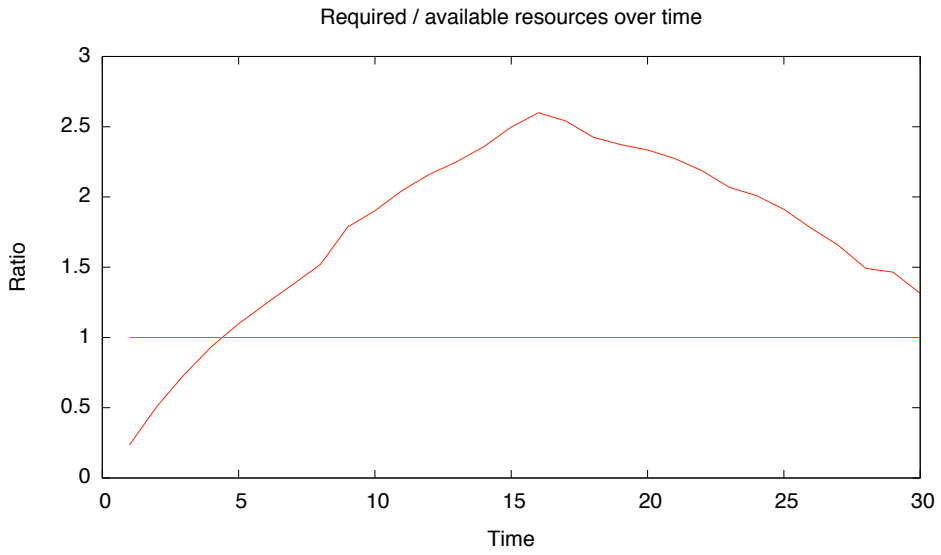

Figure 11.4: Required / available resources ratio

11.3.2 Results

Resource Utilization

The first goal was to see how good use of the available resources, do the various strategies offer. Figure 11.5 illustrates exactly that. It is straightforward to see that Flexible-times provides the best utilization, converging towards a ratio of 100% much faster than the rest. Flexible-items is second best, then follows Non-flexible. The two greedy algorithms achieve the worst resource utilization, but they should be compared only to the Non-flexible strategy, as they are also applying an all-or-nothing approach for the SLAs under consideration.

Complexity and Runtime Behavior of Optimization

It was also important to achieve an understanding of the complexity of the different ILP models, as all parameters were increasing in parallel. The solver's output was intercepted to see how fast it is approaching the best-bound values, as well as how long the pre-solving phase takes. Figures 11.6 - 11.9 illustrate these results. The numbers in parentheses (in the captions) refer to the number of SLAs, resource types, data centers, planning horizon time units, and resource requests in each SLA – in this particular order.

In Figure 11.6, the simplest of the four instances is solved instanta-

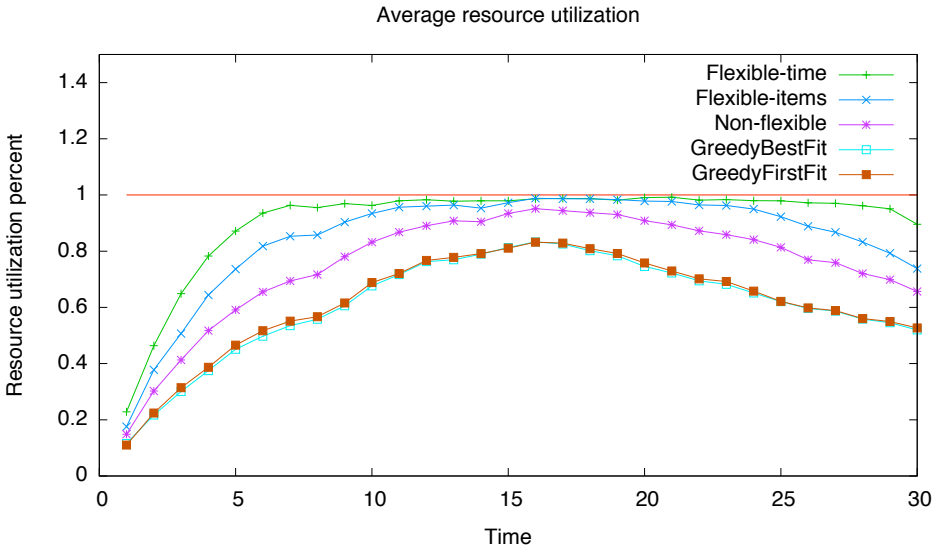

Figure 11.5: Average resource utilization (percentage)

neously in all three models. In Figure 11.7, Non-flexible and Flexible-items strategies start solving immediately with a distance from best-bound objective less than 1% on average, and quickly converge even further. The Flexible-time strategy goes through a very short pre-solving phase, then starts solving from a distance of approximately 10% on average, but converges in less than 10 seconds to almost the optimal value.

In the third problem instance, the difference of complexity starts to become more apparent. The Non-flexible strategy starts solving almost immediately, with a large distance from best-bound objective (more than 15% on average), but converges in less than 30 seconds to less than 2%. Flexible-items behaves even better, starting from a distance of approximately 2.5% and almost reaching best-bound a few seconds later. Flexible-time is pre-solving for approximately one minute on average, then remains quite far from best bound (more than 20%) for some time, and then quickly approaches best-bound – a total of almost 3.5 minutes for that).

The results from the last instance of this problem were further revealing about the exponential complexity of at least two out of three models. While the Flexible-items strategy starts far from best-bound, it approaches it to a distance less than 2% in approximately 30 seconds on average. Strategy Non-flexible starts at a distance of almost 30%, then slowly approaches best-bound taking almost 4 minutes to reach a distance of 5%, and achieving a distance very close to optimality just before

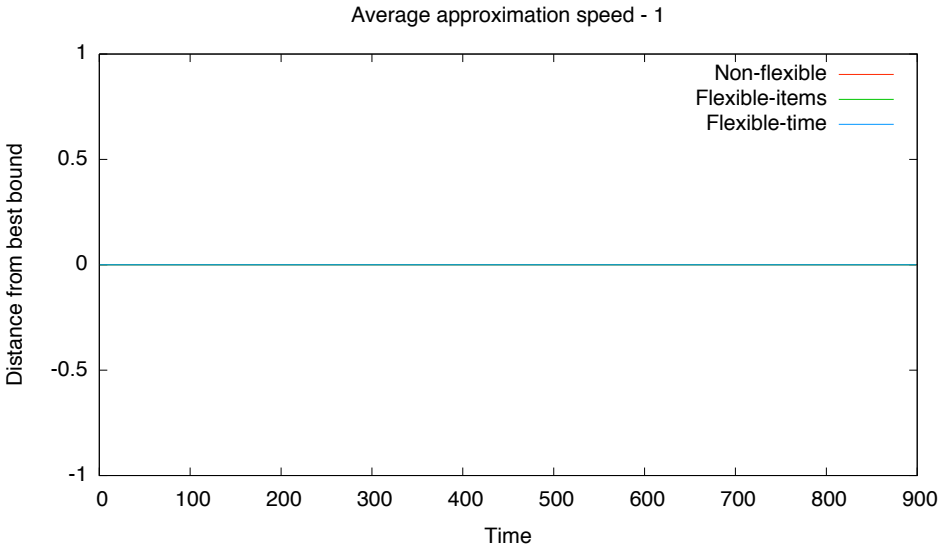

Figure 11.6: Solving/Approximation speed for (200, 1, 1, 5, 1)

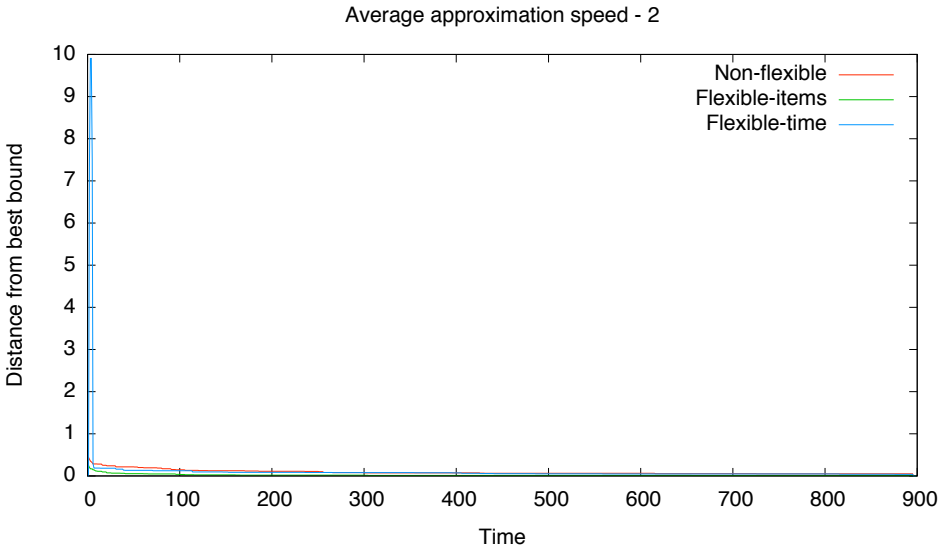

Figure 11.7: Solving/Approximation speed for (400, 2, 2, 10, 2)

