SLA-based Resource Provisioning for Management of Cloud-based Software-as-a-Service Applications

by

Linlin Wu

Submitted in total fulfillment of
the requirements for the degree of

Doctor of Philosophy

Cloud Computing and Distributed Systems Laboratory
Department of Computing and Information Systems
The University of Melbourne, Australia

March 2014

SLA-based Resource Provisioning for Management of Cloud-based Software-as-a-Service Applications

PhD Candidate: Linlin Wu

Principle Supervisor: Professor Rajkumar Buyya

Co-Supervisor: Dr. Saurabh Kumar Garg

Abstract

The Cloud computing Software-as-a-Service (SaaS) model has changed the sales model for software providers. The SaaS model transforms the traditional license based model to a subscription model, which allows customers to access applications over the Internet without software and hardware upfront costs and provides reduced maintenance costs. However, the key for sales is still customer satisfaction which is at the heart of the selling process. To guarantee Quality of Service (QoS) for customer satisfaction therefore, the Service Level Agreement (SLA) is implemented between customers and SaaS providers, where the main objectives are profit maximization and increased market share.

To achieve these objectives, there are several challenges due to the dynamic nature of the Cloud environment. Firstly, the SaaS provider utilizes shared infrastructure and various types of request loads which can lead to unpredictability in performance and availability of resources. Secondly, there is a possibility that existing customers may make changes in requirements, which can lead to resource reallocation. As such, resource allocation may cause SLA violations which could reduce the SaaS providers' profit margin and reputation, meaning a possible loss of existing customers and potential new customers. Thirdly, SaaS providers need to attract customers with special needs and consider market competition from other providers in order to increase profit and market share.

To overcome the above challenges, most proposed solutions are focused on the resource management with the aim of minimizing cost without sufficiently consideration of customer' needs. Therefore, to address these challenges, this thesis proposes algorithms and techniques for optimal provisioning of Cloud resources with the aim of maximizing profit and customer base by handling the dynamism associated with SLAs and heterogeneous resources.

The **key contributions** of the thesis are:

- A comprehensive survey of how SLAs are created, managed and used with case examples drawn from both academy and industry with a major emphasis on the SLA-based resource management systems.
- The admission control and scheduling algorithms assist in identifying which request is more acceptable based on profitability, reducing the probability of SLA violations given the heterogeneous nature of Cloud resources.
- The customer requirements driven resource provisioning algorithms can help in adapting to changes in the requirements. The proposed algorithms provide personalized attention to the customer and are also able to understand specific customer needs.
- A new negotiation framework to enlarge a SaaS provider's customer base that considers dynamism in the Cloud environment with time and market factors to make the best possible decisions for negotiation.
- A prototype of the customer requirements driven SLA-based resource management system to prove the usefulness of our proposed strategies using the latest technologies.

This is to certify that

(i) the thesis comprises only my original work,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of table, maps, bibliographies, appendices and footnotes.

Signature _____

Date _____

ACKNOWLEDGMENTS

Throughout my PhD journey, I received guidance, support and motivation from amazing people whom I wish to acknowledge. First and foremost, I would like to express my sincere gratitude to my supervisors Professor Rajkumar Buyya and Dr. Saurabh Kumar Garg for their continuous support, advice, and guidance throughout my candidature. These individuals have built and directed an environment that granted me the opportunity to learn and practice research skills, meet and collaborate with brilliant researchers, and transfer the long journey of the PhD into an immensely rewarding experience. This was especially so when I encountered personal issues, and they supported me as a family.

I also wish to extend my gratitude to the members of the PhD committee: Prof. Rao Kotagiri and Dr. Rodrigo N. Calheiros for their encouragement and insightful comments in relation to my research. In particular, it has been consistently beneficial to discuss initial research ideas with Dr. Rodrigo. Dr. Rodrigo has also generously assisted both in preparing for my experiments and in the proof-reading of my papers and thesis.

I would also like to thank the past and present members of the CLOUDS Laboratory at the University of Melbourne. They include Mohsen Amini, Anton Beloglazov, Atefe Khosravi, Sare Fotouhi, Deepak Poola, Mohammed Alrokayan, Yaser Mansouri, Marco Netto, Mustafi zur Rahman, Mukaddim Pathan, Suraj Pandey, Rajiv Ranjan, Christian Vecchiola, and Marcos Dias de Assuncao. I would also like to thank Dr. Steve Versteeg and Mr. Bevan Mailman for proof-reading this thesis, and for their extensive comments.

It has been a great pleasure and a privilege to work with you all. I wish to acknowledge the Australian Federal Government, the University of Melbourne, the School of Engineering, the Australian Research Council (ARC), Computer Associates (CA), IEEE Victoria, Google, and CLOUDS Laboratory for granting scholarships and the travel support which enabled me to pursue doctoral study and attend international conferences.

Finally, I would like thank my family members including parents, my sister and my parents-in-law for their support and love.

Linlin Wu
Melbourne, Australia
March 2014.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

A vision for delivering "computing as a utility" was introduced in 1969 by Leonard Kleinrock, the chief scientist of the original Advanced Research Project Agency (ARPA) project. Kleinrock envisioned that computer networks would be used as a "utility" [1]. From 1969, Information and Communication Technology (ICT) has made many advances in various areas to make this vision a reality [2]. The advances in networked computing environments have transformed computing to a model consisting of services that can be commoditized and delivered similarly to utilities such as water, electricity, gas, and telephony [3]. In the utility computing model, consumers can access services on-demand according to their requirements regardless of where they are hosted.

The utility computing model can be used as a new outsourcing service model that can bring extensive opportunities and benefits for ICT users. The foremost advantage is the decrease of IT-related costs and complexities, because enterprises no longer need to invest heavily on or maintain their own computing infrastructure, and are not constrained to specific computing service providers. Furthermore, this model benefits small businesses lacking working capital. Hence utility computing provides businesses with greater flexibility and resilience, and more efficient utilisation of resources at lower operating and maintenance costs. Indeed, enterprises simply need to pay for resource usage as required the computing service providers.

Today this outsourcing model has emerged in the form of Cloud computing, which promises elastic resources to the consumers (customers) [4]. Cloud computing is considered a solution for challenges, such as licensing, distribution, configuration, and operation of enterprise applications associated with the traditional IT infrastructure, software sales, and deployment models. A layered architecture for Cloud services is shown in *Figure 1.1*. From bottom to top, the Infrastructure as a Service (IaaS) layer is a resource provisioning model where a provider offers infrastructure resources like hardware, storage, servers, and networking components on demand to consumers. The Platform as a Service (PaaS) layer offers a computing platform and solution stack as a service. It includes application development tools and execution

management services. The Software as a Service (SaaS) layer licenses a software application to customers as a service on demand using PaaS layer functionalities, such as resource management and IaaS layer resources.

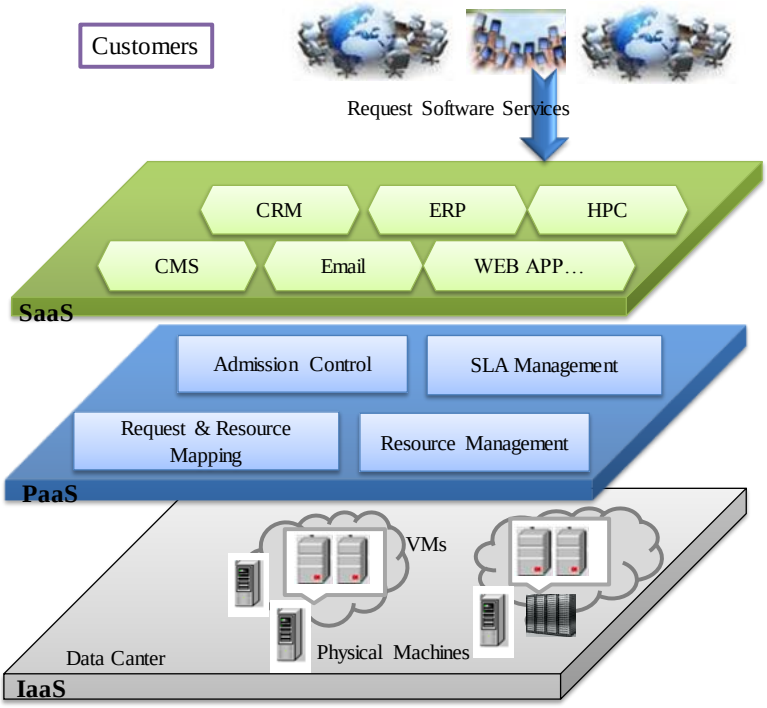


Figure 1.1 A layered architecture for Cloud computing

1.1 SaaS Model

Prior to the Cloud, the ICT administration tasks were comparatively easy since the single important objective of resource provisioning was the performance, such as the time spent on resource provisioning for web-based application [115]. Over time, the complexity of applications has grown, increasing the difficulties in their administration. Accordingly, enterprises have realized that it is more efficient to outsource some of their applications to third-party SaaS providers enabled by Cloud computing due to the following reasons [110]:

- It reduces the maintenance cost, because along with the growth in the complexity, the level of sophistication required to maintain the system has increased dramatically.
- By using SaaS, enterprises do not need to invest in expensive software licenses and hardware upfront before knowing the business value of the solution.

Therefore, by moving to the SaaS model customers benefit from continuously maintained software. The complexity of transitioning to new releases is managed transparently by SaaS providers, who pursue profit maximization by minimizing cost and enlarging market share by accepting more profitable requests and improving the Customer Satisfaction Level (CSL).

There are two design patterns for SaaS layers. The first one is the one presented in Figure 1.1, with three layered architecture using virtualized resources. This is the focus of this thesis. The second alternative utilizes dedicated software on physical servers that share resources between users. These two patterns sharing resources for multiple users are called multi-tenancy.
However, customer satisfaction is a crucial success factor to excel in the service industry, as highlighted by Yeo and Buyya [62]. The way to ensure the QoS is to define a legal contract, which is SLA (Service Level Agreement), between a service provider and a consumer [21]. In general, a customer requests web-based application services from a SaaS provider by agreeing with the QoS requirements specified in the SLA. When the SaaS provider can guarantee the SLA, the customer is satisfied. If the level of service is better than the specified in the SLA, the customer satisfaction level will be more than satisfied.

1.1.1 SaaS and Service Level Agreements

SLAs can be traced back to 1980s in telecommunication companies. As an example, telecommunication companies include an SLA within the terms of their contracts with customers to define the level(s) of service being sold in plain language terms. The SLA typically identifies parties who engage in the business processes and specifies the minimum expectations and obligations between them [21].

In Cloud computing, generally service providers define a publically published common SLA for all their customers. For instance, Microsoft promises to guarantee at least 99.9% availability in the SLA of the Microsoft Azure backup service. The SLA is established and commenced automatically when a customer requests service with confirmed payment. If any clauses in the SLA are violated, the penalty should be enforced, such as the granting of more credit for future services to the customer.

Two typical types of SLA are **provider predefined** and **negotiated** SLAs. The provider predefined SLA provides a generic SLA template for all customers. For example, Amazon EC2 has a predefined static SLA. However, customers may have special QoS requirements which may not be included in a predefined SLA. In this case, the customer and the provider will go through negotiation processes to achieve a mutually agreed SLA (Negotiated SLA). In order to ensure the agreed SLA, SaaS providers require strategies to manage resources to satisfy the QoS specified in SLA without deteriorating their profit.

Several researchers have satisfied these requirements by providing SLA-based resource management mechanisms [72][69] and negotiation strategies [152][153]. There are still several challenges for resource management, but the key issue for SaaS providers in Cloud is

how to optimize resource provisioning, which aims at improving the utilization of cloud systems in order to achieve profit maximization and market share enlargement. More details on the SLA-based resource management are discussed along with their limitations in the following section.

1.2 SLA-based Resource Management for SaaS

Resource management is a central and the most challenging task in Cloud computing, particularly when there is a legal document specified in the form of SLA, which contains QoS requirements. There are several problems to consider while managing resources given SLAs, such as, type of resource required, mapping, provisioning, allocation, adaptation, and brokering. The basic responsibility of a Resource Management System (RMS) is to accept requests from customers and then map them to the available resources, provision the matched resources, and allocate them to the customer. In practice, due to the heterogeneous and dynamic nature of Cloud environments, the RMS needs to be able to adapt to the heterogeneity from resource side and dynamic changes from customer sides. In general, there are two types of resources for SaaS - physical and logical. For example, data centres, physical machines, network elements are physical resources, on the other hand, Virtual Machines (VMs) and energy are logical resources.

Research on SLA-based market driven resource management started in 1980s [72][69]. However, the SaaS Cloud model has brought into view new challenges that have not been addressed before. As Professor David Patterson of the University of California, Berkeley, illustrates, the challenges faced by software developers currently, "*There are dramatic differences between developing software for millions to use as a service versus distributing software for millions to run their PCs*" [5].

One of the challenges is dealing with heterogeneous geographically distributed resources with different usage policies, price models, availability and performance patterns and varying loads. Moreover, the SaaS service providers and customers have different goals, objectives, strategies, and requirements. Resource sharing becomes further complicated in SaaS Cloud due to the self-interested nature of customers. In addition, each customer includes multiple user accounts, with different requests. Therefore, SLA-based resource management involved in delivering software as a service for millions of customers in Cloud environments is much more complex compared to just distribute software [6].

As mentioned before, the goal of SaaS providers are twofold i.e. maximizing profit and

enlarging the customer base by offering better services. To achieve these goals, SaaS providers employ different techniques, such as utilizing internal hosted resources of private data centres or renting resources from an IaaS provider to guarantee the SLA. For example, Saleforce.com [102] hosts resources, but Animoto rents resources from Amazon EC2 [92]. However, the main challenge for SaaS providers to achieve these goals is how to manage these resources efficiently ensuring SLA specified QoS requirements. Several research works have explored this topic to a certain degree [121][122][127][42]. However, still there is a long way to go for achieving SaaS providers goals as depicted below.

1.2.1 Limitation of Existing Solutions

The current resource management techniques for SaaS in Cloud mainly focus on either minimizing the number of VMs without considering SLA or only consider limited QoS parameter such as availability only. In contrast, most of these resource management techniques need to be extended to include the dynamic, diverse and competitive nature of participants with conflicting Quality of Service (QoS) requirements in Cloud.

In a shared resource infrastructure such as Cloud, the heterogeneous nature of resources and self-interested nature of customers can lead to problems, where every customer acquires as many types of software as possible because there is no incentive for customers to back off during times of high demand. The self-interested customers, in turn, over exploit the service by degrading the SaaS provider's ability to deliver the required service to all customers using heterogeneous resources. Therefore, resource management needs to be SLA-based, which can regulate the supply and demand of resources at peak usage time.

In order to meet the above requirements, most of the SLA-based resource management methods are either non-profit based [6] or designed for a fixed number of resources, such as FirstPrice [48] and FirstProfit [70]. To resolve the problem caused by customers' self-interest nature and conflicting interests between customer requests, admission control and scheduling was proposed as a solution[70][90][91], such as learning-based admission control in Cloud [67]. However, these works do not target profit maximisation and an increase in market share simultaneously.

SaaS providers aim to optimally provision resources so that service costs can be minimized. In general, SaaS providers utilize internal resources of its data centres or rent resources from a specific IaaS provider to guarantee SLA. For SaaS providers, in-house hosting resources can generate administration and maintenance cost while renting resources from IaaS providers can impact the service quality offered to SaaS customers due to performance variability [103].

Several profit-driven resource management solutions are proposed by minimizing the number of resources [121][122][127][42]. However, these works did not consider customer satisfaction level related QoS parameters.

To satisfy the customer requirements, customer side QoS parameters are essential. However, most of the current works consider provider side QoS parameters, such as price [105][127]. Although some work consider customer side QoS parameters, some SaaS layer related QoS parameters are missing, such as software response time [128][65].

Several projects are related at different degrees to the SLA-aware management of resources, such as SLA@SOI [182], Claudia [176], BonFIRE [179], Optimis [177], 4CaaSt [178] and Cloud-TM [180]. However, SLA@SOI does not consider Cloud computing infrastructures as their target platform, and hence it does not account for some specific needs in this area. Claudia [176], BonFIRE and 4CaaSt [178] do not consider management of heterogeneous resources. Although Optimis [177] does scheduling for resource management and PaaSage [181] provides runtime monitoring and dynamic adaptation, they do not cover SaaS level parameters, such as service response time.

Cloud-TM [180] cannot be applied to general purpose Cloud computing, since it is focused on datacentric Cloud applications. In the context of the resource allocation algorithms for enterprise applications, evolutionary algorithms, such as Genetic Algorithm (GA) have been used [111]. As evolutionary algorithms create a pre-planning schedule, they are not able to deal with dynamic environments such as Cloud.

Therefore, these approaches are not suitable for SLA-based resource management in dynamic Cloud environments to achieve the goal of maximizing profit and customer base for SaaS.