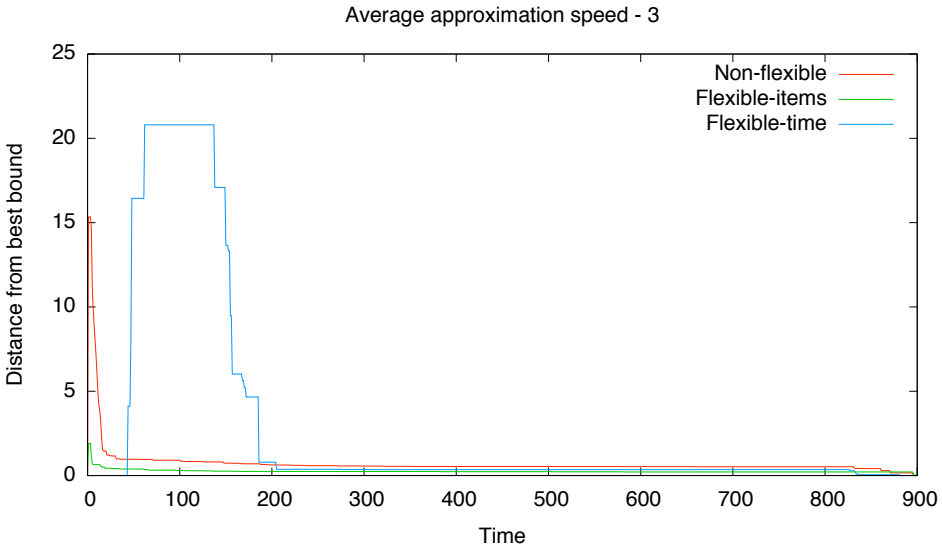

Figure 11.8: Solving/Approximation speed for (600, 3, 3, 15, 3)

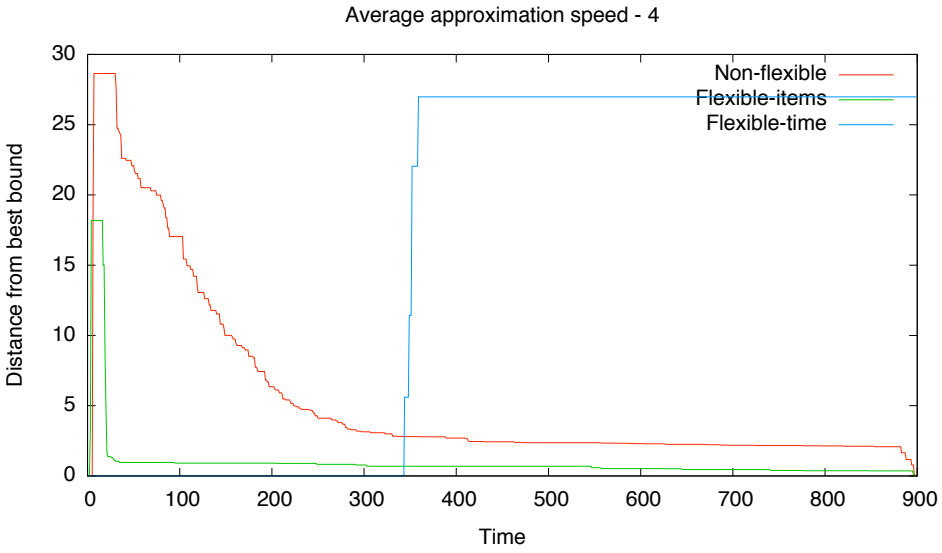

Figure 11.9: Solving/Approximation speed for (800, 4, 4, 20, 4)

the timeout. The last strategy, Flexible-time, is going through pre-solving phase to find a reasonably good first solution for an average of 7 minutes. After that, it starts solving at a distance of more than 25% from best-bound objective, but never comes any closer before the 15-minute timeout.

Clearly, with regard to complexity, Flexible-time is the most difficult problem and Flexible-items is the easiest. It is interesting to explore to some more detail, which of those five parameters causes the largest toll to performance degradation. Figures 11.10–11.14 illustrate the distance from best-bound for all combinatorial strategies. One can see that:

- Increasing SLAs affect all three ILPs positively. This is reasonable, as it provides more options for a good initial choice when the integrity constraint is relaxed during pre-solving.

- Increasing number of resource types affects negatively all strategies. This was expected; it is always underlined across literature, that increasing dimensions in multi-dimensional knapsack variants causes significant increase in the problem's complexity (e.g. [105] by Kellerer et al.)

- Like with resource types, the increase in the number of data centers also affects negatively the solving of the problem. Again, this is to be expected: In the multiple-knapsack variant, increasing number of knapsacks causes increased complexity.

- Changing the planning horizon has, initially, a small (and inconsistent) effect for all three strategies. In order to get a better picture, more extensive experiments were conducted, increasing the time horizon up to 75 time units. It was found that, starting at 70 time units, the pre-solving phase of the Flexible-time strategy did not finish within the 15-minute timeout. Therefore, the respective graph was produced with a planning horizon of 65 units at most. The result is that the respective increase affects only the Flexible-time strategy – as was expected, due to the increase in decision variables that it is causing. After a threshold (55 time units, in this case) the distance from best-bound increases from less than 5%, to more than 20%. Flexible-items and Non-flexible are not affected in some consistent way.

- Similarly with the planning horizon, changes in the number of resource requests per SLA had small and inconsistent effects on problem complexity. Thus, extended experiments were run with up to 15 resource requests per SLA. There is a trend that reduces distance

from best-bound for all three strategies, as the number of resource requests increases. This is an effect similar to the increase in the number of SLAs.

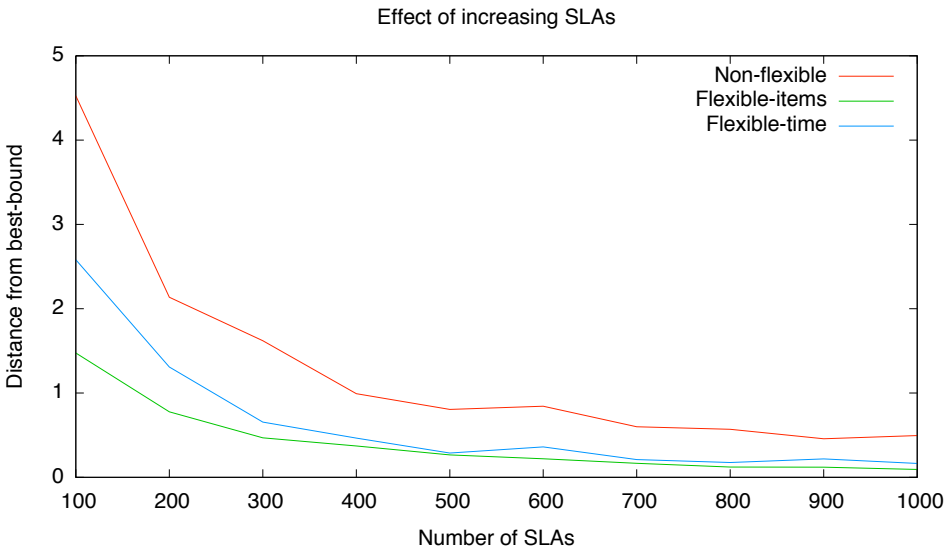

Figure 11.10: Effect of increasing number of SLAs

Both greedy strategies were executing in less than 1 second for all problem problem instances, so from a performance point of view they are preferable. However, one must assess them also taking into account their fitness for the purpose and how well they behave with regard to approximation of an objective close to optimal.