1.3 Problem Statement and Objectives

This thesis focuses on the following problem:
How to design and develop algorithms and techniques that help in maximizing profit and market share for Cloud SaaS providers, who lease applications to customers by using Cloud resources and simultaneously handle dynamism and variations associated with SLAs and available resources.
In the context of the problem, the two key stakeholders are (1) SaaS providers and (2) SaaS customers. A model/architecture that depicts key components of SaaS Cloud is shown in *Figure 1.1*. The model consists of application layer and platform layer functions. Customers

request the software service with their QoS requirements to application layer. The platform layer is responsible for application development and deployment (such as Aneka [107]). In our model, this layer includes the admission control function to analyse the customer's QoS parameters and decide whether to accept or reject the request. The request and resource mapping function is responsible for translating the customer side QoS requirements to infrastructure level parameters. Based on admission control decision, the resource management component is responsible for provisioning and allocating resources. Furthermore, the SLA management is required since we consider SLA with customers. For some customers with special requirements, which are different from what is publically offered by SaaS providers, a negotiation process is required for SLA establishment.

In dynamic Cloud environments, several issues that need to be addressed to solve the problem are:

- Can a new request be accepted without impacting accepted requests using distributed and heterogeneous resources, whose capabilities, availabilities and performance (such as service time) can change very frequently?
- How to deal with the resource level heterogeneity (such as service initiation time)?
- How to map various customer requests with different QoS parameters to the resources?
- How to manage dynamic customer demands? (such as upgrading from a standard product edition to an advanced product edition or adding more accounts)
- How to design the negotiation related processes and decision-making strategies to fulfil special customer requests?

1.3.1 Challenges and Requirements

Answering the questions above is not trivial considering the various dynamic and variety of factors associated with Cloud environments and actors. Cloud environments give access to heterogeneous resources having different price schemas and performance capabilities and that can be dynamically expanded and contracted on demand. Each customer has his own requirement in terms of services and QoS which can also change dynamically. This brings several challenges and requirements for the SaaS provider in order to manage their resources in a profitable manner.

To accept any customer request, SaaS providers need to ensure the minimum level of service specified in SLA is delivered to the customer using heterogeneous Cloud resources. Currently, most SaaS providers use VMs to host their software services and these VMs in general sharing

a common physical server with other VMs hosting similar or different software services. The challenge comes from unpredictability of the software services performance which is dependent on the unknown configuration of underline physical server and variation in other VMs resource usage. This can lead to SLA violation or revenue loss when the resource performance degradation causes the breach of the minimum level of service requirements specified in the SLA. SaaS providers need to consider which customer request is more profitable to accept given this heterogeneous nature of Cloud resources. Therefore we need new admission control and scheduling strategies that take care of these factors.

Once a customer request is accepted there is always a possibility of changes in requirement, since the SaaS provider is expected to scale up and out accordingly. When the customer changes his/her requirement, resources have to be dynamically reallocated according to the customer's on-demand requirements. Moreover, while allocating/reallocating resources the SaaS provider has to minimize the impact on existing customers while satisfying the customers' requirement changes. Therefore, new adaptive customer requirements driven resource management algorithms considering customer profile and the providers' quality parameters are required.

As discussed, SaaS providers want to expand their customer base. Therefore, they need to provide more flexibility in terms of service to cater to variations associated with individual customer requirements. This is generally done through a negotiation process between customers and the service providers. However, while undertaking negotiations, the service provider needs to take into consideration not only what they can provide to customers but also the competition with other SaaS providers. Thus, new negotiation frameworks are needed for SaaS providers that consider the dynamism in the Cloud environment with time and market factors to make best possible decisions. In summary, we identified three sub objectives to align with maximizing profit and market share for SaaS:

- To design SLA-based admission control and scheduling algorithms that differentiate customer requests based on the heterogeneous resource capability to minimize cost and SLA violations by accepting more profitable requests.
- To investigate adaptive SLA-based resource provisioning algorithms according to customer requirements changes by considering more customer factors that provide personalized attention to customers which include considering customer profiles and understanding customer specific needs.

- To investigate the architectural model for automated SLA negotiation framework to establish SLA between SaaS and customers, whose requirements are not covered by existing SaaS predefined static SLA.

In this thesis, we propose a solution that meets these objectives.

1.3.2 Proposed Solution

As discussed above, SaaS providers need to deal with the heterogeneity and variety from both the resource providers' side and the customers' side. To solve the problem as stated in the previous section, we consider the following example scenarios of SaaS to achieve the specified objectives.

SaaS providers lease web-based software as services to customers and use either 3rd party resources (such as Virtual Machines from Amazon) or in house hosted resources. Take Animoto as a SaaS example, it creates videos based on the customer uploaded pictures or videos with selected themes. Three simple steps, 1) customers upload pictures or videos; 2) customers select style, text, music to generate video; 3) customers download or share generated video [108]. In this service application model, different customers will submit their request at any time with different QoS parameters, such as different file size from customer side impact the resource management for SaaS providers. Therefore, this thesis focuses on the dynamism in terms of resource availability and capability caused by the variety of customer requests and resource heterogeneities. Admission control algorithms are proposed employing different strategies to accept more profitable requests for minimal performance impact, avoiding SLA penalties for existing customer requests that decrease the SaaS provider's profit and the customer satisfaction level. The scheduling algorithms determine where and which type of resource should be used by incorporating the heterogeneity of IaaS providers in terms of QoS factors, such as price, service initiation time, and data transfer time.

Another SaaS application model is enterprise application, which is required for day to day business. For instance, Microsoft sales Office365 software packages with three product editions (for example, small business, small business premium and midsize business) and each product edition has a fixed price. The existing customer may require an upgrade in their service by adding additional user accounts or an upgrade of the software edition at any time. In practice, the SaaS provider has to handle these on-demand customer requests in line with the SLA. Hence, to achieve SaaS providers' objectives, we minimize total cost and improve customer satisfaction levels in two ways: 1) minimizing SLA violations and 2) improve service quality. Our work further investigates the dynamic changes in customer requirements with the consideration of customer profile to pay more personalized attention to customers.

In terms of SLAs, the above two scenarios consider pre-defined SLAs, however, in many circumstances; some customers may request special services for special needs. For example, the Department of Education requires the Office 365 with a particular type of template for teachers and students to automatically provision the classes and lectures when they login the portal. In this case, the pre-defined SLA listed on the web site will not suit their requirements. Thus, our work proposes the automated SLA negotiation framework to maximize profit and enlarge market share for SaaS by considering two factors. Firstly, the dynamic nature of the Cloud, as service cost and quality are constantly changing and customers have varying needs. Secondly, time and market oriented resource allocation, as any delay incurred in waiting for a resource assignment is perceived as an overhead [145].

1.4 Contributions

This thesis makes the following research contributions towards the understanding and the advancement of SLA-based resource management in Cloud environments to achieve the goal of Cloud service providers:

1. It presents a comprehensive taxonomy and survey on SLAs and their creation, management, and usage in utility computing environments. It discusses existing use cases from Grid and Cloud computing systems to identify the level of SLA realization in state-of-art systems and emerging challenges for future research. The survey not only helps researchers to understand primary design factors and issues that are still outstanding and crucial but also provides insights for extending and reusing components of existing market-based Resource Management Systems (RMSs). Therefore, the survey can help in the design and implementation of more practical and enhanced SLA-based Cloud resource management systems in the near future. The SLA-based RMSs selected for the survey are primarily research works as they reflect the latest technological advances. The design concepts and architectures of these research-based RMSs are also well-documented in publications to facilitate comprehensive comparisons, unlike commercially released products by vendors.
2. It proposes admission control and scheduling algorithms for SaaS providers to effectively utilise heterogeneous Cloud resources to maximize profit by accepting more profitable customer requests using the least cost resources while minimizing the SLA violations for existing customers. It also conducts detailed performance analysis using trace-based simulation to highlight the effectiveness of managing the risk of inaccurate runtime estimates for various scenarios that includes varying workload,

deadline, budget, contract length, service initiation time, performance degradation, and inaccurate estimated high: low ratio.

3. Thesis proposes customers' requirements driven resource provisioning algorithms for SaaS providers who lease enterprise applications to customers. The proposed provisioning algorithms consider customer profiles and providers' quality parameters (e.g. response time) to handle dynamic customer requirement changes and infrastructure level heterogeneity by minimizing infrastructure and penalty cost. It also takes care of CSL by minimizing SLA violations and improving the quality of service (e.g. response time) expected by the customer. We also take into account customer-side parameters (such as the proportion of upgrade requests), and infrastructure-level parameters (such as the service initiation time) to compare algorithms. These algorithms are evaluated by extensive experimental study using data from a real Cloud.
4. It proposes a novel automated negotiation framework considering the SaaS Broker as the one-stop-shop for customers to efficiently get required services. The automated negotiation framework performs adaptive and intelligent bilateral bargaining of SLAs between SaaS brokers and SaaS providers including negotiation policies, protocols, and strategies. It proposes decision-making heuristics considering time, market constraints, and trade-off between different issues as well. These negotiation heuristics are evaluated by extensive experimental study of our prototype framework using data from real Cloud as detailed in particular chapters.
5. It details an implementation of SLA-based Resource Management System (SLARMS) to demonstrate the usefulness of the algorithms proposed in the thesis.

1.5 Methodology

We primarily evaluated the proposed algorithms using the CloudSim [80] simulator with workloads from real Cloud software systems, such as CloudMinder[1].

1.5.1 Workload

From the customer requests perspective, we adopted as workload data shared with us by the cloud provider CA Technologies, who offers a number of enterprise software solutions to customers delivered as SaaS [108]. The data provided includes the response, refresh and processing times of an enterprise solution hosted on VMs, as measured by the quality assurance team. As SaaS availability depends on the infrastructure availability, this

[1] CloudMinder is Software as a Service product from CA Technologies (Computer Associates).

information is collected from the CloudHarmony benchmarking system [156], which provides real data from Cloud providers.

In order to analyse technical challenges to manage resources, we performed experiments by collecting real data from both public Cloud infrastructures, such as Amazon EC2 [92], GoGrid [94], and private Clouds from industry, such as CA (Computer Associates) hosted private Cloud.

We modelled and adapted the workload data to meet the requirements of our experiments. In order to evaluate the proposed algorithms under different loads, we model request arrival rate using Poisson distribution similar to many previous works [100][101]. Similar as other works, we use a normal distribution to model all the other parameters (standard deviation = (1/2) x mean) that are not available from real workload.

1.5.2 Experiment System

CloudSim Toolkit [80] is used to model and simulate the proposed algorithms for resource management. We simulated data centres with physical machines whose configuration resembles public Cloud such as Amazon EC2 large image. We map a number of VMs of different types to physical machines. The general scheduling policy is time shared scheduling. We have extended the existing Cloud environment and added our algorithm for SLA-based resource management.

We also implemented a prototype system called Service Level Agreement Resource Management System (SLARMS) to validate and demonstrate the usefulness and practicality of the proposed algorithms and techniques. The details of experiment settings of our works will be explained throughout the thesis.

1.6 Organization

The rest of this thesis is organized as follows (*Figure 1.2*): Chapter 2 presents a comprehensive survey of how SLAs are created, managed and used in utility computing environments in practice. Chapter 3 proposes an admission control and scheduling algorithm that utilizes multiple resources to minimize the penalty cost of accepting a new request, which may violate the SLA objectives. Chapter 4 proposes customer driven SLA-based resource provisioning for web-based enterprise applications by minimizing the cost and the number of SLA violations. The proposed provisioning algorithms consider customer profiles and the providers' parameters to handle dynamic customer requests and infrastructure level

heterogeneity. Chapter 5 proposes a novel automated web-based negotiation framework considering the SaaS Broker as the one-stop-shop for customers to get required service efficiently. Chapter 6 describes an implementation of SLA-based Resource Management System to demonstrate the usefulness of the proposed algorithms. Chapter 7 concludes and provides directions for future work.

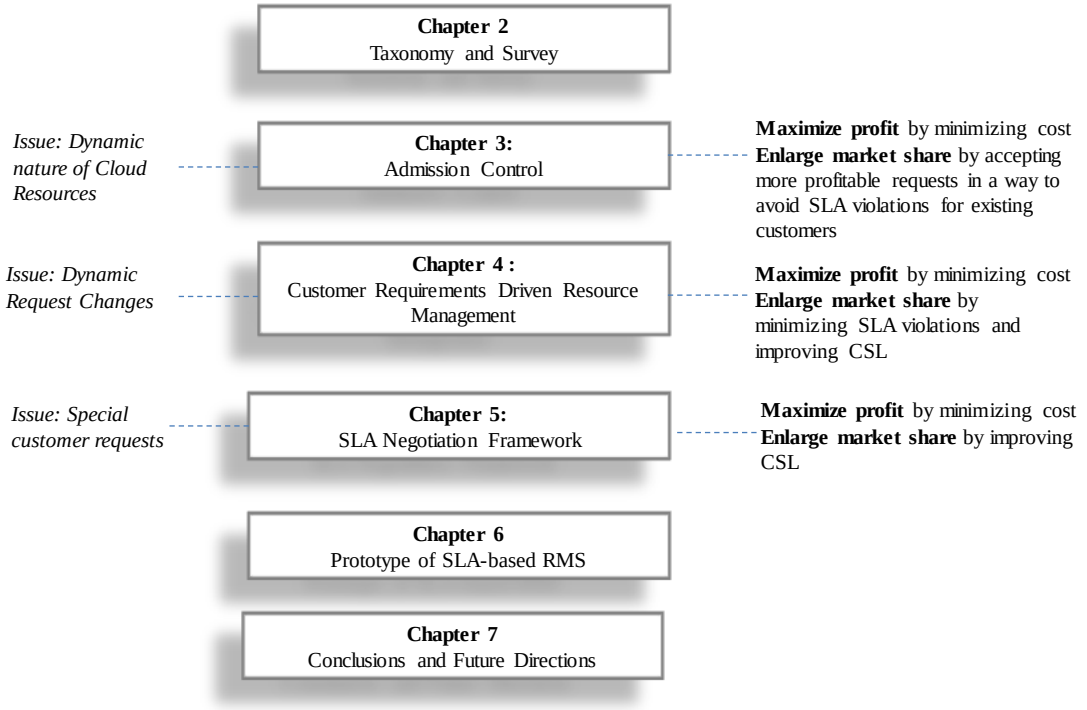


Figure 1.2 Thesis Organizations

The core chapters are derived from various research works that have been published during the course of candidature as detailed below:

• *Chapter 2* is derived from:

> **Linlin Wu** and Rajkumar Buyya, *Service Level Agreement (SLA) in Utility Computing Systems, Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, Pages: 1-25, V. Cardellini et al. (eds), ISBN: 978-1-60-960794-4, IGI Global, Hershey, PA, USA, July 2011.

• *Chapter 3* is derived from:

> **Linlin Wu**, Saurabh Kumar Garg, and Rajkumar Buyya, *SLA-based Admission Control for a Software-as-a-Service Provider in Cloud Computing Environments*, Journal of Computer and System Sciences, Volume 78, No. 5, Pages: 1280-1299, ISSN 0022-0000, Elsevier Science, Amsterdam, The Netherlands, September 2012.

• *Chapter 4* is derived from:

Linlin Wu, Saurabh Kumar Garg and Rajkumar Buyya, *SLA-based Resource Allocation for a Software as a Service Provider in Cloud Computing Environments*, Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011, IEEE CS Press, USA), Los Angeles, USA, May 23-26, 2011.

Linlin Wu, Saurabh Kumar Garg Steve Versteeg, and Rajkumar Buyya, *SLA-based Resource Provisioning for Software-as-a-Service Applications in Cloud Computing Environments*, IEEE Transactions on Services Computing (TSC), ISSN: 1939-1374, IEEE Computer Society Press, USA (in press, accepted on Oct. 11, 2013).

• *Chapter 5* is derived from:

Linlin Wu, Saurabh Kumar Garg, Rajkumar Buyya, Chao Chen, and Steve Versteeg, *Automated SLA Negotiation Framework for Cloud Computing*, Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2013, IEEE CS Press, Los Alamitos, CA, USA), Delft, the Netherlands, May 13-16, 2013.

2 Service Level Agreement (SLA) in Utility Computing Systems

This chapter presents a comprehensive survey of how SLAs are created, managed, and used in utility computing environments. We discuss existing use cases from Grid and Cloud computing systems with major emphasis on resource management to identify the level of SLA realization in state-of-art systems and emerging challenges for future research.

2.1 Introduction

As discussed before, utility computing [62] offers computing services on demand, thus it makes them consumed as other utilities, such as water, electricity, gas, and telephony. With this new service model, users no longer have to invest heavily on or maintain their own computing infrastructures, and they are not constrained to any specific computing service provider. Instead, they can outsource jobs to service providers and just pay for what they use. Utility computing has been increasingly adopted in many fields including science, engineering, and business [66]. Grid, Cloud, and Service-oriented computing are some of the paradigms that enabled delivery of computing as a utility. In these computing systems, different Quality of Service (QoS) parameters have to be guaranteed to satisfy user's request. A Service Level Agreement (SLA) is used as a formal contract between service provider and consumer to ensure service quality [21].

A typical utility computing system architecture is shown in *Figure 2.1* with the following components: the User/Broker, SLA Management, Service Request Examiner, and Resource/Service Provider. User or Broker submits its requests via applications to the utility computing system, which includes the bottom three layers. The Service Request Examiner is responsible for Admission Control. The SLA Management includes SLA establishment and

enforcement. The Resource Allocation component takes care of resources scheduling. Finally, the Resource or Service Provider offers resources or services.

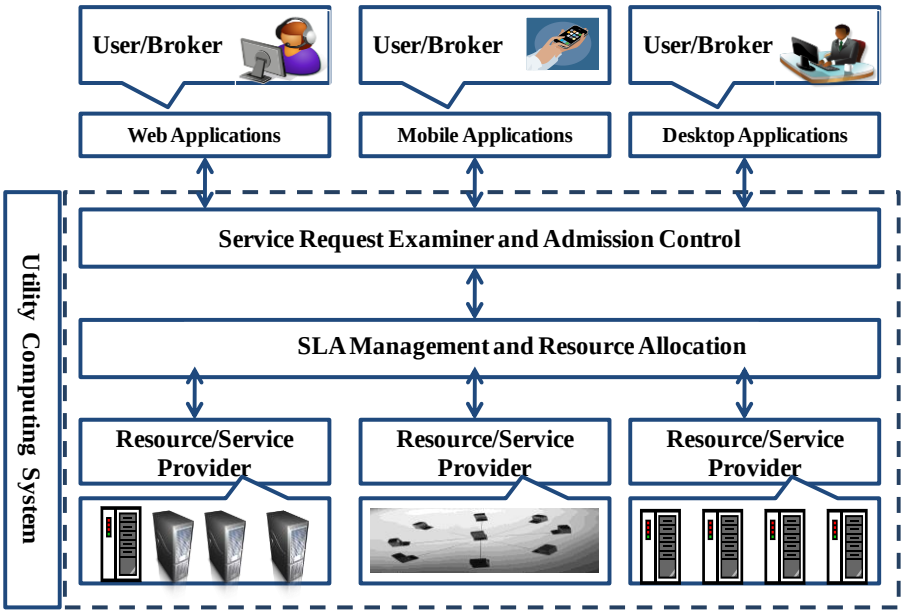


Figure 2.1 A typical architectural view of utility computing system

In the above architecture, SLAs are used to identify parties who engage in the electronic business, computation, and outsourcing processes and to specify the minimum expectations and obligations that exist between parties [21]. The most concise SLA includes both general and technical specifications, including business parties, pricing policy, and properties of the resources required to process the service [63]. According to Sun Microsystems Internet Data Center Group's report [54], a good SLA sets boundaries and expectations of service provisioning and provides the following benefits:

- **Enhanced customer satisfaction level:** A clearly and concisely defined SLA increases the customer satisfaction level, as it helps providers to focus on the customer requirements and ensures that the effort is put on the right direction.
- **Improved Service Quality:** Each item in an SLA corresponds to a Key Performance Indicator (KPI) that specifies the customer service within an organization.
- **Improved relationship between two parties**: A clear SLA indicates the reward and penalty policies of a service provision. The consumer can monitor services according to Service Level Objectives (SLOs), which are QoS items specified in the SLA. Moreover, the precise contract helps parties to resolve conflicts more easily.

A clearly defined lifecycle is essential for effective realization of an SLA. Ron, S. et al. [51] define SLA lifecycle in three high level phases, which are the ‘creation phase’, ‘operation phase’, and ‘removal phase’. Sun Microsystems Internet Data Center Group [54] defines a practical SLA lifecycle in six steps, which are ‘discover service providers’, ‘define SLA’, ‘establish agreement’, ‘monitor SLA violation’, ‘terminate SLA’, and ‘enforce penalties for violation’.

The realization of an SLA can be traced back to 1980s in telecommunication companies. Furthermore, the advent of Grid computing reinforces the necessity of using SLA [62]. Specifically, in service-oriented commercial Grid computing [22], resources are advertised and traded as services based on an SLA after users specify various levels of service required for processing their jobs [49]. However, SLAs have to be monitored and assured properly [52]. SLA management has been addressed partially by frameworks such as WS-Agreement [12] and WSLA [40].

Recently, Cloud computing has emerged as a new platform for delivering utility computing services. In Clouds, infrastructure, platform and application services are available on-demand and companies are able to access their business services and applications anywhere in the world whenever they need. In this environment, massively scalable systems are made available to end users as a service [20]. In this scenario, where both request arrival rate and resources availability continuously vary, SLAs are used to ensure that service quality is kept at acceptable levels.

This chapter reveals key design factors and issues that are still significant in utility computing platforms such as Grids and Clouds. It provides insights for extending and reusing components of the existing SLA management frameworks and it aims to be a guide in designing and implementing enhanced SLA-based management systems.

This chapter presents key use cases that reflect the latest technological advances. The design concepts and architectures of these works are well-documented in publications to facilitate comprehensive investigation.

The rest of the chapter is organized as follows: Utility architecture and SLA foundational concepts are summarized in Section 2.2. In Section 2.3, the key challenges and solutions for SLA management are discussed. SLA use cases are proposed in Section 2.4. The open problems

addressing some of the issues in current systems are presented in Section 2.5. Finally, the chapter concludes with the open challenges in SLA management in Section 2.6.

2.2 Utility Architecture and SLA Foundations

In this section, initially, a typical utility computing architecture is presented. SLA definitions from different areas are summarized in Section 2.2.2. SLA components are described in Section 2.2.3. In Section 2.2.4, two types of SLA lifecycle are presented and compared.

2.2.1 Utility Architecture

The layered architecture of a typical utility computing system is shown in *Figure 2.2*. From top to bottom it is possible to identify four layers, a **User or Broker** submits its requests using various applications to the utility computing system, the **Service Request Examiner** is responsible for admission control, **SLA Management** balances workloads, and a **Resource or Service Provider** offers resources or services. Users or Brokers, who act on users' behalf, submit their service requests of using applications, from anywhere in the world, to be processed by utility computing systems. When a service request is submitted, the Service Request Examiner (SRE) uses Admission Control mechanism to interpret request's QoS requirements before determining whether to accept or reject it after interacting with SLA Management component which is responsible for enforcing SLA. Thus, the SRE ensures that there is no overloading of resources whereby many service requests cannot be fulfilled successfully due to limited availability of resources/services.

The SLA Management component is responsible for resource allocation and consists of several components: Discovery, Negotiation/Renegotiation, Pricing, Scheduling, Monitoring, SLA Enforcement, Dispatching and Accounting. The Discovery component is responsible for discovering service providers that can satisfy user requirements. In order to define mutually agreed terms between parties, it is common to put in place price negotiation mechanisms or to rely on quality metrics. The Pricing mechanism decides how service requests are charged. Pricing serves as a basis for managing supply and demand of computing resources within the utility computing system, and facilitates in prioritizing resource allocations. Once the negotiation process is completed, the Scheduling mechanism uses algorithms or policies to decide how to map requests to resource providers. Then the Dispatching mechanism starts the execution of accepted service requests on allocated resources.

The Monitoring component consists of a Resource Monitoring mechanism and a Service Request Monitoring mechanism. The Resource Monitoring mechanism keeps track of the availability of Resource Providers and their resource entitlements. On the other hand, the Service Request Monitoring mechanism keeps track of the execution progress of service requests. The SLA enforcement mechanism manages violation of contract terms during the execution. Due to the SLA violation, sometimes Renegotiation is needed in order to keep ongoing trading. The Accounting mechanism maintains the actual usage of resources by requests so that the final cost can be computed and charged to the users. At the bottom of the architecture, there exists a Resource/Service Provider that comprises multiple services such as computing services, storage services and software services in order to meet service demands.

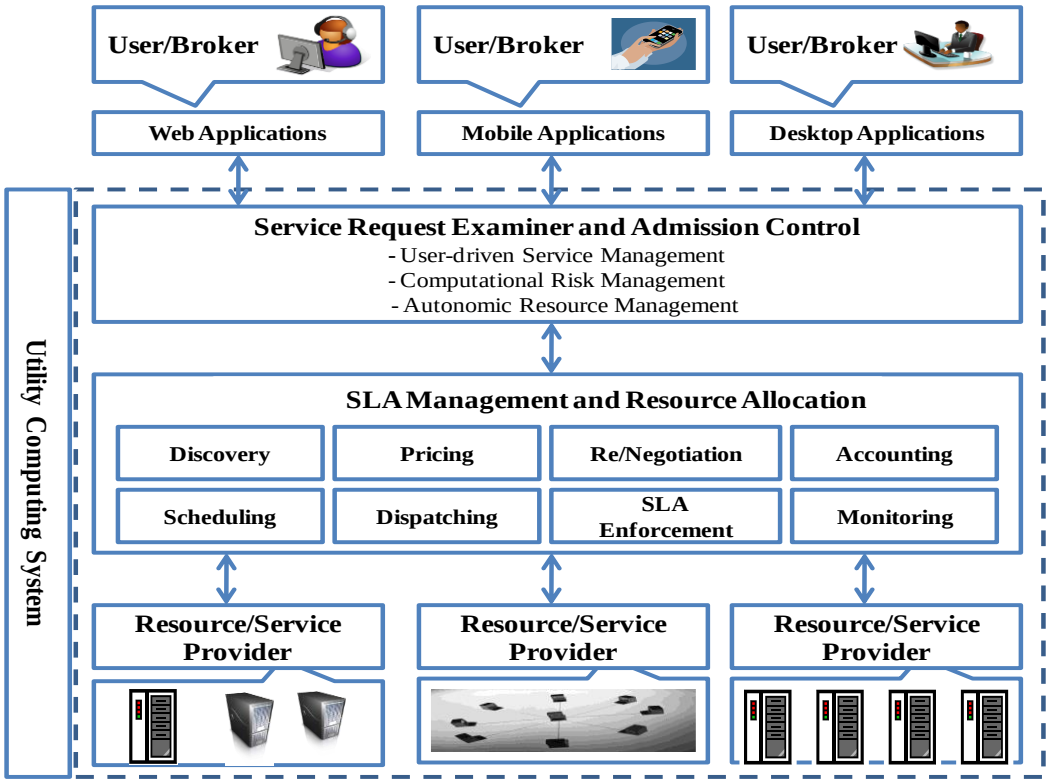


Figure 2.2 SLA-based Utility Computing System Architecture

2.2.2 SLA Definitions

Dinesh et al. [27] define an SLA as: "*An explicit statement of expectations and obligations that exist in a business relationship between two organizations: the service provider and customer*". Since SLA has been used since 1980s in a variety of areas, most of the available definitions are contextual and vary from area to area. Some of the main SLA definitions in Information Technology related areas are summarized in *Table 2.1*.

Table 2.1 Summary of SLA definitions classified by the area

Area	Definition	Source
Web Services	*"SLA is an agreement used to guarantee web service delivery. It defines the understanding and expectations from service provider and service consumer".*	HP Lab [36]
Networking	*"An SLA is a contract between a network service provider and a customer that specifies, usually in measurable terms, what services the network service provider will supply and what penalties will assess if the service provider cannot meet the established goals".*	Research Project
Internet	*"SLA constructed the legal foundation for the service delivery. All parties involved are users of SLA. Service consumer uses SLA as a legally binding description of what provider promised to provide. The service provider uses it to have a definite, binding record of what is to be delivered".*	Internet NG [51]
Data Center Management	*"SLA is a formal agreement to promise what is possible to provide and provide what is promised".*	Sun Microsystems Internet Data Center group [54]

2.2.3 SLA Components

An SLA defines the delivery ability of a provider, the performance target of consumers' requirement, the scope of guaranteed availability, and the measurement and reporting mechanisms [50].

Jin et al. [36] provided a comprehensive description of the SLA components, including: (*Figure 2.3*):

- **Purpose**: Objectives to achieve by using an SLA.
- **Restrictions**: Necessary steps or actions that need to be taken to ensure that the requested level of services are provided.
- **Validity period:** SLA working time period.
- **Scope**: Services that will be delivered to the consumers, and services that will not be covered in the SLA.
- **Parties**: Any involved organizations or individuals involved and their roles (e.g. provider and consumer).

- **Service-level objectives (SLO)**: Levels of services which both parties agree on. Some service level indicators such as availability, performance, and reliability are used.
- **Penalties:** If delivered service does not achieve SLOs or is below the performance measurement, some penalties will occur.
- **Optional services**: Services that are not mandatory but might be required.
- **Administration**: Processes that are used to guarantee the achievement of SLOs and the related organizational responsibilities for controlling these processes.

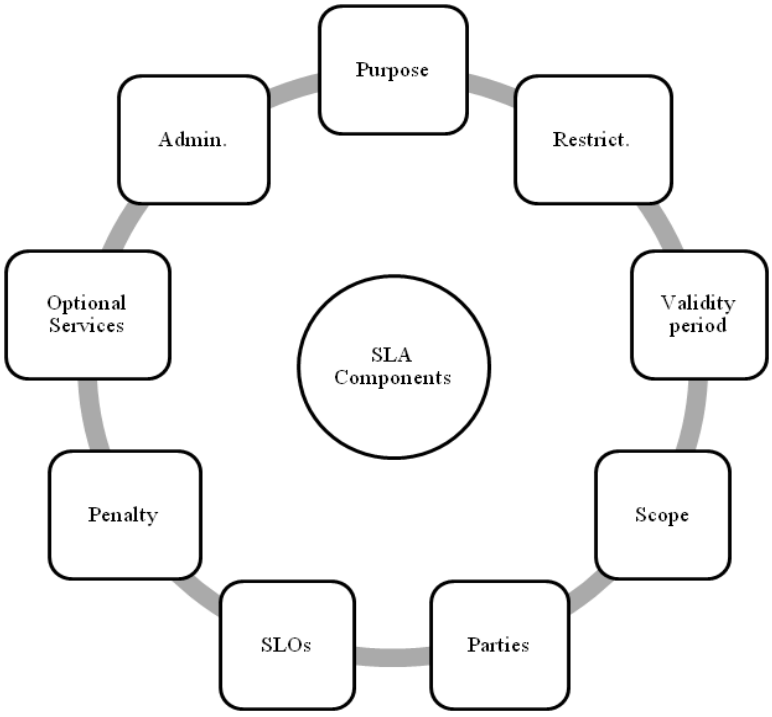


Figure 2.3 SLA Components

2.2.4 SLA Lifecycle

Ron et al. [51] define the SLA life cycle in three phases (*Figure 2.4*). Firstly, the **creation phase,** in which the customers find service provider who matches their service requirements. Secondly, the **operation phase,** in which a customer has read-only access to the SLA. Thirdly, the **removal phase,** in which SLA is terminated and all associated configuration information is removed from the service systems.

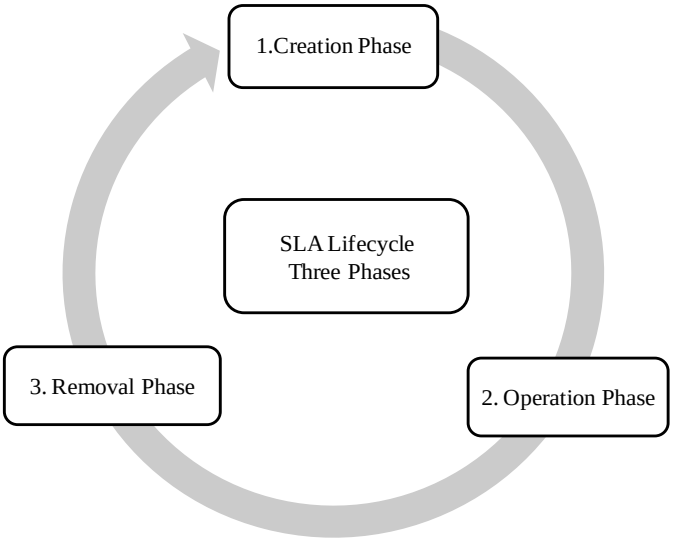


Figure 2.4 SLA high level lifecycle phases, according to the description of Ron et al. [51]

A more detailed life cycle has been characterized by the Sun Microsystems Internet Data Center Group [54] , which includes six steps for the SLA life cycle: the first step is **'discover - service providers'**, in where service providers are located according to consumer's requirements. The second step is **'define – SLA',** which includes definition of services, parties, penalty policies and QoS parameters. In this step it is possible to negotiate between parties to reach a mutual agreement. The third step is **'establish – agreement',** in which an SLA template is established and filled in by specific agreement, and parties are starting to commit to the agreement. The fourth step is **'monitor – SLA violation'**, in which the provider's delivery performance is measured against to the contract. The fifth step is **'terminate – SLA'**, in which SLA terminates due to timeout or any party's violation. The sixth step is **'enforce - penalties for SLA violation'**, if there is any party violating contract terms, the corresponding penalty clauses are invoked and executed. These steps are illustrated in *Figure 2.5*.

The mapping between three high level phases and six steps of SLA lifecycle is shown in *Table 2.2* Mapping between two types of SLA lifecycle. The 'creation' phase of three phase lifecycle maps to the first three steps of the other lifecycle. In addition, the 'operation' phase of three phase lifecycle is the same as the fourth step of the other lifecycle.

Table 2.2 Mapping between two types of SLA lifecycle

Three phases	Six steps
1.	1.2 .3
2.	4.
3.	5.6.

The six steps SLA lifecycle is more reasonable and provides detailed fine grain information, because it includes important processes, such as re/negotiation and violation control. During the service negotiation or renegotiation, a consumer exchanges a number of contract messages with a provider in order to reach a mutual agreement. The result of these processes leads to a new SLA [66]. In six steps lifecycle, steps 2 and 3 map to these processes. However, the three phase's lifecycle does not include them. Furthermore, the 'Enforce Penalties for SLA violation' phase is important because it motivates parties adhere to follow the contract. We believe that the six steps formalization of the SLA life cycle provides a better characterization of the phenomenon and from here onwards we will refer to this as SLA life cycle.