Affected Users

Figures 11.15 and 11.16 provide an account of user satisfaction, based on numbers of accepted SLAs, and accepted SLA resource requests. Due to the allowed flexibility, as expected, Flexible-items and Flexible-time can accept many more SLAs, even if partially. Nevertheless, when looking at accepted SLA resource requests, their number is smaller – especially for Flexible-items, which accepts a resource request in full, or not at all. Although the model's objective is to minimize penalties in all cases, it is important to pay attention to how many users are affected, and to what extent. The number of users receiving no service whatsoever is the largest with the two greedy algorithms, and the smallest with Flexible-time.

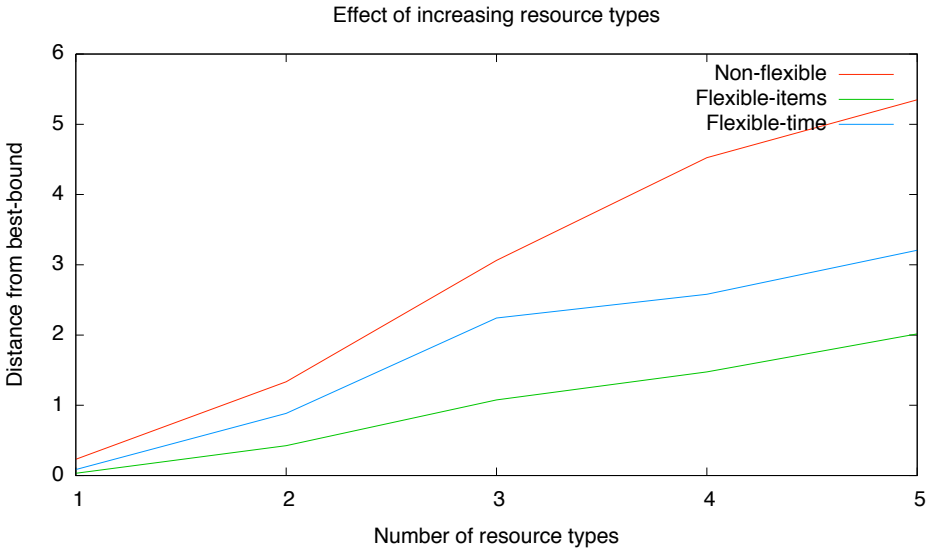

Figure 11.11: Effect of increasing number of resource types

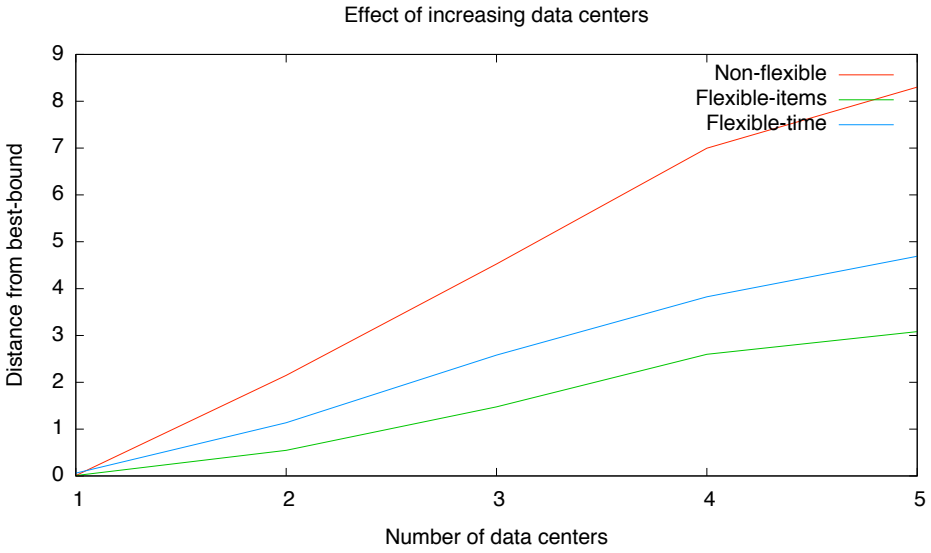

Figure 11.12: Effect of increasing number of data centers

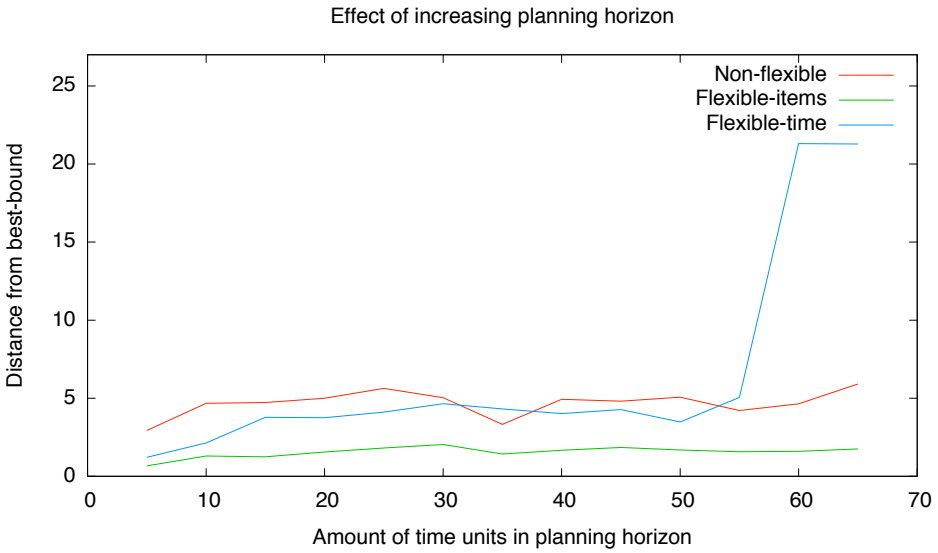

Figure 11.13: Effect of increasing planning horizon

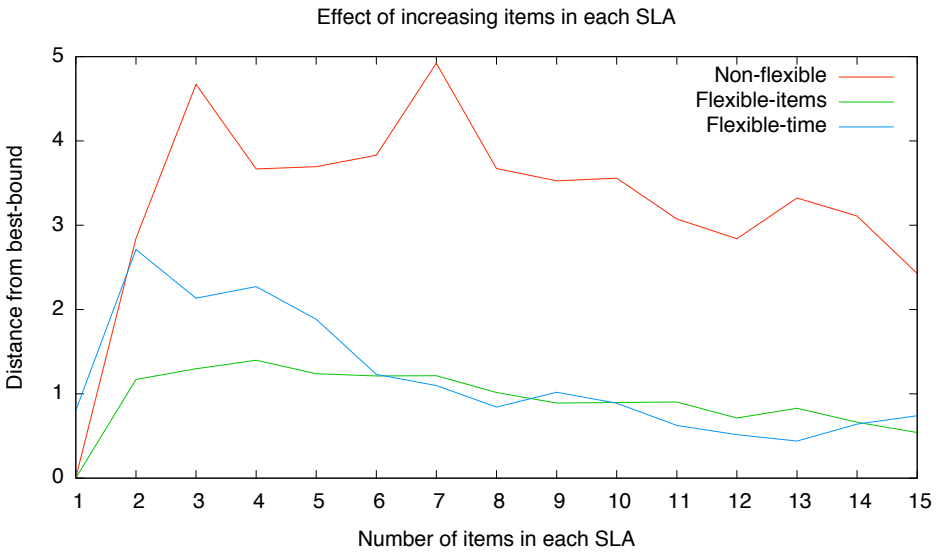

Figure 11.14: Effect of increasing number of resource requests in each SLA

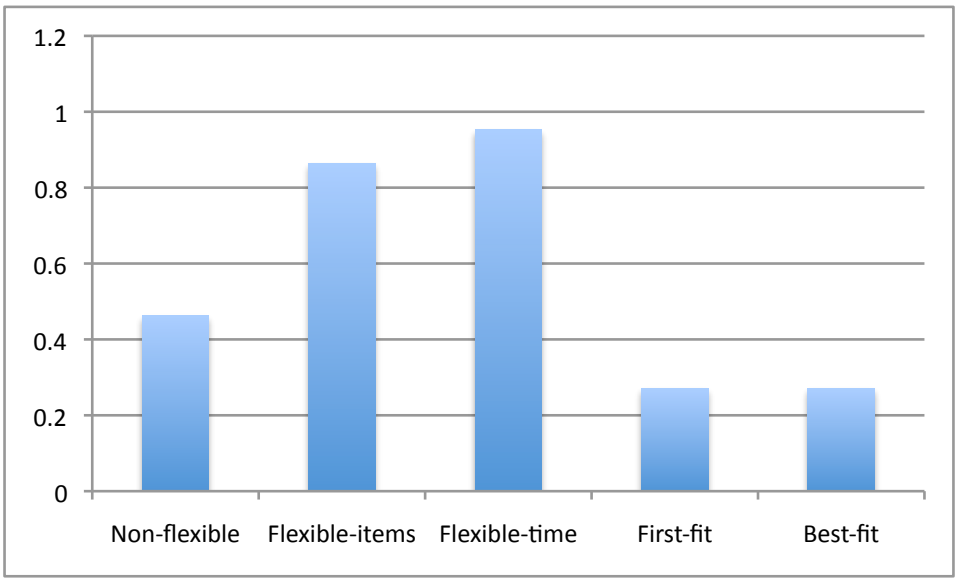

Figure 11.15: Placed SLAs per strategy (percent of total SLAs)