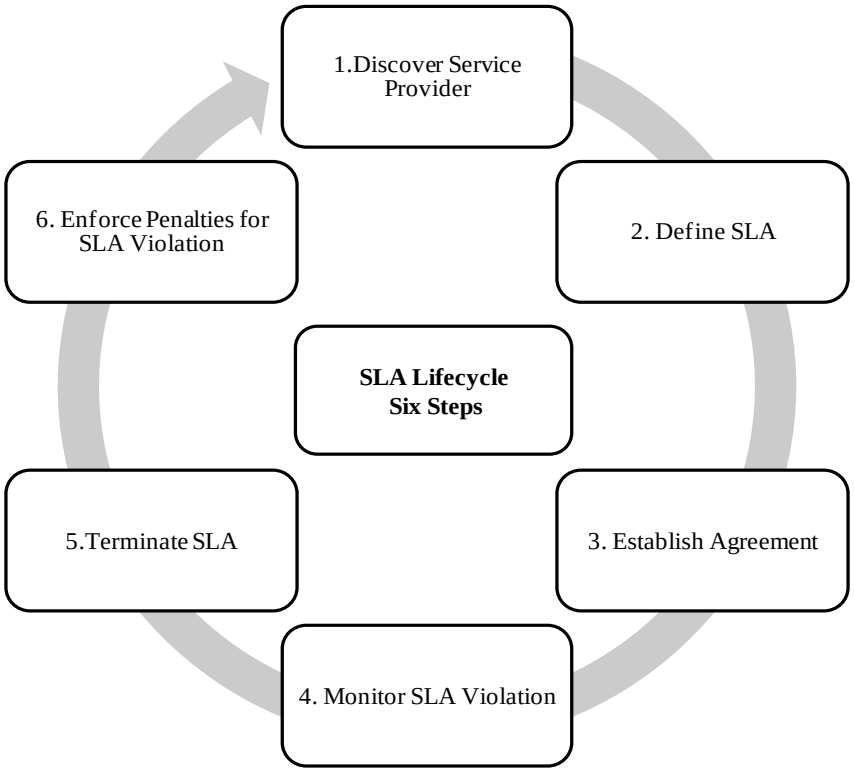


Figure 2.5 SLA life cycle six steps, as defined by Sun Microsystems Internet Data Center Group [54]

2.3 SLA in Utility Computing Systems

As highlighted by Patterson [5], there are many challenges involved in developing software for a million users to use as a service via a data center as compared to distributing software for a million users to run on their individual personal computers. Using SLAs to define service parameters that are required by users, the service provider knows how users value their service requests, hence it provides feedback mechanisms to encourage and discourage service request submissions. In particular, utility models are essential to balance the supply and the demand of computing resources by selectively accepting and fulfilling limited service requests out of many competing service requests submitted.

However, in the case of service providers making available a commercial offer to enable crucial business operations of companies, there are other critical QoS parameters to be considered in a service request, such as reliability and trust/security. In particular, QoS requirements cannot be static and need to be dynamically updated over time due to continuing changes in business operations and operating environments. In short, there should be greater importance on customers since they pay for accessing services. Therefore, the emphasis of this section is to describe SLA management in utility computing systems.

2.3.1 SLA Management in Utility Computing Systems

SLA management includes several challenges and in this section we will discuss them as part of the steps of the SLA life cycle.

Discover - Service Provider

In current utility computing environments, especially Grid and Cloud, it is important to locate resources that can satisfy consumers' requirement efficiently and optimally [32]. Such computing environments contain a large collection of different types of resources, which are distributed worldwide. These resources are owned and operated by various providers with heterogeneous administrative policies. Resources or services can join and leave a computing environment at any time. Therefore, their status changes dynamically and unpredictably. Solutions for service provider discovery problems must efficiently deal with scalability, dynamic changes, heterogeneity and autonomous administration.

Define - SLA

Once service providers have been discovered, it is necessary to identify the various elements of an SLA that will be signed by agreeing metrics. These elements are called service terms and include QoS parameters, the delivery ability of the provider, the performance target of diversity components of user's workloads, the bounds of guaranted availability and performance, the measurement and reporting mechanisms, the cost of the service, the data set for renegotiation, and the penalty terms for SLA violation. In this stage of the SLA lifecycle, measurement metrics and definition of each of these elements is done by a negotiation process between both parties [16][25].

Other challanges are related to the negotiation process. Firstly, parties may use different negotiation protocols or they may not have the common definition of the same service [19]. Secondly, service descriptions, in an SLA, must be defined unambiguously and be contextually specified by the means of its domain and actor. Therefore, an SLA language must allow the parameterisation of service description [43]. Moreover it should allow a high degree of flexibility and enable a precise formalisation of what a service guarantee means. Another aspect is how to keep SLA definition consistent throughout the entire SLA lifecycle.

Establish - Agreement

In this step an SLA template is constructed. A template has to include all aspects of SLA components. In utility computing environments, to facilitate dynamic, versatile, and adaptive IT infrastructures, utility computing systems have to promply react to environmental changes, software failures, and other events which may influence the system's behavior. Therefore, how to manage SLA-based adaptive systems, which exploit self-renegotiation after system failure, becomes an open issue [20]. Although most of the works recognise SLA negotiation as a key aspect of SLA managemet, recent works only provide little insight on how negotiation (especially automated negotiation) can be realised. In generalclients provide their QoS requirements; however, given the dynamic and hetergeneous nature of underline computing system, it is not trivial for the service providers to reflect or gurantee the quality aspects of SLA components in a template.

Monitor - SLA Violation

SLA violation monitoring begins once an agreement has been established. It plays a critical role in determining whether SLOs are achieved or violated. There are three main concerns.

Firstly, which party should be in charge of this process? There are two types of SLAs, negotiable and non-negotiable. When a non-negotiable SLA is offered, the provider administers those portions stipulated in the agreement. In the case of PaaS or IaaS, it is usually the responsibility of the consumer's system administrators to effectively manage the residual services specified in the SLA, with some offset expected by the provider to ensure basic quality of service [183]. In the case of SaaS, it is the customer who monitors the quality of service and SaaS provider will be responsible for the SLA violations, and this responsibility might be transferred to the PaaS or IaaS providers if SaaS using their services. Secondly, how fairness can be assured between parties. Thirdly, how the boundaries of SLA violation are defined.

SLA violation means '*un-fulfillment*' of service agreement. According to the Principles of European Contract Law, the term '*un-fulfillment'* is defined as defective performance (parameter monitored at lower level than agreed), late performance (service delivered at the appropriate level but with unjustified delays), and no performance (service not provided at all). There are three broad provisioning categories based on the above definition [48]. '*All-or-Nothing*' provisioning, characterizes the case in which all SLOs must be satisfied or delivered by the provider. '*Partial*' provisioning identifies some SLOs as mandatory ones, and must be met for the successful service delivery by both parties. '*Weighted Partial*' provisioning, is the case in which the "*provision of a service meets SLO if it has a weight greater than a threshold (defined by the client)*" [48]. '*All-or-Nothing*' provisioning is used in most cases of SLA violation monitoring, because violation leads to complete failure and negotiation to create a new SLA. An SLA contains mandatory SLOs that must be delivered by the provider. Hence, in '*Partial*' provisioning, all parties assign these SLOs the highest priority to reduce violation risk. How much the SLO affects the '*Business Value*' a measure of the importance of a particular SLO term? The more important the violated SLO, the more difficult it is to renegotiate the SLA, because any party does not want to lose their competitive advantages in the market.

Terminate - SLA

In terminating a SLA, a key aspect is to decide when it should be terminated, and once decided, all associated configuration information is removed from the service systems.

If the termination is due to a SLA violation, two questions need to be answered, who is the party that triggered this activity and what are the consequences of it.

Enforce Penalties for SLA Violation

In order to enforce penalties for SLA violation, penalty clauses are need to be defined. In utility computing systems, where consumers and provides are globally distributed, the penalty clauses work differently in various countries.

This leads to two problems, which particular clause should be used and whether it is fair for both sides. Moreover, due to the different types of violation, the penalty clauses need to be comprehensive. Recently, some works used the linear model for penalty enforcement of SLA violations in simple contexts [42][63]. The linear model exhibits a poor performance, thus, the selection of these best models for SLA violation penalty clauses enforcement is still an open problem.

2.3.2 Solutions for SLA Management in Utility Computing Systems

This section introduces solutions for the problems presented in the previous section. Six SLA management languages and frameworks are analyzed, because they can be used as solutions in multiple steps of SLA lifecycle.

SLA Management Frameworks and Languages

SLA can be represented by specialized languages for easing SLA preparation, automating SLA negotiation, adapting services automatically according to SLA terms, and reasoning about their composition. In this section we introduce six languages for SLA specification and management. Among them, the WS-Agreement and Web Service Level Agreement (WSLA) are the most popular and widely used in research and industry. The comparison among all of these languages is shown in *Table 2.3.*

Bilateral Protocol: Venugopal et al. [56] presented a negotiation mechanism for advanced resource reservation. It is a protocol for negotiating SLAs based on Rubinsteins Alternating Offers protocol for bargaining between parties. Any party is allowed to modify the proposal in order to reach a mutually-agreed contract. The authors implemented this protocol by using the Gridbus Broker on the customer's side and Aneka on the provider's side. Web services enable platform independence, and are therefore used to communicate between consumers and providers because the Gridbus Broker is implemented in Java, and Aneka is a .Net based

enterprise Grid. The advantage of these high level languages is that they are object oriented and web services enable semantic definition. Thus, this protocol supports SLA component reuse, and type and semantic definition.

WS-Agreement: Open Grid Forum (OGF) has defined a standard for the creation and the specification of SLAs called Web Services Agreement Specification (WS-Agreement) [12]. It is a language and a protocol for establishing, negotiating, and managing agreements on the usage of services at runtime between providers and consumers. It uses an XML-based language for specifying the nature of an agreement template, which facilitates discovery of compatible providers. Its interaction is based on request and response. Moreover, it helps parties in exposing their status, so SLA violation can be dynamically managed and verified. Originally the language did not support negotiation and currently it has been complemented. WS-Agreement Negotiation, which lies on the top of WS-Agreement and describes the re/negotiation of the SLA. Its main feature is the robust signaling protocol for the negotiation.

Web Service Level Agreement (WSLA): WSLA [40] is a framework developed by IBM to specify and monitor SLA for Web Services. It provides a formal XML schema based language to express SLAs, and architecture to interpret this language at runtime. It can measure, and monitor QoS parameters and report violations to the party. It separates monitoring clauses from contractual terms for outsourcing purposes. It provides the capability to create new metrics over existing metrics to implement multiple QoS parameters [40]. However, the semantic of metrics is not formally defined, hence, there are limitations for the creation of new terms base on existing terms.

WSOL: Web Service Offerings Language (WSOL) defines a syntax for service offers' interaction [53]. It provides template instantiation and reuse of definitions. WSOL and WSLA support definition of management information and actions, such as violation notifications. However, they are not defined by a formal semantic. WSOL and QML (Quality Management Language) support type systems allowing the same SLA to be described either in abstract or specific values to create a new SLA. The generalization relationships between SLAs facilitate definitions of SLA types.

SLAng: Skene et al. [55] propose Service Level Agreement Language (SLAng), which uses Extensible Markup Language (XML) to define SLAs. It is motivated by the fact that federated distributed systems must manage the quality of all aspects of their deployment. SLAng is different from other languages and frameworks. Firstly, it defines an SLA vocabulary for Internet services. Secondly, its structure is based on the specific industry requirement, aiming to provide usable terms. Thirdly, it is modeled using Unified Markup Language (UML) and defined according to the behavior of services and consumers involved in service usage, unlike other languages, such as WSLA and WSOL, where QoS definition is based on metrics. Moreover, it supports third party monitoring schemes. However, it lacks of the ability to define management information, such as associated financial terms. Thus, it is not suitable for commercial computing environments.

QML: QML [31] define a type system for SLAs, allowing users to define their own dimension types. However, it does not support extension of individual defined metrics because the exchange of SLAs between parties requires a common understanding of metrics. QML defines semantic for both its type system and its notion of SLA conformance.

QUO: It is a CORBA specific framework for QoS adaption based on proxies [43]. It includes a quality description language used for describing QoS parameters, adaptations and notifications. QUO properties are the response of invoking instrumentation methods on remote objects. Like WSLA, no formal constraints are placed on the implementation of these methods.

Discover - Service Provider

In the Grid computing community, Fitzgerald [28] introduced the Monitoring and Discovery System, Gong et al. [32] proposed the VEGA Grid Project and also relevant is the work of Iamnitchi et al. [35].

Monitoring and Discovery System (MDS) is the information service described in the Globus project [28]. In its architecture, Lightweight Directory Access Protocol (LDAP) is used as directory service, and information stored in information servers are organized in tree topology. In utility computing systems, resources' availability and capability are dynamic in nature. However, in MDS, the relationship between information and information servers is

static. In addition, service provider's information is frequently updated in these dynamic changing environments, whilst LDAP is not designed for writing and updating information.

VEGA Infrastructure for Resource Discovery (VIRD) has three-level hierarchy architecture. The top level is a backbone, which is responsible for the inter-domain resource discovery and consists of Border Grid Resource Name Servers (BGRNS). The second level consists of several domains and each domain consists of Grid Resource Name Servers (GRNS). The third level includes all clients and resource providers. There is no central control in this architecture, thus resource providers register themselves to GRNS server within a domain. When clients submit requests, GRNS responses to them with requested resources. The limitation of this architecture is that it only focuses on the issue of scalability and dynamic environmental changes but not on heterogeneity and autonomous administration.

Iamnitchi et al. [35] propose a resource discovery framework using peer-to-peer (P2P) technologies in Grids. P2P architecture is fully distributed and all the nodes are equivalent. However, one major limitation of their work is that every node has little knowledge about resources distribution and their status. Specifically, when there is large number of resource types or the work-set is very large, the opportunity for inaccurate results increases, because the framework is not able to use history data to accurately discover resources.

Define - SLA and Establish - Agreement

'Define – SLA' and 'Establish – Agreement' are two dependent steps, and SLA languages facilitate their development. For example, WSLA and WS-Agreement are the most widely used languages in these steps. Creation and Monitoring of Agreements (CREMONA) is a WS-Agreement framework implemented by IBM [26]. It proposes a Commitment Agreement and architecture for the WS-Agreement. All of these agreements are normal WS-Agreements, following a certain naming convention. This protocol basically aims at solving problems related to the creation of agreements on multiple sites. However, it is unable to solve limitations when service providers and consumers have different standards, policies, and languages during negotiations. For example, if a consumer uses WSLA but a provider uses WS-Agreement, the interaction is actually not possible. In order to solve this, Brandic et al. [19] proposed a Meta-Negotiation Architecture for SLA-Aware Grid Services based on meta-negotiation documents. These documents record supported protocols, document

languages, and the prerequisites for starting negotiations and establishing agreements for all participants.

SLA-based Resource Management Systems (RMS) have been developed for addressing negotiation problems in Grids, for example, Wurman et al. [61] state a set of auction parameters and a price-based negotiation platform, which serves as an auction server for humans and software agents. Nevertheless, their solution only support one-dimensional auction (only focus on price), but not multiple-dimensional auctions, which are important in utility computing environments.

Table 2.3 Comparison of SLA Management frameworks and Languages

Name	Type	Domain	Dynamic Establish / Management	Negotiation	Metrics	Define Management Actions	Support Reuse	Provide Type Systems	Define Semantic	Cope with SLA lifecycle
Bilateral Protocol	Java, .Net and Web Service based protocol	Originally for resource reservation in Grids.	Yes	Yes	Yes	Yes	Yes.	Yes	Support by Web Service.	Step 1 to Step 4.
WS-Agreement	XML language; Framework; A protocol	Any domain	Establish and manage dynamically	Re/negotiation with WS-Agreement Negotiation	Do not define specification of metrics associated with agreement parameters.	Yes	Yes	Yes	Not formally defined	Step 1 to step 6
WSLA	Provide language; Framework; runtime architecture	Originally for Web services	Establish and manage dynamically	Re/negotiation.	Allows creation of new metrics	Yes	Yes	NA	Not formally defined	Step 1 to step 6
QML	language	Any Domain	Yes	Yes	Allows creation of new metrics	Yes	Yes	Yes, allows definition of new	Yes	Step 1 to step 4

								type systems		
WSOL	XML	Originally for Web Services	Yes	Originally do not support	NA	Yes	Yes	Yes	No	Step 1 to step 4
QUO	CORBA specific framework	Any domain	Yes	Yes	NA	Yes	Yes	Yes	No	Step 1 to step 4
SLAng	XML Language	Originally for Internet DS environment	NA	Yes	No But based on behavior of SLA parties	NA	Yes	Yes	Yes	Step 1 to Step 4

Monitor - SLA Violation

Monitoring infrastructures are used to measure the difference between the pre-agreed and actual service provision between parties [48]. There are three types of monitoring infrastructures, which are trusted third party (TTP), trusted module on the provide side, and trusted module on the client side. Nowadays, TTP provides most of functionalities for monitoring in most typical situations to detect SLA violation.