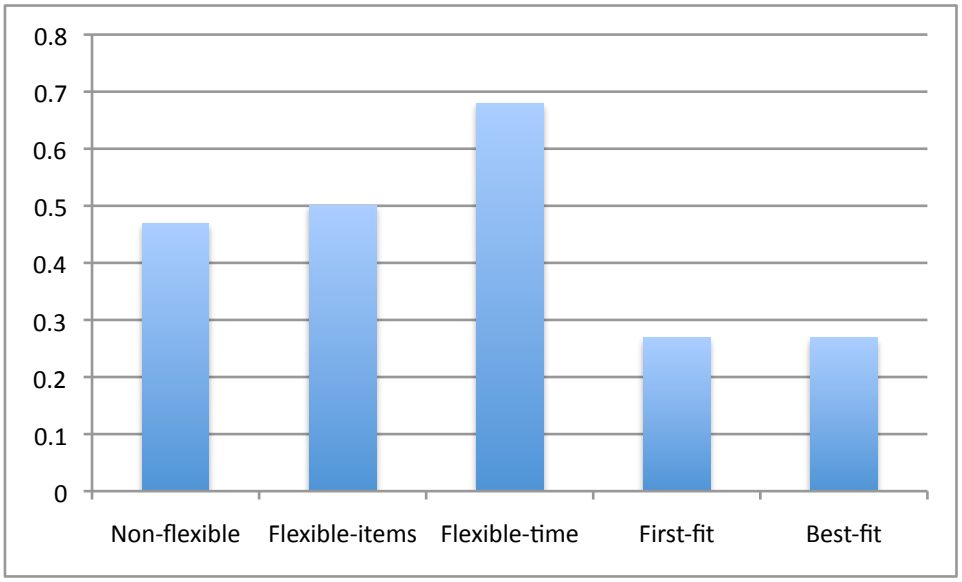

Figure 11.16: Placed resource requests per strategy (percent of total resource requests)

Distance from Optimality

The last chart, Figure 11.17, provides an overview of *avoided* penalties, as a percentage to total penalties. It is reminded that the objective was to maximize avoided penalties. In order to be able to compare with the greedy algorithms, for which there is no notion of "best-bound objective", the charts do not use that value as it was the case with Figures 11.6–11.9. Rather, the full penalties of all SLAs/resource requests under consideration are used.

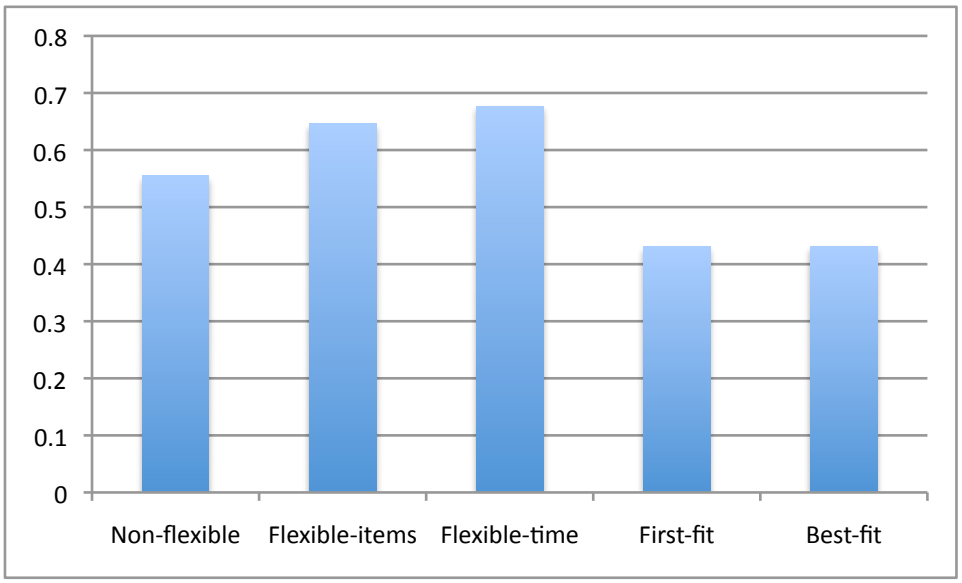

Figure 11.17: Avoided penalties per strategy (percent of total penalties)

One can see that Flexible-time provides the best result, with Flexible-items a close second. Non-flexible is third, with quite some difference from the second, and finally the two greedy algorithms fall behind significantly.

Experimental Conclusions

Assessing all information presented, it appears that the best strategy to follow is Flexible-items. It performs very well in comparison to the other two ILPs with regard to the speed of approaching a reasonably good solution; it offers a very small difference from the best objective found by Flexible-time, less than 5% on average; and affected users, although more than Flexible-time, they are acceptable as a number under the extreme circumstances simulated.

11.4 Related Work

Some prior art related to SLA adjustment in case of resource failures, was already presented in Chapter 10. Goiri et al. [102] and Roy / Mukherjee [103] monitor SLAs during runtime, and for each one that fails, the proposed systems try to find alternative execution environments. Conversely, the work proposed here looks into the case of massive failures, in which case it is not clear whether the former approaches would succeed due to the conflicts between the requesting agents. Penalty consequences are also unclear. In addition, the present chapter proposes a comprehensive model to treat such problems of extreme failures in their entirety.

Outside the context of SLAs, nevertheless trying to optimize *service performance*, Song et al. present in [110] an architecture and a methodology for controlled, adaptive resource scheduling in the data center. They produce a linear program called the "resource flowing" problem, to model reservations and releases of resources by VMs; then solve it with the Simplex method.

In the context of internal IT infrastructure at Intel Corporation used for chip simulations, Phan et al. present in [111] an approach for cost-based dynamic rescheduling of low-priority jobs. The problem they try to tackle is that low-priority jobs are often pre-empted by high-priority ones, although there are significant amounts of available resources in the infrastructure as a whole. Process migration is not possible due to problems with multi-threaded processes, and VMs are not used due to the heavy performance toll (therefore, VM migration is not possible either). Hence, the report explores the possibility of *restarting* the processes. Whether a job will be kept suspended or be rescheduled is a cost-based decision, and has to do with "wasted run time" (the process execution time leading to pre-emption from a high-priority job), expected remaining completion time, and other similar measures.

With regard to the knapsack problem and its applications in the area, the work closest to the one presented here is [112] by Lau and Lim. They use the multi-period, multi-dimensional knapsack problem to model what is known as *Available-to-Promise* in logistics; that is, the capability of a supplier to commit to meeting the requests of customers, during a period of time (the planning horizon). Clearly this bears a lot of similarity to IaaS SLAs, where the provider promises to make available the resources during some time period. It is particularly applicable to the case of batch SLAs, where the provider has some freedom as regards when to offer the resources. The authors develop a two-phase heuristic, comprising a greedy algorithm followed by a Tabu-search improvement. Additionally,

an ant-colony based approach is evaluated. Their work is trying to optimize profit; and an additional difference to the work presented here is the single vs multiple knapsacks.

Finally, in [113], Kelly makes explicit the relationship between the knapsack problem, and the *Winner Determination Problem* in combinatorial auctions, which offer a suitable model for multiple parallel SLA negotiations among autonomous agents. The actual motivation comes from resource allocation requirements in data centers, and the formulation is a multi-dimensional, multiple-choice knapsack problem. Multiple-choice is related to the selection of only one, out of many options in a list of requested resource bundles (which has been solved here at an earlier stage, using BDDs). The multiple dimensions represent resource types, as is also the case in the present work. The author presents a dynamic programming solver, and recognizes the significant complexity increase as the number of resource types increases. The author's model concerns a single data center, so no discussion exists on multiple knapsacks.

Summary and Conclusions

In this Chapter, an Integer Linear Programming model was developed, as a means of SLA rescheduling and penalty minimization in the case that resources become scarce. The working example here was the malfunction of a complete data center. Through extensive experimentation, it was possible to see that the constructed model behaves favorably to greedy algorithms, although foreseeably slower to approach some optimal value. Three combinatorial rescheduling strategies were explored: One with no flexibility whatsoever (complete acceptance of an SLA, or complete rejection); one where it was possible to accept and reschedule only some of the requested resource bundles within an SLA; and one where it was additionally possible to offer some bundles later, or release them earlier than agreed. This third approach offered the best final objective *on average*, normalized across all experiments. Yet, for large instances the complexity grows so much that even after a lot of time it remained very far from optimal values. The non-flexible strategy was faster, but its results with regard to average achieved objective were quite worse. The fastest method was the one to accept some of the resource bundles, but for the time and duration agreed in the SLA. Its results, on average, were only slightly worse than the Flexible-time strategy, therefore its performance is enough reason to prefer it in realistic situations.

Part IV

Conclusions

Chapter 12

Conclusions

Throughout this dissertation the question asked was, what are those necessary building blocks to enable service hierarchies with dependability, using SLAs as an instrument for this purpose. It was the question that the thesis tried to answer to, by providing new approaches and solutions towards this goal.

As mentioned in Section 1.2, when looking at the topic from a very high level, automated SLA negotiation and provisioning can be reduced to the schematic of Figure 1.2 (repeated here for convenience, as Figure 12.1). The dashed boxes illustrate the gaps to which this thesis tried to provide solutions and answers, in addition to a formal problem definition.

12.1 Summary of Contributions

Defining the problem generically, it was possible to express the relationships among services and their negotiable properties independent of the application domain. This definition cannot be directly applied, but rather needs significant refinement before using it in any specific use case; nevertheless, it provides the necessary grounds for getting an understanding of the problem, and coming closer to domain-independent recipes that can be combined with other tools and achieve the set goal of automated, hierarchical SLA management. Prior art was discussing service dependencies, or looking at QoS-related dependencies in specific domains. This is, to the best of the author's knowledge, the first effort to define SLA dependencies and hierarchies formally. An additional very important result of this specific work, is the conclusion that converting an SLA to many others cannot be disentangled from SLA negotiation – rather, the two processes are closely related and, as a matter of fact, one could say that "SLA

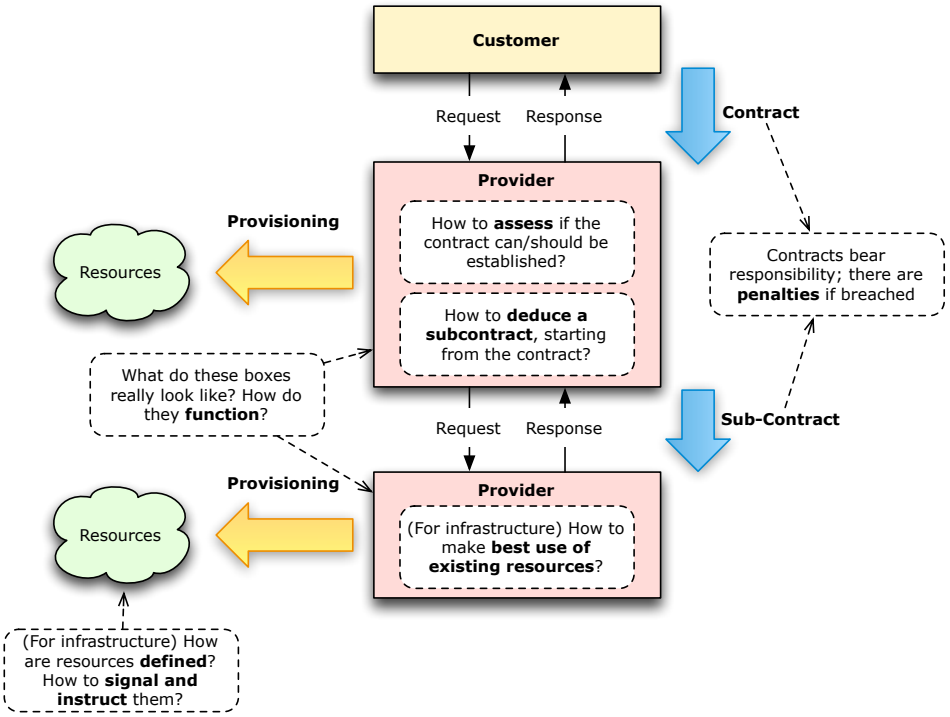

Figure 12.1: High-level scenario

translation" is a part of negotiation itself.

The next step was to define a management architecture. The resulting *SLA Management Instance* (SAMI) is a complete architectural pattern that has no specific ties to any application domain; rather, it foresees that it must be extended before applied. That said, a generic infrastructure for a SAMI was created as part of the project that supported this work [1], using state-of-the-art tooling and resulting in a framework already being customized and applied by four industrial use cases. The presented architecture is extending State-of-the-Art in a number of ways, the most important of which is the special consideration for SLA hierarchies. Further to that, the domain-agnostic and technology-independent nature of the architecture makes it applicable to very diverse use cases.

Before discussing SLA assessment, a link to business objectives was necessary. A Service Level Agreement is a contract, and as such, it includes not only rewards (service price), but also penalties if the described service is not delivered. One could argue that without the penalties, an SLA is nothing more than a smart provisioning request. There has been some prior art in SLA penalty models, but the author felt that it was not sufficient and was lacking necessary expressive power. Thus,

a new model was proposed in this work. Using this model it is possible to form very complex penalty expressions, also associated with contract price and customer's business values; therefore, it includes notions of *fairness*. In addition, the application of this generic penalty model to an example use case illustrated that it is possible to deduce subcontract penalties based on the penalties within the original contracts.

The last part of generic, domain-independent contributions was related to the decision-making part of negotiation. The dissertation's contribution is the application of a structure well-known in VLSI design, the *Binary Decision Diagrams* as a model for SLA representation. This is based on the capability to reduce SLA elements in boolean constants or variables. The author deems this part of the thesis to be an important contribution, believing it can pave the way for efficient automated SLA management. In the SLA@SOI project, it became obvious that SLAs, if pragmatic, they can be extremely complex. Indeed, so complex that their sheer parsing, and of course their processing for reasons of decision-making, is a task of enormous difficulty. The Reduced Ordered BDD structure has the very interesting property of being *canonical*; that is, it provides a unique and unambiguous representation of the boolean function from which it is constructed. Eventually, given domain-specific term dictionaries and the knowledge how to use them for creating SLAs, combined with BDDs, can result in structures that are immensely more manageable than their original SLA counterparts. As a proof of concept, for a specific SLA offer the approach showed how it is possible to use a BDD and decide if the offer should be accepted, or not.

Following, Part III of the thesis applied these principles, albeit in simple form, in the domain of Cloud Computing and more specifically *Infrastructure as a Service*. One necessary detail for the specific domain, that had to prepend the discussion on IaaS SLA planning, was how to actually manage *the resources* themselves. This concerns a proper representation, as well as a set of functions for reserving them – immediately or in advance. The supporting models provided are standards-compliant and designed to be agnostic of specific resource types: They may refer to computing servers, network storage, active network equipment and circuits, scientific instruments, sensors, etc. Already and by itself, the supported resource diversity is an interesting property of the model, and a contribution to State-of-the-Art.

Having the necessary constructs available, an infrastructure SLA content was described, and based on that two different greedy algorithms were tested as a means for online SLA planning within a data center. This corresponds to the lower of the two provider entities in Figure 12.1. The

evaluation took place in the context of the Planning, Optimization and Adjustment module of the SAMI architecture, as a possible implementation for this specific IaaS setup. As there has been significant amount of work in this area during the last few years, the main interest of the work here was to see what is the energy footprint of these two fast algorithms, and how well they were performing with regard to resource utilization. The conclusion after experimentation was that a best-fit approach, that assigns SLAs to those servers that end up with the least available resources after the assignment, results in significant gains with regard to the amount of servers that are switched on. Nevertheless, the completion time of the algorithm was an order of magnitude worse than its first-fit counterpart, for three pools of 5000 servers each. Even as such, the delay of up to 20 seconds for our setup can probably be considered as acceptable.