Terminate - SLA

There are two scenarios in which an SLA may be terminated. The first is termination due to normal time out. The second one is termination because any party violated its contract terms. Normally, in Clouds, this step is conducted by customers and termination typically is caused by normal time out or the provider's SLA violation. Sometimes, providers also terminate SLAs depending on the task priorities. If the reason for SLA termination is violation, then the 'Enforce Penalties for SLA Violation' step of the SLA lifecycle has to be applied. This step is normally performed manually.

Enforce Penalties for SLA Violation

A penalty clause can be applied to the party who violates SLA terms. First is a direct financial deposit being negotiated and agreed between parties. Second is a decrease in price along with the extra compensation for any subsequent interaction. In other words, this option is according to the value of loss caused by the violation. In this case, TTP is usually used as a mediator. The workflow for this option is that clients transfer their deposit, bond, and any other fees into the Third Party's account, and then if the SLOs have been met, the money is paid to provider via TTP. Otherwise, the TTP returns the amount of fees back to the consumer as compensation for SLA violations. The SLA violation has two indirect side impacts on providers. The first is that consumers use less service from the provider in the future. The second is that provider' reputation decreases and it affects other clients' willing to choose this provider subsequently. The major indirect influence on consumer is future request will be rejected due to bad credit record.

A major issue, in the above discussion, is the variety of laws enforced in different countries. This problem can be solved by a '*choice of law clause*', which indicates expressly which country' laws are applied when a conflict happens between parties. '*Legal templates*' [27] can be used to refine these clauses [48].

2.4 SLA Use Cases in Utility Computing Systems

Utility computing provides access to on-demand delivery of IT capabilities to the consumer according to cost-effective pricing schema. Typically, a resource in a Data Center is idle during 85% of time [63]. Utility computing provides a way for enterprises to lease this 85% of idle resource or to use outsourcing to pay for resources according to their usage. Two approaches of utility computing that achieve above goals are Grid and Cloud. In the rest part of this section, we present use cases in Grid and Cloud computing environments.

2.4.1 SLA in Grid Computing Systems

In this section we introduce the definition of Grid computing, and some recent significant Grid computing projects that have focused on SLAs and enabled them in their frameworks.

According to Buyya et al. (2009) "*A Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed 'autonomous' resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements [22]*." Grid computing is a paradigm of utility computing, typically used for access to NPC and scientific resources, even though it has been also used in the industry.

SLA has been adopted in Grid computing, and many Grid projects are SLA oriented. We classify them into three categories, which are SLA for business collaboration, SLA for risk assessment, and SLA renegotiation supports dynamic changes.

SLA for Business Collaboration: GRIA (The GRIA Project) is a service-oriented infrastructure designed to support B2B collaborations across organizational boundaries by providing services. The framework includes a service manager with the ability to identify the available resources (e.g. CPUs and applications), assign portions of the resources to consumers by SLAs, and charge for resource usage. Furthermore, a monitoring service is responsible for monitoring the activity of services with respect to agreed SLOs.

The BREIN consortium (The BREIN Project, 2006-2009) defines a business framework prototype for electronic business collaborations. Some capabilities of this framework prototype include Service Discovery with respect to SLA capabilities, SLA negotiation in a single-round

phase, system monitoring and evaluation, and SLA evaluation with respect to the agreed SLA. The WSLA/WS-Agreement specifications are suggested for SLAs management. The project focuses on dynamic SLAs. This initiative shows that the industry is demonstrating their interest in SLA management.

In the work of Joita et al. [37], WS-Agreement specification is used as a basis to conduct negotiation between two parties. An agent-based infrastructure takes care of the agreement offer made by the requesting party. In this scenario, many one-to-one negotiations are considered in order to find the service that best matches the offer.

Risk Assessment: The AssessGrid [15] project focuses on risk management and assessment in Grid. It aims at providing service providers with risk assessment tools, which help them to make decisions on the suitable SLA offer by assigning, mapping, and associating the risk of failure to penalty fees. Similarly, end-users get knowledge about the risk of an SLA violation by a resource provider that helps them to make appropriate decisions regarding acceptable costs and penalty fees. A broker is the matchmaker between end-users and providers. WS-Agreement-Negotiation protocol is responsible for negotiating SLAs with external contractors.

SLA renegotiation supporting dynamic changes: Ludwig et al. [44] propose an extension of WS-Agreement allowing a run-time SLA renegotiation. Some modifications are proposed in the 'GuaranteeTerm' section of the agreement schema and a new section is added to define possible negotiations, to be agreed by parties before the offer is submitted. The limitation is that it does not support run-time renegotiation to adapt dynamic operational and environmental changes, because after the agreement's acceptance, there is no interaction between the provider and the consumer. Sakellariou et al. [53] specify the guarantee terms of an agreement as variable values rather than fixed values. This work aims at minimizing the number of re-negotiations to reach consensus with agreement terms. BabelNet, is a Protocol Description Language for automated SLA negotiation, has been proposed [34] to handle multiple-dimensional auctions.

2.4.2 SLA in Cloud Computing

Cloud computing is a paradigm of service oriented utility computing. In this section we introduce a definition of Cloud computing and SLA use cases in industry and academia. Finally, we compare SLA usage difference between Cloud computing and traditional web services.

Cloud Computing

Based on the observation of the essence of what Clouds are promising to be, Buyya et al. (2009) propose the following definition: "*A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumer[22]*." Hence, Clouds fit well into the definition of utility computing.

Figure 2.6 shows the layered design of Cloud computing architecture. Physical Cloud resources along with core middleware capabilities from the bottom for delivering IaaS. The user-level middleware aims at providing PaaS capabilities. The top layer focuses on application services (SaaS) by making use of services provided by the lower layer services. PaaS/SaaS services are often provided by 3rd party service providers, who are different from IaaS providers [23].

User-Level Applications: this layer includes the software applications, such as social computing applications and enterprise applications, which be deployed by PaaS providers renting resources from IaaS providers.

Core Middleware: this layer provides runtime environment enabling Capabilities to application services built using User-Level Middleware. Dynamic SLA management, Accounting, Monitoring and Billing are examples of core services in this layer. The commercial example suit this layer are Google App Engine and Aneka.

System Level: physical resources including physical machines and virtual machines sit in this layer. These resources are transparently managed by higher level virtualization services and toolkits that allow sharing of their capacity among virtual instances of servers.

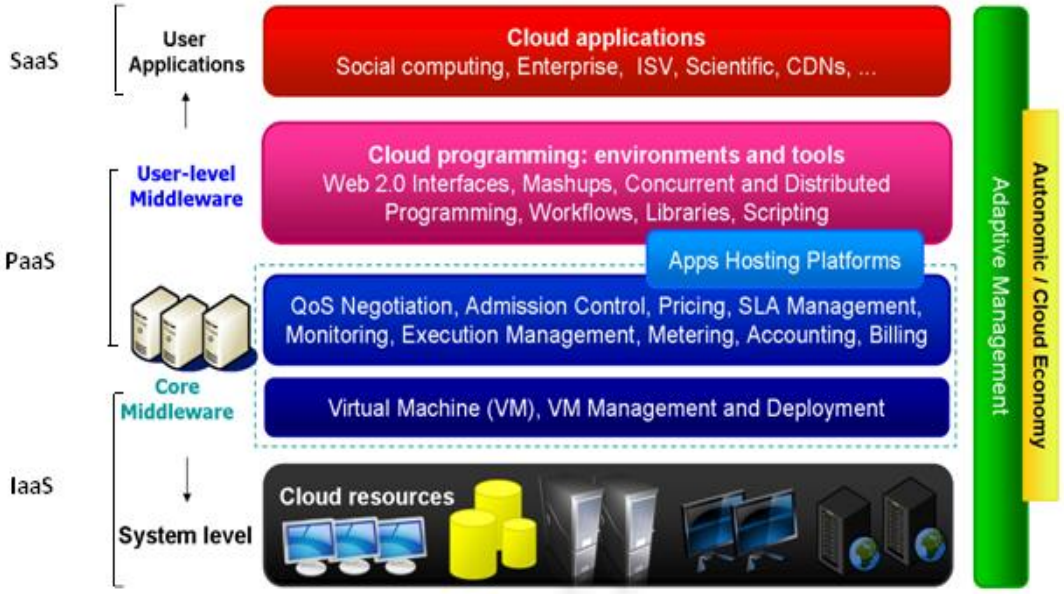


Figure 2.6 Layered Cloud computing architecture [23]

Use Cases

In this section, we present industry and academic use cases in Cloud computing environments.

Industry Use Cases: In this section, we present how Cloud providers implement SLA. Important parameters are summarized in *Table 2.4*. All elements in *Table 2.4*, are original from formal published SLA documents of AmazonEC2 and S3 (IaaS provider), and Microsoft Azure[1] Compute and Storage (IaaS/PaaS provider).

A Characterization of studied systems following the six steps of SLA lifecycle model is summarized in *Table 2.5*. From the users' perspective, the process of activating SLA lifecycle with Amazon and Microsoft is simple because the SLA has been pre-defined by the provider. According to SLA lifecycle, the first step is to find the service providers according to users' requirements. For example, users find the provider via searching on the Internet, and then explore the providers' web site for collecting further information. Most Cloud service providers offer pre-defined SLA documents. In this case, the second step and third step are pre-defined and always be entwined together. The check for SLA violation monitoring can be done by third party tools, such as Cloudwatch, Cloudstatus, Monists,

Nimsoft. Developers are able to develop their own monitoring systems by taking use of these tools.

For what concerns the termination of a SLA we can consider IaaS services as a reference example. In this case three scenarios may occur. The normal termination of a SLA is constituted by the release of Cloud release of Cloud resources by the user. An SLA can also be actively terminated by a provider if the resource usage lasts beyond the predefined expire time. A termination with penalty may occur in case the resource is unable to provide resources according to the expected Quality of Service. The last step of SLA lifecycle will be invoked if any party violates contract terms. Currently most of service providers give service credit to customer if they violate SLA.

Table 2.4 SLA Use Cases of the most famous Cloud Provider and related characteristics in SLAs

Cloud Provider Name	**Service Commitment**	**Effective Date**	**Monthly Uptime Percentage (MUP)%**	**Service Credits Percentage (%)**
Amazon AWS EC2	"AWS will use commercially reasonable efforts to make Amazon EC2 and Amazon EBS each available with a Monthly Uptime Percentage (defined below) of at least 99.95%, in each case during any monthly billing cycle (the "Service Commitment"). In the event Amazon EC2 or Amazon EBS does not meet the Service Commitment, you will be eligible to receive a Service Credit "(AWS EC2 Service Level Agreement).	01 June, 2013	99%=<MUP<99.9%	**10%**
			MUP%<99%	**30%**
Amazon	"AWS will use	01 June,	99%=<MUP<99.9%	**10%**

AWS S3	commercially reasonable efforts to make Amazon S3 available with a Monthly Uptime Percentage (defined below) of at least 99.9% during any monthly billing cycle (the "Service Commitment"). In the event Amazon S3 does not meet the Service Commitment, you will be eligible to receive a Service Credit as described below. "(AWS S3 Service Level Agreement).	2013	MUP<99	**25%**
Microsoft Azure	"For Cloud Services, we guarantee that when you deploy two or more role instances in different fault and upgrade domains, your Internet facing roles will have external connectivity at least 99.95% of the time. For all Internet facing Virtual Machines that have two or more instances deployed in the same Availability Set, we guarantee you will have external connectivity at least 99.95% of the time. For Virtual Network, we guarantee a 99.9% Virtual Network Gateway availability." (Windows Azure Service Level Agreement)	NA	<99.95%	**10%**
			<99%	**25%**

1.The formula used to calculate Monthly Connectivity Uptime Percentage (MCUP) is depending on Maximum Connectivity Minutest (MCM), Connectivity Downtime (CD) and Maximum Connectivity Minutest (MCM). The equation is given as follows $MCUP = (MCM - CD) \div MCM$ *Source*: *Windows Azure Service Level Agreement*

Table 2.5 From users' perspective SLA Use Cases of Cloud Provider follows six steps SLA lifecycle

Cloud Service Provider	**Service Type**	**Step 1: Discover-Service Provider**	**Step 2: Define-SLA**	**Step 3: Establish-Agreement**	**Step 4: Monitor-SLA Violation**	**Step 5: Terminate-SLA**	**Step 6: Enforce Penalties for SLA Violation**
Amazon EC2	IaaS (Computing)	Discover manually (e.g. via web site)	Pre-defined SLA terms and QoS parameters	Pre-defined SLA document by provider	Can use third party monitor systems (e.g. CloudWatch)	By user, or provider programmatically or manually	Service Credit given by provider
Amazon S3	IaaS (Storage)	Discover manually	Pre-defined SLA terms and QoS parameters	Pre-defined SLA document by provider	Can use third party monitor systems (e.g. CloudStatus)	By user, or provider programmatically or manually	Service Credit given by provider
Microsoft Azure Compute	PaaS	Discover manually (e.g. via web site)	Pre-defined SLA terms and QoS parameters	Pre-defined SLA document by provider	Can use third party monitor systems (e.g. Monitis)	By user, or provider programmatically or manually	Service Credit given by provider
Microsoft Azure Storage	PaaS	Discover manually	Pre-defined SLA terms and QoS parameters	Pre-defined SLA document by provider	Can use third party monitor systems (e.g. Monitis)	By user, or provider programmatically or manually	Service Credit given by provider

Academy Use Cases: In this section, we present SLA-based projects and algorithms as academy use cases.

SLA-based Resource Allocation for Data Centers and Cloud Computing Systems: The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, at the University of Melbourne has proposed the use of market-based resource management to support utility-based resource management for cluster computing [65][64]. The initial work successfully demonstrated that market-based resource allocation strategies are able to deliver better utility for users than traditional system-centric strategies. However, early research focused on satisfying only two static Quality of Service (QoS) parameters: the deadline for completing a service request and the budget that the consumer is willing to pay for completing the request before the deadline. In the commercial computing environment, there are other critical QoS parameters to consider in a service request, such as reliability and trust/security. In particular, QoS requirements cannot be static and need to be dynamically updated over time due to continuing changes in business operations and operating environments.

SLA based Management and Scheduling: Lee et al. [42] propose profit-driven SLA based scheduling algorithms in Clouds to maximize the profit for service providers. The application model used in this work can be classified as SaaS and PaaS. The service types supported by their algorithm are dependent services, which mean one sub-service can not start until the pre-required services complete. However, their work does not support multiple providers and full simulation configuration is not available. We recommend possible future research direction is SLA management with multiple providers, since it is required for emerging research in InterCloud. We define InterCloud as multiple Cloud providers with peer agreement to support collaborative activities.

Several projects in the last years are related at different degrees to the SLA-aware management of resources, such as Claudia[176], BonFIRE [179], Optimis [177] and 4CaaSt [178].

Claudia: is a toolkit aims to provide dynamic provision and scalability of services in IaaS Clouds. BonFIRE is a European project provides a unified federation environment for developers to manage Cloud deployments. In addition, European project 4CaaSt targets to provide a platform for the deployment, management and trade of Cloud services. It allows

providers to federate their resources in a common marketplace and enables users to compose services. However these works neither consider dynamic management of resources nor consider QoS parameters, so SLA-based resource management is not in their scope.

Optimis: A European project aimed to enable private Cloud to automatically interact with public Cloud providers, optimizing the usage of resources by means of Cloud federation; it does scheduling operations by deciding the best provider to host resources. It allows specifying requirements at IaaS level and constraints in Cloud services. However, this work does not cover SaaS level requirements and only considers cost but not customer satisfaction level.

SLA@SOI: The SLA@SOI project has developed a methodology for the SLA-aware management of infrastructures and services, and encompasses activities such as dynamic service discovery and composition, service monitoring and assessment, infrastructure planning and optimization etc. However this project does not consider Cloud computing infrastructures as their target platform, and hence it does not account for some specific needs of this area.

Cloud-TM [180]: a European project aimed to provide a data centric PaaS middleware for the development of distributed Cloud applications. However, this work does not cover SaaS level. The SLA system is based on SLA@SOI. However this project does not cover the PaaS and SaaS levels of Cloud computing, and is focused on data centric Cloud applications, instead of the general purpose Cloud computing.

PaaSage [181] : another recent European project providing runtime monitoring and dynamic adaptation, intelligent metadata retrieval, multi provider support, etc. Although this project covers several topics dealing with QoS assessment and dynamic management of resources, it does not use SLAs for the definition of resources or QoS requirements, nor cover SaaS Level of Cloud computing.

SLA related difference between Cloud and Web Service

In this section we compare the difference between SLAs applied in cloud computing and in traditional web services as follows:

QoS Parameters: Most web services focus on parameters such as response time, SLA violation rate for the task, reliability, availability, levels of user differentiation, and cost of service. In Cloud computing more QoS parameters than traditional web services need to be considered, for example, energy related QoS, Security related QoS, Privacy related QoS, trust related QoS. More than 20 QoS parameters are defined by the SMI (Service Management Index) consortium to be used in the industry and academy as standard benchmark.

Automation: The whole process of SLA negotiation and provisioning, service delivery and monitoring need to be automated for highly dynamic and scalable service consumption. Researches in traditional web services explored this topic, for example, Jin L.J et al [36] proposed a model for SLA analysis of Web Services. Nevertheless, SLA automation is a rapidly growing area in Cloud computing. In fact there are some research projects starting to focus on it, such as CLOUDS Lab at the University of Melbourne and SLA@SOI.

Resource Allocation: SLA oriented resource allocation in Cloud computing is possible different from allocation in traditional web services, because web services have a Universal Description Discovery and Integration (UDDI) for advertising and discovering between web services. However, in Clouds, resources are allocated and distributed globally without central directory, so the strategy and architecture for SLA based resource allocation in such environment are different from traditional web services.

2.5 Open Problems

SLA management must provide ways for reliable provisioning of services, monitoring of SLA violations and detection of any potential performance decrease during service execution [41][45]. The goal of SLA management is to establish a scalable and automatic SLA management framework for automatically adapting to dynamic environmental changes by considering multiple QoS parameters. In addition, an SLA has to be suitable for multiple domains with heterogeneous resources. The VIRD architecture is a three-level hierarchy focused on scalability. Wurman et al. [61] state a set of auction parameters and price-based negotiation platform. Nevertheless, this solution only supports one-dimensional auction, thus could not handle

multiple-dimensional auctions, which are important in utility computing environments. Recently, BabelNet handles multiple-dimensional auctions.

Nevertheless, somehow consumers still need to be involved in the management process to some extent. Moreover, multiple QoS parameters have been investigated by CLOUDS Lab's initial work. Whilst that work only focused on the most common QoS parameters (price and deadline), there are other critical QoS parameters that should be considered in a service request, such as reliability and trust/security. In particular, QoS parameters are must be updated dynamically over time due to continuing changes in business operations environments. Thus, multiple QoS parameters should be investigated in the future research work.

More specifically, there are some open challenges for SLA-based resource management. First and foremost, different SLA negotiation protocols and processes constraint the negotiation for establishing SLAs, the modification of an implemented SLA, and SLA negotiation between distinct administrative domains. Second, the SLA has to be established between providers and consumers from different end-to-end viewpoint. For example, if the system service has been outsourced from one provider to another, there should be SLA agreement between them as well. Similar to Business to Consumer (B2C) models and Business to Business (B2B) models, there will be different types of SLAs that needs to be established depending on entities involved.. Third, admission control policies, because decision on which user request to accept affects the performance, profit, and reputation of the resource provider. Moreover, the resource allocation management has to be considered carefully, because it addresses which resource is best suitable for current admitted requests from both parties' point of view. In addition, management of QoS metrics, different parties using different parameters, and the failure management become a challenge especially for the automatic handling, such as cause analysis and automatic problem resolution. We can also mention, performance forecast management is another open question in utility computing environments because it enables the recommendation for performance improvement.

2.6 Summary

This chapter presented the literature survey, issues and solutions of SLA management in utility computing systems and how SLAs have been used in these systems. An SLA is a formal contract between service providers and consumers to guarantee that the service quality is delivered to

satisfy pre-agreed consumers' expectations. SLA management is important in utility computing systems because it helps to improve the CSL and to define clear relationship between business parties. In this chapter, we summarized the main fundamental concepts of SLA and analyzed two types of SLA lifecycle. One is the three phase high level lifecycle, which includes creation phase, operation phase and removal phase; the other is more specific lifecycle including six steps, which are 'discover-service provider', 'define-SLA elements', 'establish-agreement', 'monitor-SLA violation', 'terminate-SLA' and 'SLA violation control'. The second type of lifecycle is more comprehensive, and introduces the characterization of SLA violation that is a foundation in utility computing environments where services are consumed on a pay-as-you-go basis.

The analysis carried out in this chapter identified four major goals in case of SLA-based utility computing. First, supporting customer-driven service management based on customer profiles and requested service requirements. Second, defining computational risk management tactics to identify and manage risks involved in the execution of applications with regards to service requirements and customer needs. Third, deriving appropriate market-based resource management strategies encompassing customer-driven service management to sustain SLA-based resource allocation. Fourth, how to incorporate adaptive resource management models and dynamic changes in service requirements in order to satisfy both new service demands and existing service obligations.

To achieve these goals, the main challenges and solutions of SLA-based resource management in utility computing environments are discussed by following the steps of SLA lifecycle. In the 'discover-service provider', the main issues are scalability, dynamic changes, heterogeneity, and autonomous administration. Some architectures and algorithms have been proposed to cope with them, such as the MDS and VIRD architectures. To design an automatic negotiation framework is a challenge during the 'define-SLA' and 'establish- agreement' steps, because two parties need to negotiate before they agree on the terms to be included in SLAs. SLA frameworks and languages are used as solutions. Currently, the most widely used languages are WSLA and WS-Agreement. However, there are not many effective solutions for the automatic negotiation framework for SLA-based resource management. Thus, the automatic negotiation is still an open issue. Regarding the 'monitor SLA violation' step, the most popular solution is using Third Party (TTP) who provides most of functionalities for monitoring a service in most typical situations to detect SLA violations. The main issues for the last two steps 'terminate SLA' and 'enforce penalties for SLA violation', are automatic failure management, such as cause analysis, penalty

clauses invocation, and automatic failure resolution. Some penalty strategies were presented. However, resource management with penalty model and automatic problem resolution still are open challenges and more investigation is needed in the future.

In conclusion, SLA in utility computing systems is a rapidly moving target although some works have been explored in the past. The rest of this thesis will explore three major challenges listed in the Chapter 1. In addition, the next chapter will investigate admission control and scheduling algorithms for SaaS providers to effectively utilise public Cloud resources to maximize profit by minimizing cost and improving customer satisfaction level.

3 SLA-based Admission Control for Software-as-a-Service Providers

This chapter presents innovative admission control and scheduling algorithms for SaaS providers to effectively utilise heterogeneous Cloud resources to maximize profit by minimizing cost and enlarging market share by accepting more user requests while minimizing the SLA violations for existing customers. Then, an extensive evaluation study is conducted to analyse which algorithm suits best in which scenario to achieve SaaS (Software-as-a-Service) providers' objectives. Simulation results show that our proposed algorithms provide substantial improvement (up to 40% cost saving) over reference ones across all ranges of variation in QoS parameters.

3.1 Introduction

The general objective of SaaS providers is to maximize profit and enlarge market share. To maximize profit, SaaS (Software-as-a-Service) providers need to minimize the infrastructure cost, administration operation cost and penalty cost caused by SLA violations. Market share can be enlarged by accepting more user requests, which also increases the profit. Market share can also be improved by satisfying more customers. To satisfy the customer, SaaS providers need to guarantee Quality of Service (QoS) specified in SLAs.

In general, SaaS providers utilize internal resources of its data centres or rent resources from a specific IaaS provider. For example, Saleforce.com [102] hosts resources but Animoto rents resources from Amazon EC2 [92]. In-house hosting can generate administration and maintenance cost while renting resources from a single IaaS provider can impact the service quality offered to SaaS customers due to the variable performance [103].

To overcome the above limitations, multiple IaaS providers and admission control are considered in this chapter. Procuring from multiple IaaS providers brings huge amount of resources, various price schemas, and flexible resource performance to satisfy Service Level Objectives, which are items specified in Service Level Agreement (SLA). Admission control has been used as a general mechanism to avoid overloading of resources and SLA satisfaction [2]. However, current SaaS providers do not have admission control and how they conduct scheduling is not publicly known. Therefore, the following questions need to be answered to allow efficient use of resources in the context of SaaS providers using multiple resources from IaaS providers, where resources can be dynamically expanded and contracted on demand:

- Can a new user request be accepted without impacting accepted requests?
- How to map various user requests with different QoS parameters to VMs?
- What available resource should the request be assigned to? Or should a new VM be initiated to support the new user request?

This chapter provides answers to the above questions by proposing an innovative cost-effective admission control and scheduling algorithms to maximize the SaaS provider's profit and CSL. Our proposed solutions are able to maximize the number of accepted users through the efficient placement of requests on VMs leased from multiple IaaS providers. We take into account various customer's QoS requirements and infrastructure heterogeneity. The key contributions of this chapter are twofold: 1) it proposes the system and mathematical models for SaaS providers to satisfy customers; and 2) it proposes three innovative admission control and scheduling algorithms for profit and market share maximization by accepting as many new user requests as possible with guaranteed SLA and minimized cost.

3.2 System Model

In this section, we introduce a model, which consists of actors and 'admission control and scheduling' system (as depicted in *Figure 3.1*). The actors are users/customers, SaaS providers, and IaaS providers. The system consists of application layer and platform layer functions. Take Animoto.com as an example of SaaS provider, who leases video generation software to users. There are three steps for users to generate video using Animoto.com: 1) upload pictures or videos; 2) select themes, music and styles for the video; 3) download or share the video. In this example, customers expect video to be generated within deadline and budget. We extended this application model by focusing more on customer requirements satisfaction. Thus, users request the software

service from a SaaS provider by submitting their QoS requirements, such as service deadline and budget. The QoS model considered is adapted from utility models proposed in previous work [6]. In general, budget is computed by clients through own their market research and strategic plans. The platform layer uses **admission control** to interpret and analyse the user's QoS parameters and decides whether to accept or reject the request based on the capability, availability and price of VMs. Then, the **scheduling** component is responsible for allocating resources based on admission control decision. Furthermore, in this section we design two SLA layers with both users and resource providers, which are SLA (U) and SLA (R) respectively.

3.2.1 Actors

The participating actors involved in the process are discussed below along with their objectives and constraints:

User

On users' side, a request for application is sent to a SaaS provider's application layer with QoS constraints, such as, deadline, budget and penalty rate. Then, the platform layer utilizes the 'admission control and scheduling' algorithms to admit or reject this request. If the request can be accepted, a formal agreement (SLA) is signed between both parties to guarantee the QoS requirements. SLA with Users – SLA (U) includes the following properties:

- **Deadline:** Maximum time user would like to wait for the result.
- **Budget:** How much user is willing to pay for the requested services.
- **Penalty Rate Ratio:** A ratio for consumers' compensation if the SaaS provider misses the deadline.
- **Input File Size:** The size of input file provided by users. Users upload the file, and the size is calculated by the application layer function.
- **Request Length:** How many Millions of Instructions (MI) are required to be executed to serve the request? This value is predefined in the SLA (U) by the SaaS provider.

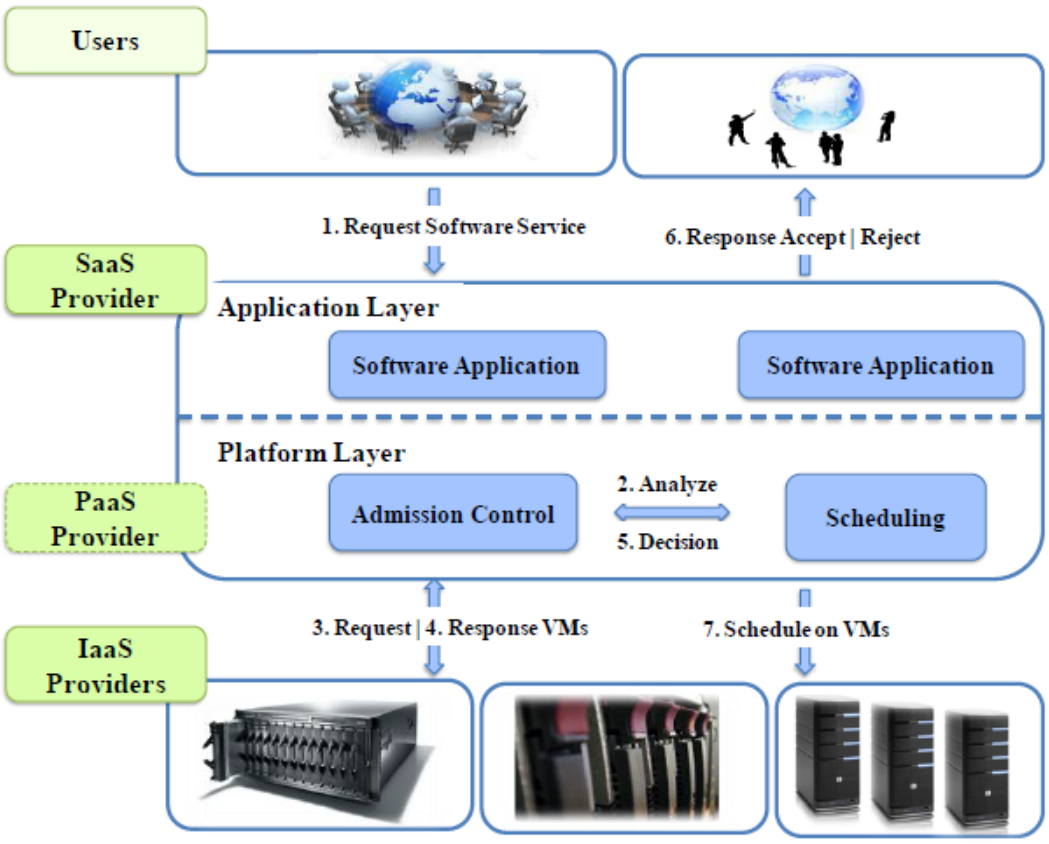


Figure 3.1 A high level system model for application service scalability for in IaaS providers.

SaaS provider

A SaaS provider rents resources from IaaS providers and leases software as services to users. SaaS providers aim at minimizing their operational cost by efficiently using resources from IaaS providers, and improving CSL by satisfying SLAs, which are used to guarantee QoS requirements of accepted users. From SaaS provider's point of view, there are two layers of SLA with both users and resource providers, which are described in Section A and Section C. It is important to establish two SLA layers, because SLA with user can help the SaaS provider to improve the CSL by gaining users' trust of the QoS; SLA with resource providers can enforce resource providers to deliver the satisfied service. If any participants in the contract violate its terms, the defaulter has to pay for the penalty according to the clauses defined in the SLA.

IaaS Provider

An IaaS resource provider (*RP*), offers VMs to SaaS providers and is responsible for dispatching VM images to run on their physical resources. The platform layer of SaaS

provider uses VM images to create instances. It is important to establish SLA with a resource provider – SLA (R), because it enforces the resource provider to guarantee service quality. Furthermore, it provides a risk transfer for SaaS providers, when the terms are violated by resource provider. In this work, we do not consider the compensation given by the resource provider because 85% resource providers do not really provide penalty enforcement for SLA violation currently [93]. The SLA (R) includes the following properties:

- **Service Initiation Time**: How long it takes to deploy a VM.
- **Price**: How much a SaaS provider has to pay per hour for using a VM from a resource provider?
- **Input Data Transfer Price:** How much a SaaS provider has to pay for data transfer from local machine (their own machine) to resource provider's VM.
- **Output Data Transfer Price**: How much a SaaS provider has to pay for data transfer from resource provider's VM to local machine?
- **Processing Speed**: How fast the VM can process? We use Machine Instruction Per Second (MIPS) of a VM as processing speed.
- **Data Transfer Speed**: How fast the data is transferred? It depends on the location distance and also the network performance.

3.2.2 Profit Model

In this section we describe mathematical Equations used in our work. Let assume at a given time instant *t*, *I* be the number of initiated VMs, and *J* be the total number of IaaS providers. Let IaaS provider *j* provides N_j types of VM, where each VM type *l* has P_{jl} price. The prices/GB charged for data transfer-in and –out by the IaaS provider *j* are $inPri_j$ *and* $outPri_j$ *respectively.* Let ($iniT_{ijl}$) be the time taken for initiating VM *i* of type *l* from provider *j*.

Let a *new* user submit a service request at submission time $subT^{new}$ to the SaaS Provider. The *new* user offers a maximum price B^{new} (Budget) to SaaS provider with deadline DL^{new} *and Penalty Rate* β^{new}. Let $inDS^{new}$ and $outDS^{new}$ be the user requests required transfer in and transfer out data.