Finally, a penalty-driven planning and optimization methodology was considered in the last Chapter. The scenario in mind was that of a resource-scarce environment, where resource requirements (far) exceed resource availability. Three ILP models were developed and solved, to represent three different strategies of varying flexibility with regard to time, or grouping constraints. These strategies express different possible ways that a coordinating SAMI instance (the higher one in Figure 1.2) would react to this scenario, while negotiating with data-center SAMIs. Via extensive experimentation, it became clear that the best strategy to apply – at least in small to average problem instances – is to consider the existing SLAs with the possibility of discarding some of their parts (i.e. some of the requested resources). Experimenting with the time dimension resulted in very poor performance, while selecting to accept SLAs in full or not at all, resulted in significantly more penalties. With regard to the latter, two tested greedy algorithms (first-fit and best-fit) performed even worse.

12.2 Critical View of Presented Work

The area tackled by this doctoral dissertation is a very large one, far too large to be addressed here in full. Various limitations of time and resources in general affected the results to some extent, although hopefully not enough to have a negative effect on the thesis' contributions.

The most important problem that the author identifies, is the complexity of some of the produced formalizations. The penalty model from Chapter 7 is extremely powerful, nevertheless this also means that a large number of parameters must be taken into account. It assumes that relationships among different SLA guarantees and service parameters are

known to the customer, and therefore assumes a very detailed model of interactions within an application and the supporting SOI. In addition, it is assumed that the customer is fully aware of the business value that certain objectives and guarantees have, which may be a wrong assumption in plenty realistic scenarios. The choice to favor expressiveness over simplicity was a conscious decision, in hope that future developments will allow such fine-grained understanding of SOAs, SOIs and the respective applications.

Similarly, the application of BDDs to SLA modeling presumes the capability to identify terms generically, within a domain and application. It requires detailed knowledge of the semantics of an application, which could have important repercussions on the wide applicability of the proposed methodology. Additionally to that, there is a gap in using the canonical form of the structure for outsourcing and decision-making, related to matching paths from different BDDs and finding out whether they are equivalent. The author believes that the area to look into is that of *Term Rewriting Systems* [114]. This could, nevertheless, be a different doctoral thesis on its own right, and therefore no further work towards that direction took place.

Finally, the author would be very pleased if the proposed architecture was further applied to real-world use cases in the context of the project that funded the present work. Due to time restrictions and the fact that the project is ongoing, this was not possible.

12.3 Concluding Statements and Future Directions

If one final statement must be extracted from the present thesis, is that hierarchical SLA management *is* possible. There are multiple challenging issues about it, and certainly this dissertation does not reply to everything, yet it hopefully contributes to the more apparent areas. Furhermore, the extension to IaaS planning illustrates the applicability of these concepts onto a domain which is currently one of the most active areas of service computing.

One problem that has been identified in multiple occasions and various fora, is the lack of a common language for SLA negotiation. This includes both a model and a protocol. WS-Agreement is an important step in that direction, but at the moment has two significant problems:

1. It is considered to be *very* complex; specifically, its reliance on the *Web Services Resource Framework* (WSRF) [115] has proved to be

problematic for the community;

2. It supports only single-shot negotiations, without multiple rounds which are necessary for meaningful business interactions.

Service registries for automated discovery and binding are not sufficiently polished yet, either. Both compatibility issues, but also the complexity of the topic (security, matching and semantics, policy / administrative domains) hinder the adoption of global service registries and large, dynamic service ecosystems.

Monitoring SLAs is also a topic that is widely researched at the moment, especially due to the commercial interest in the area. Here, there are two important considerations; one is of technical nature, related to *what* must be monitored and what are the *associations* among different signals received as part of a single SLA being monitored. The other problem is one of policy; that is, providers do now allow transparent access to service execution parameters, making it very difficult for customers to have an educated understanding of the SLA state. The gap seems to be closing, with recent trends of trusted third parties: companies that act as mediators, are trusted by both the providers and the customers, and monitor the SLAs on behalf of both.

Negotiation and planning strategies (including subcontracting decisions) remains, without doubt, the most demanding field of automated SLA management. The author strongly believes that the way to tackle this problem is to produce a sufficient amount of generic recipes (such as the BDD representation), that can be combined and applied on different domains. The complexity of SLA planning can be notoriously large, as also illustrated in Chapter 11. Areas of mathematics such as combinatorial optimization, operational research, evolutionary computing, and others, provide a solid foundation to build upon towards this direction; “standing on the shoulders of giants”, combined with original cross-disciplinary ideas are sure to equip us better for the purpose.

Bibliography

[1] "The SLA@SOI Project." http://www.sla-at-soi.eu/ (Last retrieved: 09/2010).

[2] M. Papazoglou and D. Georgakopoulos, "Service-Oriented Computing," *Communications of the ACM*, vol. 46, no. 10, pp. 25–28, 2003.

[3] L. Zhang, J. Zhang, and H. Cai, *Services Computing*. Springer-Verlag New York Inc, 2007.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1." World Wide Web Consortium - W3C, 2001.

[5] Organization for the Advancement of Structured Information Standards - OASIS, "Web Services Business Process Execution Language (WSBPEL)." http://www.oasis-open.org/committees/wsbpel/ (Last retrieved: 09/2010).

[6] I. Katzela and M. Schwartz, "Schemes for fault identification in communication networks," *IEEE/ACM Transactions on Networking*, vol. 3, pp. 753–764, Dec 1995.

[7] B. Gruschke, "Integrated event management: Event correlation using dependency graphs," in *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, pp. 130–141, 1998.

[8] E. Knorr and G. Gruman, "What cloud computing really means." http://www.infoworld.com/d/cloud-computing/what-cloud-computing-really-means-031 (Last retrieved: 09/2010).

[9] "Amazon Elastic Compute Cloud - EC2." http://aws.amazon.com/ec2/ (Last retrieved: 09/2010).

[10] "Amazon Simple Storage Service - S3." http://aws.amazon.com/s3/ (Last retrieved: 09/2010).

[11] M. Rappa, “The utility business model and the future of computing services,” *IBM Systems Journal*, vol. 43, no. 1, pp. 32–42, 2004.

[12] I. Foster, C. Kesselman, and S. Tuecke, “The anatomy of the grid: Enabling scalable virtual organizations,” *International Journal of High Performance Computing Applications*, vol. 15, no. 3, p. 200, 2001.

[13] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai, “SLA Decomposition: Translating Service Level Objectives to System Level Thresholds,” *International Conference on Autonomic Computing*, p. 3, 2007.

[14] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, “Web Services Agreement specification (WS-Agreement).” http://www.ogf.org/documents/GFD.107.pdf, 2007 (Last retrieved: 09/2010).

[15] K. Nichols, S. Blake, F. Baker, and D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers,” tech. rep., RFC 2474, 12 1998.

[16] C.-H. Chang, P. Kourtessis, and J. Senior, “GPON service level agreement based dynamic bandwidth assignment protocol,” *Electronics Letters*, vol. 42, pp. 1173–1174, September 2006.

[17] W. Fawaz, B. Daheb, O. Audouin, M. Du-Pond, and G. Pujolle, “Service level agreement and provisioning in optical networks,” *IEEE Communications Magazine*, vol. 42, pp. 36–43, Jan 2004.

[18] D. Nowak, P. Perry, and J. Murphy, “Bandwidth allocation for service level agreement aware Ethernet passive optical networks,” in *Global Telecommunications Conference, 2004. GLOBECOM ’04. IEEE*, vol. 3, pp. 1953–1957, 2004.

[19] A. Sahai, A. Durante, and V. Machiraju, “Towards automated SLA management for Web services,” *Hewlett-Packard Research Report HPL-2001-310 (R. 1)*, 2001.

[20] R. Yahyapour and P. Wieder, *Encyclopedia of Software Engineering*, ch. Service Level Agreements in Grids. Taylor & Francis Group, 2009. To appear.