Let $Cost_{ijl}^{new}$ be the total cost incurred to the SaaS provider by processing the user request on VM *i* of type *l* uses resource provider *j*. Then, the profit $Prof_{ijl}^{new}$ gained by the SaaS provider is defined as:

$$\text{Prof}_{ijl}^{new} = B^{new} - Cost_{ijl}^{new} \; \forall i \in I, j \in J, l \in N_j \tag{3.1}$$

The total cost incurred to SaaS provider for accepting the new request consists of request's processing cost (PC_{ijl}^{new}), data transfer cost (DTC_{jl}^{new}), VM initiation cost (IC_{ijl}^{new}), and penalty delay cost (PDC_{ijl}^{new}) (to compensate for miss deadline). Thus, the total cost is given by processing the request on VM *i* of type *l* on IaaS provider *j*.

$$Cost_{ijl}^{new} = PC_{ijl}^{new} + DTC_{jl}^{new} + IC_{ijl}^{new} + PDC_{ij}^{new} \; \forall i \in I, j \in J, l \in N_j \tag{3.2}$$

The processing cost (PC_{ijl}^{new}) for serving the request is dependent on the new request's processing time ($procT_{ijl}^{new}$) and hourly price of VM_{il} offered by IaaS provider *j* . Thus, PC_{ijl}^{new} is given as:

$$PC_{ijl}^{new} = procT_{ijl}^{new} \times P_{jl}, \forall i \in I, j \in J, l \in N_j \tag{3.3}$$

Data transfer cost as described in Equation (3.4) includes cost for both data-in and data-out.

$$DTC_{jl}^{new} = inDS^{new} \times inPri_{jl} + outDS^{new} \times outPri_{jl} \; \forall j \in J, l \in N_j \tag{3.4}$$

The initiation cost (IC_{ij}^{new}) of VM *i* (type *l*) is dependent on the type of VM initiated in the data center of IaaS provider *j*.

$$IC_{ijl}^{new} = iniT_{ij} \times P_{jl}, \forall i \in I, j \in J, l \in N_j \tag{3.5}$$

In Equation (3.6), penalty delay cost (PDC_{ij}^{new}) is how much the service provider has to give discount to users for SLA(U) violation. It is dependent on the penalty rate (β^{new}) and penalty delay time (PDT_{ijl}^{new}) period. We model the SLA violation penalty as linear function which is similar to other related works [65][48][68].

$$PDC_{ijl}^{new} = \beta^{new} \times PDT_{ijl}^{new} \; \forall i \in I, j \in J, l \in N_j \tag{3.6}$$

To process any new request, SaaS provider either can allocate a new VM or schedule the request on an already initiated VM. If service provider schedules the new request on an already initiated $VM_{i,}$ the new request has to wait until *VM i* becomes available. The time for which the new request has to wait until it starts processing on VM *i* is $\sum_{k=1}^{K} procT_{ijl}^{k}$, where K is the number of request yet to be processed before the new request. Thus, PDT_{ijl}^{new} is given by:

$$PDT_{ijl}^{new} = \begin{cases} t + \sum_{k=1}^{K} procT_{ijl}^{k} + procT_{ijl}^{new} - DL^{new}, & if\ new\ VM\ is\ not\ initiated \\ procT_{ijl}^{new} + iniT_{ijl} + DTT_{ijl}^{new} - DL^{new}, & if\ new\ VM\ is\ initiated \end{cases} \tag{3.7}$$

DTT_{ijl}^{new} is the data transfer time which is the summation of time taken to upload the input ($inDT_{ill}^{new}$) and download the output data ($outDT_{ijl}^{new}$) from the VM_{il} on IaaS Provider *j*. The data transfer time is given by:

$$DTT_{ijl}^{new} = inDT_{ijl}^{new} + outDT_{ijl}^{new} \; \forall i \in I, j \in J, l \in N_j \tag{3.8}$$

Thus, the response time (T_{ijl}^{new}) for the new request to be processed on VM_{il} of IaaS Provider *j* is calculated in Equation (3.9) and consists of VM initiation time ($iniT_{ijl}^{new}$), request's service processing time ($procT_{ijl}^{new}$), data transfer time (DTT_{ijl}^{new}), and penalty delay time (PDT_{ijl}^{new}).

$$T_{ijl}^{new} = \begin{cases} \sum_{k=1}^{K} procT_{ijl}^{k} + procT_{ijl}^{new}, & if\ new\ VM\ is\ not\ initiated \\ procT_{ijl}^{new} + iniT_{ijl} + DTT_{ijl}^{new}, & if\ new\ VM\ is\ initiated \end{cases} \tag{3.9}$$

The investment return (ret_{ijl}^{new}) to accept new user request per hour on a particular VM_{il} on IaaS Provider *j* is calculated based on the profit ($prof_{ijl}^{new}$) and response time (T_{ijl}^{new}):

$$\text{ret}_{ijl}^{new} = \frac{prof_{ijl}^{new}}{T_{ijl}^{new}} \; \forall i \in I, j \in J, l \in N_j \tag{3.10}$$

3.3 Algorithms and Strategies

In this section, we present four strategies to analyse whether a new request can be accepted or not based on the QoS requirements and resource capabilities. Then, we propose three algorithms utilizing these strategies to allocate resources. In each algorithm, the admission control uses different strategies to decide which user requests to accept in order to cause minimal performance impact, avoiding SLA penalties that decrease SaaS provider's profit. The scheduling part of the algorithms determines where and which type of VM will be used by incorporating the heterogeneity of IaaS providers in terms of their price, service initiation time, and data transfer time.

3.3.1 Strategies

In this section, we describe four strategies for request acceptance: **a) initiate new VM, b) queue up the new user request at the end of scheduling queue of a VM, c) insert (prioritize) the new user request at the proper position before the accepted user requests and, d) delay the new user request to wait all accepted users to finish**. Inputs of all strategies are QoS parameters of the new request and resource providers' related information. Outputs of all strategies are admission control and scheduling related information, for example, which VM and in which resource provider the request can be scheduled. All flow charts in this section are in the context of each VM in each resource provider.

Initiate New VM Strategy

Figure 3.2 illustrates the flow chart of "initiate new VM strategy", which first checks for each type of VMs in each resource provider in order to determine whether the deadline of new request is long enough comparing to the estimated finish time. The estimated finish time depends on the estimated start time, request processing time, and VM initiation time.

If the new request can be completed within the deadline, the investment return is calculated (Equation 3.10). If there is value added according to the investment return, and then all related information (such as resource provider ID, VM ID, start time and estimated finish time) are stored into the potential schedule list. This strategy is represented as canInitiateNewVM () in algorithms.

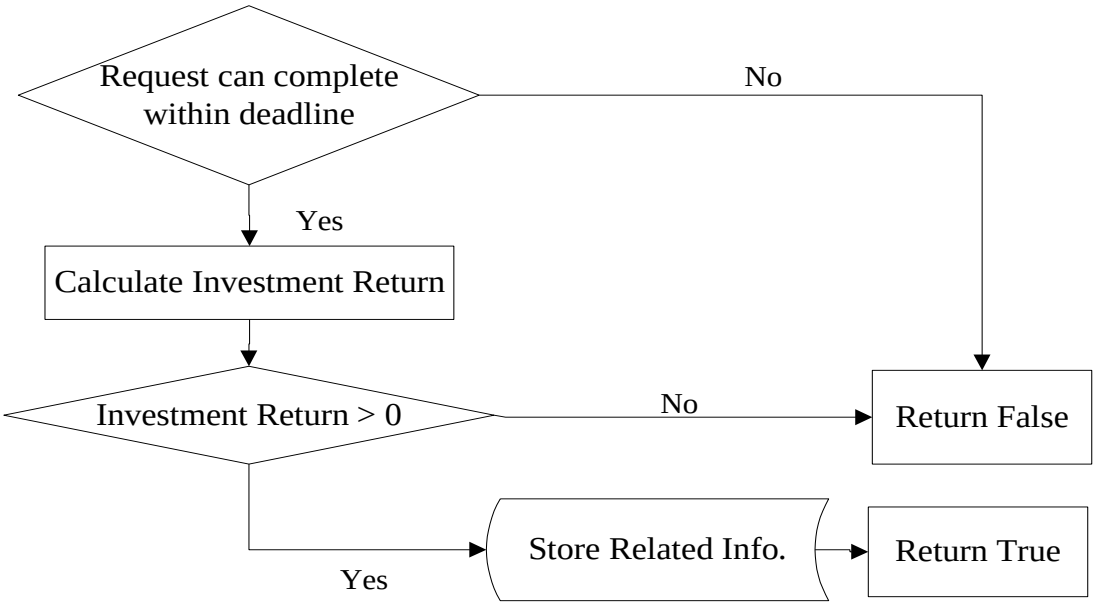


Figure 3.2 Flow Chart of 'Initiate new VM strategy'

Wait Strategy

Figure 3.3 illustrates the *wait strategy*, which first verifies each VM in each resource provider if the flexible time (fT_{ijl}^{new}) of the new request is enough to wait all accepted requests in vm_{il} to complete. The fT_{ijl}^{new} is given by Equation (3.11), in which K indicates total number of all accepted requests, I indicates all VMs, J indicates all resource providers, l indicates VM type, and N_j indicates all VM types provided by resource provider j.

$$fT_{ijl}^{new} = DL^{new} - \sum_{k=1}^{K} procT_{ijl}^{k} - subT^{new} \quad \forall i \in I, j \in J, k \in K, l \in N_j \qquad (3.11)$$

If new request can wait for all accepted requests to complete, and then the investment return is calculated and the remaining steps are the same as those in initiate new VM strategy. This strategy is called as *canWait ()* in algorithms.

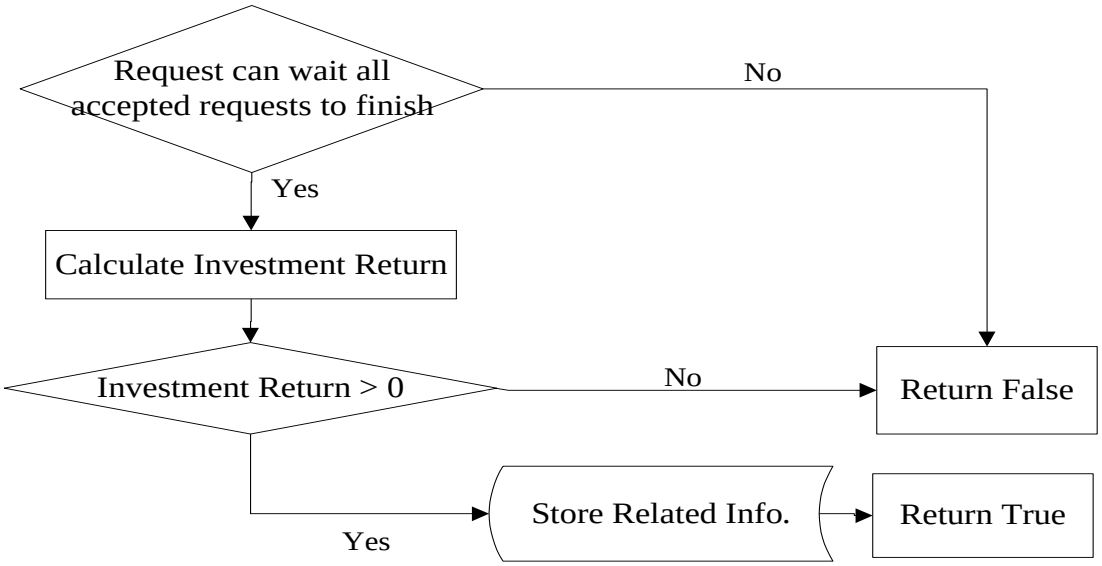


Figure 3.3 Flow Chart of 'wait strategy'

Insert Strategy

Figure 3.4 shows the flow chart of "*insert strategy"*, which first checks verifies if any accepted request u_k according to latest start time in vm_{il} can wait the new request to finish. If the flexible time of accepted request (fT_{ijl}^{k}) is enough to wait for a new user request to be completed then the new request is inserted before request *k*. The fT_{ijl}^{k} indicates the duration of request wait time with deadline and it is given by Equation (3.12), in which DL^k indicates the deadline of accepted request, *k* indicates the position of accepted request, and *K* indicates the total number of accepted user requests, *l* indicates the VM type and N_j indicates all VM types provided by resource provider *j*.

$$fT_{ijl}^{k} = DL^{k} - \sum_{\substack{n=1, \\ n \neq k}}^{K} procT_{ijl}^{n} - T_{ijl}^{new} - subT^{new} \quad \forall i \in I, j \in J, k \in K, l \in N_j \qquad (3.12)$$

If there is an already accepted request u^k that is able to wait for the new user request to complete, the strategy checks if the new request can complete before its deadline. If so, u^{new} gets priority over u^k, then the algorithm calculates the investment return and the remaining steps are the same as those in *initiate new VM strategy*. This strategy is presented as *canInsert ()* in algorithms.

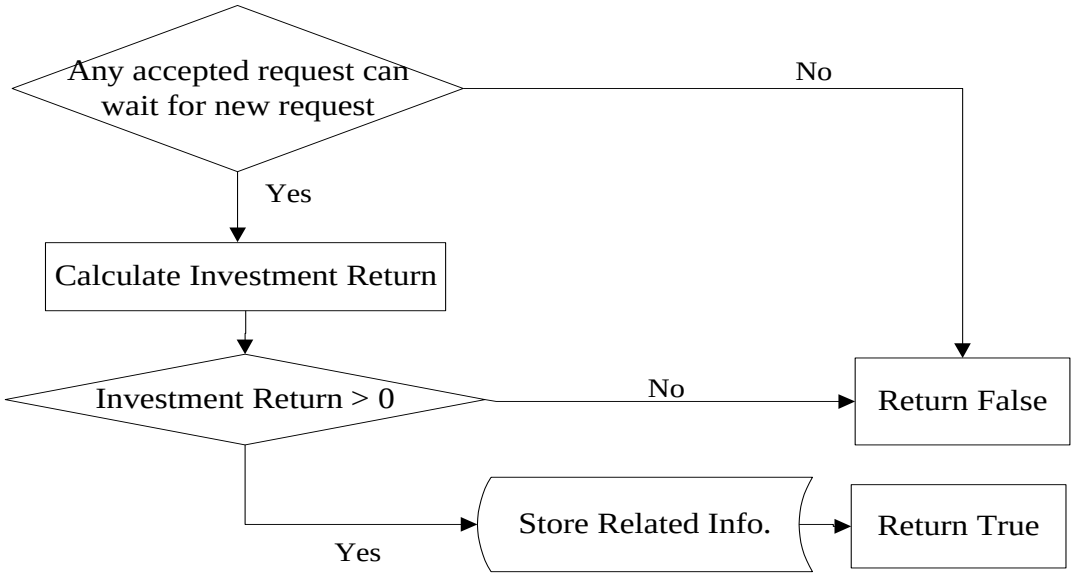


Figure 3.4 Flow Chart of 'insert strategy'

Penalty Delay Strategy

Figure 3.5 describes the flow chart of "penalty delay strategy", which first checks if the new user request's budget is enough to wait for all accepted user requests in vmi to complete after its deadline. Equation (3.1) is used to check whether budget is enough to compensate the penalty delay loss, and then the investment return is calculated and the remaining steps are the same as those in initiate new VM strategy. This strategy is presented as funciton canPenaltyDelay() in algorithms.

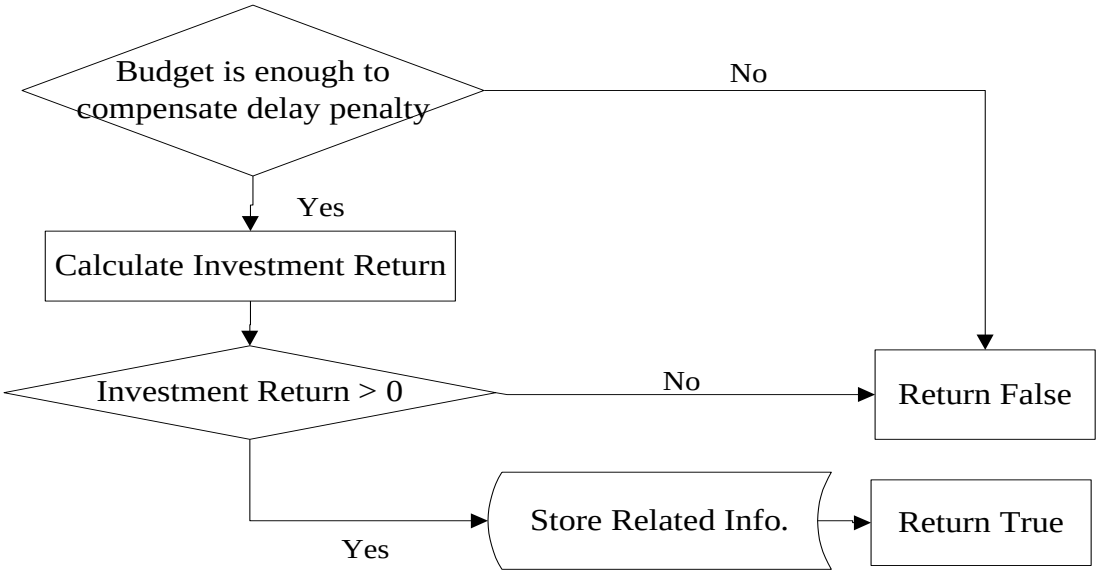


Figure 3.5 Flow Chart of 'penalty delay strategy'

3.3.2 Proposed Algorithms

A service provider can increase the profit by reducing the infrastructure cost, which depends on the number and type of initiated VMs in IaaS providers' data centre. Therefore, our algorithms are designed to minimize the number of VMs by maximizing the utilization of already initiated VMs. The assumption here is that SaaS provider will offer proper security protection for business data, especially when data is copied to VMs that are already created. In this section, based on above strategies we propose three algorithms, which are *ProfminVM, ProfRS*, and *ProfPD:*

- Maximizing the profit by minimizing the number of VMs (*ProfminVM*).
- Maximizing the profit by rescheduling (*ProfRS*).
- Maximizing the profit by exploiting the penalty delay (*ProfPD*).

Maximizing the Profit by Minimizing the number of VMs (ProfminVM)

Algorithm 1 describes the ProfminVM algorithm, which involves two main phases: a) admission control and b) scheduling.