[21] The TeleManagement Forum, "SLA Management Handbook, Volume 2, Concepts and Principles, Release 2.5," tech. rep., The TeleManagement Forum, Morristown, New Jersey, United States, 2005.

[22] A. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003.

[23] R. Smith, "The contract net protocol," *IEEE Transactions on Computers*, vol. 29, no. 12, pp. 1104–1113, 1980.

[24] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou, "The complexity of computing a Nash equilibrium," *Commun. ACM*, vol. 52, no. 2, pp. 89–97, 2009.

[25] L. Jin, V. Machiraju, and A. Sahai, "Analysis on Service Level Agreement of Web Services," tech. rep., Software Technology Laboratory, HP Laboratories Palo Alto, 2002.

[26] OASIS, "Universal Description, Discovery and Integration (UDDI)." http://www.oasis-open.org/committees/uddi-spec, 02 2005 (Last retrieved: 09/2010).

[27] A. Sahai, V. Machiraju, M. Sayal, A. van Moorsel, and F. Casati, "Automated sla monitoring for web services," *Management Technologies for E-Commerce and E-Business Applications*, pp. 28–41, 2002.

[28] A. Leff, J. T. Rayfield, and D. M. Dias, "Service-level agreements and commercial grids," *IEEE Internet Computing*, vol. 7, no. 4, pp. 44–50, 2003.

[29] H. Zhang, K. Keahey, and W. Allcock, "Providing data transfer with qos as agreement-based service," *Services Computing, 2004. (SCC 2004). Proceedings. 2004 IEEE International Conference on*, pp. 344–353, Sept. 2004.

[30] M. Aiello, G. Frankova, and D. Malfatti, "What's in an agreement? an analysis and an extension of ws-agreement," *Service-Oriented Computing - ICSOC 2005*, pp. 424–436, 2005.

[31] J. Sauvé, F. Marques, A. Moura, M. Sampaio, J. Jornada, and E. Radziuk, "Sla design from a business perspective," *Ambient Networks*, pp. 72–83, 2005.

[32] S. S. Chawathe, "Strategic web-service agreements," *Web Services, IEEE International Conference on*, vol. 0, pp. 119–126, 2006.

[33] P. McKee, S. Taylor, M. Surridge, R. Lowe, and C. Ragusa, "Strategies for the service market place," *Grid Economics and Business Models*, pp. 58–70, 2007.

[34] D. Barbagallo and M. Comuzzi, "Towards understanding the role of adverse selection and moral hazard in automated negotiation of service level agreements," in *SIPE '08: Proceedings of the 3rd international workshop on Services integration in pervasive environments*, (New York, NY, USA), pp. 7–12, ACM, 2008.

[35] H. Wada, P. Champrasert, J. Suzuki, and K. Oba, "Multiobjective optimization of sla-aware service composition," *Services, IEEE Congress on*, vol. 0, pp. 368–375, 2008.

[36] H. Bidgoli, *The Internet Encyclopedia*. John Wiley & Sons Inc, 2004.

[37] "Enterprise resource planning." http://en.wikipedia.org/wiki/Enterprise_resource_planning (Last retrieved: 09/2010).

[38] M. Ehrgott, *Multicriteria Optimization*. Springer-Verlag New York, Inc., 2005.

[39] S. Fatima, M. Wooldridge, and N. Jennings, "A Comparative Study of Game Theoretic and Evolutionary Models of Bargaining for Software Agents," *Artificial Intelligence Review*, vol. 23, pp. 187–205, 04 2005.

[40] C. Figueroa, N. Figueroa, A. Jofre, A. Sahai, Y. Chen, and S. Iyer, "A Game Theoretic Framework for SLA Negotiation," tech. rep., HP Laboratories, 2008.

[41] A. Keller, U. Blumenthal, and G. Kar, "Classification and Computation of Dependencies for Distributed Management," *IEEE Symposium on Computers and Communications*, p. 78, 2000.

[42] A. Keller and G. Kar, "Determining service dependencies in distributed systems," in *IEEE International Conference on Communications (ICC 2001)*, vol. 7, pp. 2084–2088, 2001.

[43] P. Hasselmeyer, "Managing dynamic service dependencies," in *12th International Workshop on Distributed Systems: Operations & Management (DSOM 2001)*, pp. 141–150, 2001.

[44] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via

inference of multi-level dependencies," in *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 13–24, 2007.

[45] E. Di Nitto, M. Di Penta, A. Gambi, G. Ripa, and M. Villani, "Negotiation of service level agreements: An architecture and a search-based approach," *LECTURE NOTES IN COMPUTER SCIENCE: Proc. ICSOC'07*, vol. 4749, p. 295, 2007.

[46] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[47] D. Wolpert and W. Macready, "Coevolutionary free lunches," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 721–735, 2005.

[48] "Spring." http://www.springsource.org/osgi (Last retrieved: 09/2010).

[49] "OSGi." http://www.osgi.org/ (Last retrieved: 09/2010).

[50] "Equinox." http://www.eclipse.org/equinox/ (Last retrieved: 09/2010).

[51] H. Ludwig, A. Dan, and R. Kearney, "Cremona: an architecture and library for creation and monitoring of WS-agreements," in *Proc. 2nd international Conference on Service Oriented Computing (ICSOC '04)*, pp. 65–74, 2004.

[52] J. Padgett, K. Djemame, and P. Dew, "Grid-Based SLA Management," *Advances in Grid Computing - EGC 2005*, pp. 1076–1085, 2005.

[53] G. Koumoutsos, S. Denazis, and K. Thramboulidis, "SLA e-Negotiations, Enforcement and Management in an Autonomic Environment," *Modelling Autonomic Communications Environments*, pp. 120–125, 2008.

[54] A. Keller and H. Ludwig, "Defining and Monitoring Service Level Agreements for dynamic e-Business," in *Proceedings of the 16th System Administration Conference (LISA 2002)*, 2002.

[55] M. Debusmann and A. Keller, "SLA-driven management of distributed systems using the common information model," *IFIP/IEEE*

8th Int. Symp. on Integrated Network Management, pp. 563–576, March 2003.

[56] Distributed Management Task Force, "Common Information Model," 01 2008.

[57] C. Bartolini, A. Boulmakou, A. Christodoulou, A. Farrell, M. Salle, and D. Trastour, "Management by Contract: IT Management driven by Business Objectives," *Proc. 11th HP Openview University Association (HP-OVUA) Workshop*, 2004.

[58] A. Paschke and M. Bichler, "Knowledge representation concepts for automated SLA management," *Decision Support Systems*, vol. 46, no. 1, pp. 187 – 205, 2008.

[59] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu, "Utility computing SLA management based upon business objectives," *IBM Syst. J.*, vol. 43, no. 1, pp. 159–178, 2004.

[60] M. B. Chhetri, J. Lin, S. Goh, J. Y. Zhang, R. Kowalczyk, and J. Yan, "A Coordinated Architecture for the Agent-based Service Level Agreement Negotiation of Web Service Composition," *Australian Software Engineering Conference*, pp. 90–99, 2006.

[61] C. Bartolini, C. Preist, and N. Jennings, "A generic software framework for automated negotiation," in *First International Conference on Autonomous Agent and Multi-Agent Systems*, 2002.

[62] J. B. Kim and A. Segev, "A framework for dynamic eBusiness negotiation processes," in *IEEE International Conference on E-Commerce (CEC 2003)*, pp. 84–91, 06 2003.

[63] M. Ayatollahzadeh Shirazi and A. Barfouroush, "A Conceptual Framework for Modeling Automated Negotiations in Multiagent Systems," *Negotiation Journal*, vol. 24, no. 1, pp. 45–70, 2008.

[64] A. Dan, H. Ludwig, and G. Pacifici, "Web service differentiation with service level agreements," *White Paper, IBM Corporation*, 2003.

[65] M. Becker, N. Borrisov, V. Deora, O. Rana, and D. Neumann, "Using k-Pricing for Penalty Calculation in Grid Market," in *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pp. 97–97, Jan. 2008.

[66] R. Jurca and B. Faltings, "Reputation-Based Service Level Agreements for Web Services," *Service-Oriented Computing - ICSOC 2005*, pp. 396–409, 2005.

[67] O. Rana, M. Warnier, T. Quillinan, and F. Brazier, "Monitoring and Reputation Mechanisms for Service Level Agreements," *Grid Economics and Business Models*, pp. 125–139, 2008.

[68] J. Kosinski, D. Radziszowski, K. Zielinski, S. Zielinski, G. Przybylski, and P. Niedziela, "Definition and Evaluation of Penalty Functions in SLA Management Framework," *Networking and Services, International conference on*, pp. 176–181, 2008.

[69] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 151–158, ACM, 1971.

[70] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[71] C. Lee, "Representation of switching circuits by binary decision diagrams," *Bell System Technical Journal*, no. 38, pp. 985–999, 1959.

[72] S. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, pp. 509–516, June 1978.

[73] R. Ebendt, R. Drechsler, and G. Fey, *Advanced BDD optimization*. Springer, 2005.

[74] C. E. Shannon, "A symbolic analysis of relay and switching circuits," *Trans. AIEE*, no. 57, pp. 713–723, 1938.

[75] P. Bhoj, S. Singhal, and S. Chutani, "SLA management in federated environments," *Computer Networks*, vol. 35, no. 1, pp. 5 – 24, 2001.

[76] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, 1992.

[77] S. Friedman and K. Supowit, "Finding the optimal variable ordering for binary decision diagrams," *IEEE Transactions on Computers*, vol. 39, pp. 710–713, May 1990.

[78] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *ICCAD '93: Proc. 1993 IEEE/ACM international conference on Computer-aided design*, pp. 42–47, IEEE Computer Society Press, 1993.

[79] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour, "Establishing and Monitoring SLAs in Complex Service Based Systems," *IEEE International Conference on Web Services*, pp. 783–790, 2009.

[80] W. Binder, I. Constantinescu, and B. Faltings, "Scalable Automated Service Composition Using a Compact Directory Digest," *Database and Expert Systems Applications*, pp. 317–326, 2006.

[81] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams," in *ICSE '01: Proc. 23rd International Conference on Software Engineering*, pp. 443–452, 2001.

[82] "JavaBDD." http://javabdd.sourceforge.net/ (Last retrieved: 09/2010).

[83] A. S. McGough, A. Akram, L. Guo, M. Krznaric, L. Dickens, D. Colling, J. Martyniak, R. Powell, P. Kyberd, C. Huang, C. Kotsokalis, and P. Tsanakas, "GRIDCC: A Real-time Grid workflow system with QoS," *Scientific Programming*, vol. 15, no. 4, pp. 213–234, 2007.

[84] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 3, pp. 281–308, 2004.

[85] D. Colling, T. Ferrari, Y. Hassoun, C. Huang, C. Kotsokalis, A. S. McGough, E. Ronchieri, Y. Patel, and P. Tsanakas, "On Quality of Service Support for Grid Computing," *Grid Enabled Remote Instrumentation*, pp. 313–327, 2009.

[86] "OGF GLUE-WG." http://forge.ogf.org/sf/projects/glue-wg (Last retrieved: 09/2010).

[87] A. Sim, A. Shoshani, et al., "The Storage Resource Manager Interface Specification v2.2." http://sdm.lbl.gov/srm-wg/doc/SRM.v2.2.pdf (Last retrieved: 09/2010).

[88] M. Botts, "Sensor Model Language (SensorML) for In-situ and Remote Sensors," tech. rep., Open Geospatial Consortium, 2004.

[89] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke, "SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems," in *Job*

Scheduling Strategies for Parallel Processing, vol. 2537 of *Lecture Notes in Computer Science*, pp. 153–183, Springer Berlin / Heidelberg, 2002.

[90] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, and A. Chien, "Efficient resource description and high quality selection for virtual grids," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 1, pp. 598–606, 2005.

[91] G. P. Koslovski, P. V.-B. Primet, and A. S. Charão, "VXDL: Virtual Resources and Interconnection Networks Description Language," in *Networks for Grid Applications*, vol. 2 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 138–154, Springer Berlin Heidelberg, 2009.

[92] I. Wegener, *The complexity of Boolean functions*. Teubner, 1987.

[93] C. Gröpl, *Binary decision diagrams for random boolean functions*. PhD thesis, Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin, 1999.

[94] L. Epstein and M. Levy, "Dynamic multi-dimensional bin packing," *Journal of Discrete Algorithms*, 2010 (Currently under publication).

[95] T. Gonzalez, ed., *Handbook of approximation algorithms and metaheurististics*. Chapman & Hall, 2007.

[96] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, *Approximation algorithms for bin packing: a survey*. Boston, MA, USA: PWS Publishing Co., 1997.

[97] H. N. Van, F. D. Tran, and J.-M. Menaud, "SLA-Aware Virtual Resource Management for Cloud Infrastructures," *Computer and Information Technology, International Conference on*, vol. 1, pp. 357–362, 2009.

[98] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (New York, NY, USA), pp. 41–50, ACM, 2009.

[99] H. Nakada, T. Hirofuchi, H. Ogawa, and S. Itoh, "Toward Virtual Machine Packing Optimization Based on Genetic Algorithm," in

Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living, vol. 5518 of *Lecture Notes in Computer Science*, pp. 651–654, Springer Berlin / Heidelberg, 2009.

[100] Y. Ajiro and A. Tanaka, "Improving packing algorithms for server consolidation," in *Proc. Computer Measurements Group 2007 International Conference*, 2007.

[101] M. A. S. Netto, K. Bubendorfer, and R. Buyya, "SLA-Based advance reservations with flexible and adaptive time QoS parameters," in *Service-Oriented Computing – ICSOC 2007*, vol. 4749 of *Lecture Notes in Computer Science*, pp. 119–131, Springer Berlin / Heidelberg, 2010.

[102] I. Goiri, F. Julià, J. Ejarque, M. De Palol, R. Badia, J. Guitart, and J. Torres, "Introducing Virtual Execution Environments for Application Lifecycle Management and SLA-Driven Resource Distribution within Service Providers," in *8th IEEE International Symposium on Network Computing and Applications*, pp. 211–218, IEEE, 2009.

[103] S. Roy and N. Mukherjee, "Adaptive Execution of Jobs in Computational Grid Environment," *Journal of Computer Science and Technology*, vol. 24, pp. 925–938, September 2009.

[104] D. Ardagna, M. Trubian, and L. Zhang, "SLA based resource allocation policies in autonomic environments," *Journal of Parallel and Distributed Computing*, vol. 67, no. 3, pp. 259 – 270, 2007.

[105] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems*. Springer Verlag, 2004.

[106] "PuLP-OR Modeler." http://code.google.com/p/pulp-or/ (Last retrieved: 09/2010).

[107] "GUROBI Solver." http://gurobi.com/ (Last retrieved: 09/2010).

[108] H. Mittelmann, "Mixed Integer Linear Programming Benchmark (parallel codes)." http://plato.asu.edu/ftp/milpc.html (Last retrieved: 09/2010).

[109] "Performance Profile Summary." http://www.coin-or.org/GAMSlinks/benchmarks/MIP/bestSolver_100209/profile_summary.htm (Last retrieved: 09/2010).

[110] Y. Song, H. Wang, Y. Li, B. Feng, and Y. Sun, "Multi-Tiered On-Demand Resource Scheduling for VM-Based Data Center," in *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, (Washington, DC, USA), pp. 148–155, IEEE Computer Society, 2009.

[111] L. Phan, Z. Zhang, S. Jain, G. Tan, B. Loo, and I. Lee, "Cost-based Dynamic Job Rescheduling: A Case Study of the Intel Distributed Computing Platform," tech. rep., University of Pennsylvania and Intel Corporation, 2010.

[112] H. Lau and M. Lim, "Multi-period multi-dimensional knapsack problem and its applications to available-to-promise," in *Proceedings of the International Symposium on Scheduling (ISS), Hyogo, Japan*, pp. 94–99, Citeseer, 2004.

[113] T. Kelly, "Generalized Knapsack Solvers for Multi-unit Combinatorial Auctions: Analysis and Application to Computational Resource Allocation," in *Agent-Mediated Electronic Commerce VI*, vol. 3435 of *Lecture Notes in Computer Science*, pp. 73–86, Springer Berlin / Heidelberg, 2005.

[114] M. Bezem, J. Klop, and R. de Vrijer, *Term rewriting systems*. Cambridge Univ Pr, 2003.

[115] OASIS, "Web Services Resource Framework (WSRF) v1.2." http://www.oasis-open.org/committees/wsrf/, 04 2006 (Last retrieved: 09/2010).

Index

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text, and a list of references is given.

Konstantinos (Costas) Kotsokalis