In admission control phase, the algorithm analyses if the new request can be accepted either by queuing it up in an already initiated VM or by initiating a new VM. Hence, firstly, it checks if the new request can be queued up by waiting for all accepted requests on any initiated VM - using Wait Strategy (Step 3). If this request cannot wait in any initiated VM, then the algorithm checks if it can be accepted by initiating a new VM provided by any IaaS provider - using Initiate New VM Strategy (Step 8). If a SaaS provider does not make any profit by utilizing already initiated VMs nor by initiating a new VM to accept the request, then the algorithm rejects the request (Step 9). Otherwise, the algorithm gets the maximum investment return from all of the possible solutions (Step 13). The decision also depends on the minimum expected investment return (expInvRetijlnew) of the SaaS provider. If the investment return ret_{ijl}^{new} is more than the SaaS provider's expInvRetijlnew, the algorithm accepts the new request (Step 14, 15), otherwise it rejects the request (Step 16, 17). The expected investment return ratio w is customized by SaaS providers. The expected investment return (expInvRetijlnew) is given by Equation (3.13):

$$\mathrm{expInvRet}_{ijl}^{new} = \omega \times \frac{Cost_{ijl}^{new}}{T_{ijl}^{new}} \quad \forall i \in I, j \in J, l \in N_j \tag{3.13}$$

The scheduling phase is the actual resource allocation and scheduling based on the admission control result; if the algorithm accepts the new request, the algorithm first finds out in which IaaS Provider rpj and which VM vmi a SaaS provider can gain the maximum investment return by extracting information from PotentialScheduleList (Step 20). If the maximum investment return is gained by initiating a new VM (Step 22), then the algorithm initiates a new VM in the referred resource provider (rpj), and schedule the request to it. Finally, the algorithm schedules the new request on the referred VM (vmi) (Step 23). The time complexity of this algorithm is O(KIJ+KI), where K indicates the total number of accepted requests, I indicates the total number of initiated matched type of VMs and J indicates the number of resource providers.

Algorithm 1. Pseudo-code for ProfminVM algorithm

```
Input: New user's request parameters (unew), expInvRetijnew
Output: Boolean
Functions:
admissionControl( ) {
 1.        If ( there is any initiated VM ) {
 2.            For each vmi in each resource provider rpj {
 3.                If (! canWait ( unew, vmi ) ) {
 4.                    continue;
 5.                }
6.            }
 7.        }
 8.        Else If (! canInitiateNew(unew, rpj))
9.                Return reject
10.       If (PotentialScheduleList is empty)
11.           Return reject
12.       Else {
13.           Get the max[retijnew, SDij ] in PotentialScheduleList
14.           If ( max(retijnew) ≥ expInvRetijnew )
15.               Return accept
16.           Else
17.               Return reject
18.       }
```

```
19.             }
}
schedule ( ) {
20.             Get the [retmaxnew, SDmax ] in  maxRet(PotentialScheduleList)
21.             If ( SDmax is initiateNewVM)
22.                 initiateNewVM in rpj
23.             Schedule the unew in VMmax in  rpmax according to SDmax.
 }
```

Maximizing the Profit by Rescheduling (ProfRS)

In *ProfminVM* algorithm, a new user request does not get priority over any accepted request. This inflexibility affects the profit of a SaaS provider since many urgent and high budget requests will be rejected. Thus, *ProfRS* algorithm reschedules the accepted requests to accommodate an urgent and high budget request. The advantage of this algorithm is that a SaaS provider accepts more users utilizing initiated VMs to earn more profit.

Algorithm 2 describes *ProfRS* algorithm. In the **admission control** phase, the algorithm analyses if the new request can be accepted by waiting in an already initiated VM, inserting into an initiated VM, or initiating a new VM. Hence, firstly it verify if new request can wait all accepted requests in any already initiated VM - invoking *Wait Strategy* (Step 3). If the request cannot wait, then it checks if the new request can be inserted before any accepted request in an already initiated VM -using *Insert Strategy* (Step 4). Otherwise the algorithm checks if it can be accepted by initiating a new VM provided by any IaaS provider - using *Initiate New VM Strategy* (Step 5). If a SaaS provider does not make sufficient profit by any strategy, the algorithm *rejects* this user request (Step 10, 11). Otherwise the algorithm gets the maximum return from all analysis results (Step 15). The remaining steps are the same as those in *ProfminVM* algorithm. The time complexity of this algorithms is O ($KIJ+IK^2$), where K indicates the total number of accepted requests, I indicates the total number of initiated matched type of VMs and J indicates the number of resource providers.

Algorithm 2. Pseudo-code for *ProfRS* algorithm

Input: New user's request parameters (u^{new}), $expInvRet_{ij}^{new}$

Output: Boolean

Functions:

admissionControl( ) **{**

1. **If** (there is any initiated VM) {

2. **For each** vm_i in each resource provider rp_j {

3. **If** (**!** ***canWait*** (u^{new}, vm_i)) {

4. **If** (**!** ***canInsert*** (u^{new}, vm_i)) {

5. **If** (**!** ***canInitiateNew***(u^{new}, rp_j)) {

6. **continue;**

7. }

8. }

9. }

10. **Else If** (**!** ***canInitiateNew***(u^{new}, rp_j))

11. **Return** ***reject***

12. **If** (PotentialScheduleList is empty)

13. **Return** ***reject***

14. **Else** {

15. Get the ***max***[ret_{ij}^{new}, SD_{ij}] in PotentialScheduleList

16. **If** (***max***(ret_{ij}^{new}) $\geq$ $expInvRet_{ij}^{new}$)

17. **Return** ***accept***

18. **Else**

19. **Return** ***reject***

20. }

}

schedule ( ) {

21. Get the [ret_{max}^{new}, SD_{max}] in *maxRet*(PotentialScheduleList)

22. **If** (SD_{max} is initiateNewVM)

23. initiateNewVM in rp_j

24. Schedule the u^{new} in VM_{max} in rp_{max} according to SD_{max}.

}

Maximizing the Profit by exploiting penalty delay (ProfPD)

To further optimize the profit, we design the algorithm *ProfPD* by considering delaying the new requests to accept more requests.

Algorithm 3 describes *ProfPD* algorithm. In the **admission control** phase, we analyse if the new user request can be processed by queuing it up at the end of an already initiated VM, by inserting it into an initiated VM, or by initiating a new VM. Hence, firstly the algorithm check if the new request can wait all accepted requests to complete in any initiated VM - invoking *Wait Strategy* (Step 3). If the request cannot wait, then it checks if the new request can be inserted before any accepted request in any already initiated VM -using *Insert Strategy* (Step 4). Otherwise the algorithm checks if the new request can be accepted by initiating a new VM provided by any resource provider - using *Initiate New VM Strategy* (Step 5) or by delaying the new request with penalty compensation - using *Penalty Delay Strategy* (Step 7). If a SaaS provider does not make sufficient profit by any strategy, the algorithm *rejects* the new request (Step 14). Otherwise, the request is accepted and scheduled based on the entry in *PotentialScheduleList* which gives the maximum return (Step 23). The rest of the steps are the same as those in *ProfminVM*. The time complexity of this algorithms is $O(KIJ+IK^2)$, where K indicates the total number of accepted requests, I indicates the total number of initiated matched type of VMs and J indicates the number of resource providers.

Algorithm 3. Pseudo-code for *ProfPD* algorithm

Input: New user's request parameters (u^{new}), $expInvRet_{ij}^{new}$

Output: Boolean

Functions:

admissionControl( ) **{**

1. **If** (there is any initiated VM) {

2. **For each** vm_i in each resource provider rp_j {

3. **If** (**!** ***canWait*** (u^{new}, vm_i)) {

4. **If** (**!** ***canInsert*** (u^{new}, vm_i)) {

5. **If** (**!** ***canInitiateNew***(u^{new}, rp_j))

6. **continue;**

7. **If** (**!** ***canPenaltyDelay***(u^{new}, rp_j))

8. **continue;**

9. }

10. }

11. }

12. }

13. **Else If** (**!** ***canInitiateNew***(u^{new}, rp_j))

14. **Return** ***reject***

15. **If** (PotentialScheduleList is empty)

16. **Return** ***reject***

17. **Else** { Get the ***max***[ret_{ij}^{new}, SD_{ij}] in PotentialScheduleList

18. **If** (***max***(ret_{ij}^{new}) $\geq$ $expInvRet_{ij}^{new}$)

19. **Return** ***accept***

20. **Else**

21. **Return** ***reject***

22. }

}

schedule ( ) {

23. Get the [ret_{max}^{new}, SD_{max}] in *maxRet*(PotentialScheduleList)

24. **If** (SD_{max} is initiateNewVM)

25. initiateNewVM in rp_j

26. Schedule the u^{new} in VM_{max} in rp_{max} according to SD_{max}.

}

3.4 Performance Evaluation

In this section, we first explain the reference algorithms and then describe our experiment methodology, followed by performance evaluation results, which includes comparison with reference algorithms and among our proposed algorithms.

As existing algorithms in the literature are designed to support scenarios different to those considered in our work, we are comparing proposed algorithms to reference algorithms exhibiting lower and up bounds: *MinResTime* and *StaticGreedy.*

- The *MinResTime* algorithm selects the IaaS provider where new request can be processed with the earliest response time to avoid deadline violation and profit loss, therefore it minimizes the response time for users. Thus, it is used to know how fast user requests can be served.
- The *StaticGreedy algorithm* assumes that all user requests are known at the beginning of the scheduling process. In this algorithm, we select the most profitable schedule obtained by sorting all the requests either based on *Budget or Deadline*, and then using *ProfPD* algorithm. Thus, the profit obtained from *StaticGreedy* algorithm acts as an upper bound

of the maximum profit that can be generated. It is clear that assumption taken in *StaticGreedy* algorithm is not possible in reality as all the future requests are not known.

3.4.1 Experimental Methodology

We use CloudSim [80] as a Cloud environment simulator and implement our algorithms within this environment. We observe the performance of the proposed algorithms from both users' and SaaS providers' perspectives. From users' perspective, we observe how many requests are accepted and how fast user requests are processed (we call it average response time). From SaaS providers' perspective, we observe how much profit they gain and how many VMs they initiate. Therefore, we use four performance measurement metrics: total profit, average request response time, number of initiated VMs, and number of accepted users. All the parameters from both users' and IaaS providers' side used in the simulation study are given in following sub-sections:

Users' side

We examine our algorithms with 5000 users. From the user side, five parameters (deadline, service time, budget, arival rate and penalty rate factor) are varied to evaluate their impact on the performance of our proposed algorithms. Request arrival rate follows poisson distribution as many previous works [100][101] model arrival rate as poisson distribution. Similar as other works, we use a normal distribution to model all parameters (standard deviation =(1/2)xmean), because there is no available workload specifiying these parameters. Equation 3.14 is used to calculate the **deadline** (DL_{ijl}^{new}). α is the factor which is used to vary the deadline from "very tight" (α=0.5) to "very relax" (α=2.5). $estprocT_{ijl}^{new}$ indicates the new service request's estimated processing time.

$$DL_{ijl}^{new} = \alpha \times estprocT_{ijl}^{new} + estprocT_{ijl}^{new} \quad \forall i \in I, j \in J, l \in N_j \qquad (3.14)$$

Service time is estimated based on the Request Length (*MI*) and the Millions of Instruction per Second (*PS*) of a VM. The mean Request Lengths are selected between 10^6 MI ("very small") to $5x10^6$ MI ("very large"), while *MIPS* value for each VM type is fixed.

In common economic models, **budget** is generated by random numbers [65]. Therefore, we follow the same random model for budget, and vary it from "very small" (mean=0.1$) to "very large" (mean=1$). We choose budget factor up to 1, because the trend of results does

not show any change after 1. Five different types of **request arrival rate** are used by varying the mean from 1000 to 5000 users per second. The penalty rate β (the same as in Equation 3.1) is modelled by Equation 3.15. It is calculated in terms of how long a user is willing to wait (r) in proportion to the deadline when SLA is violated. In order to vary the penalty rate, we vary the mean of r from "very small" (4) to "very large" (44).

$$\beta = \frac{B^{new}}{DL^{new} \times r} \quad \forall i \in I, j \in J \tag{3.15}$$

Resource Providers' side

We consider five resouce providers – IaaS providers, which are Amazon EC2[92], GoGrid[94], Microsoft Azure[96], RackSpace[95] and IBM[97]. To simulate the effect of using different VM types, MIPS ratings are used. Thus, a MIPS value of an equivalent processor is assigned to the request processing capability of each VM type. The price schema of VMs follows the price schema of GoGrid [94] , Amazon EC2 [92], RackSpace [95], Microsoft Azure [96], and IBM [97]. The detail resource characteristics which are used for modelling IaaS providers are shown in *Table 3.1*. The three different types of average VM **initiation time** are used in the experiment, and the mean initiation time varies from 30 seconds to 15 minutes (standard deviation= (1/2)xmean). The mean of initiation time is calculated by conducting real experiments of 60 samples on GoGrid [94] and Amazon EC2 [92] done for four days (2 week days and 2 weekend days).

3.4.2 Performance Results

In this section, we first compare our proposed algorithms with reference algorithms by varying number of users. Then, the impact of QoS parameters on the performance metrics is evaluated. Finally, robustness analysis of our algorithm is presented. All of the results present the average obtained by 5 experiment runs. In each experiment we vary one parameter, and others are given constant mean vaule. The constant mean, which are used during experiment, are as follows: arrival rate=5000 requests/sec, deadline=2*estprocT, budget=1 $, requst length= $4x10^6$ MI, and penalty rate factor (r) =10.

Table 3.1 The summary of resource provider characteristics.

Provider	VM Types	VM Price ($/hour)
Amazon EC2	Small / Large	0.12/0.48
GoGrid	1 Xeon / 4 Xeon	0.19/0.76
RackSpace	Windows	0.32
Microsoft Azure	Compute	0.12
IBM	VMs 32-bit (Gold)	0.46

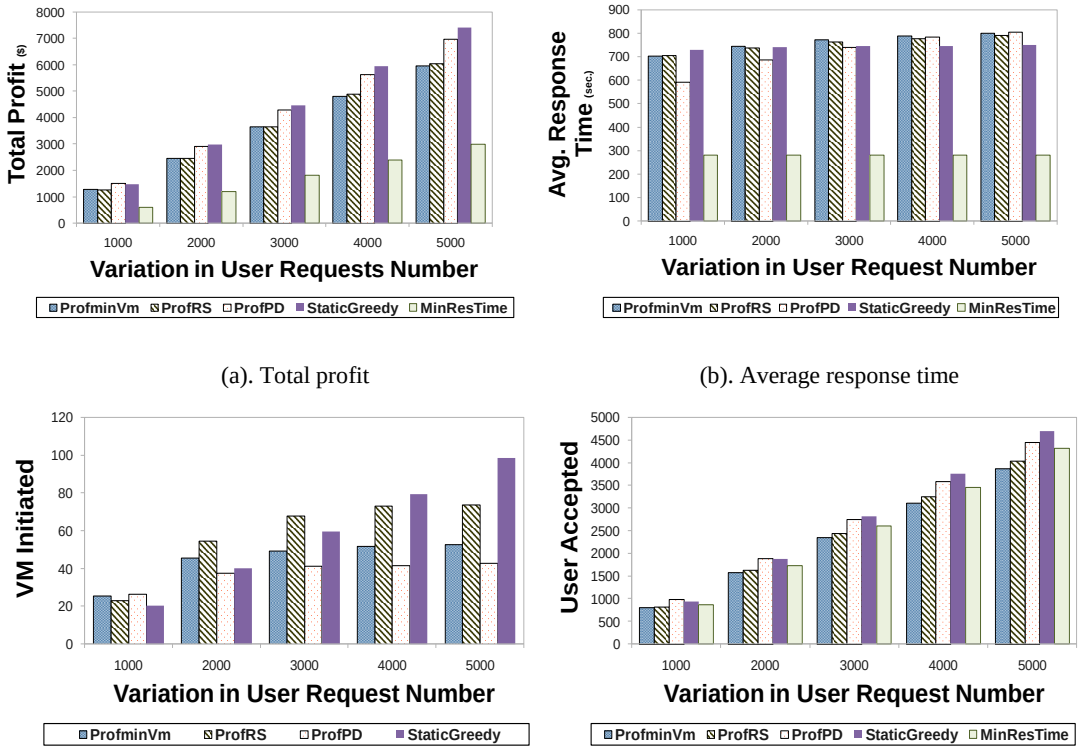
Comparison with Reference Algorithms

To observe the overall performance of our algorithms, we vary the number of users from 1000 to 5000 without varying other factors such as deadline and budget. *Figure 3.6* presents the comparison of our proposed algorithms with reference algorithms *StaticGreedy* and *MinResTime* in terms of the four performance metrics. When the number of user requests varies from 1000 to 5000, for each algorithm the total profit and average response time has increased, because of more user requests.

Figure 3.6 shows that *ProfPD* earns 8% less profit (Requests = 5000) for SaaS provider than *StaticGreedy* which is used as the upper bound. That is because in the case of *StaticGreedy*, all the user requests are already known from the beginning to the SaaS provider. The base algorithm *MinResTime* has smaller (two third of *StaticGreedy*) response time, but earns less profit (approximately half of *ProfPD*). These observations indicate the trade-off between response time and profit, which SaaS provider has to manage while scheduling requests.

Figure 3.6a shows that the *ProfPD* achieves (15%) more profit over *ProfRS* and (17%) over *ProfminVM* by accepting (10%, 15%) more user requests and initiating (19%, 40%) less number of VMs, when number of users changes from 1000 to 5000. When number of users is 1000 *ProfPD* earns 4% and 15% more profit over *ProfminVM* and *ProfRS* respectively. When the user number is increased from 1000 to 5000, the profit difference between *ProfPD* and other two algorithms became larger. This is because when the number of requests increased, the number of users being accepted increased by utilizing initiated VMs. If all requests are known before scheduling, then *StaticGreedy* is the best choice for maximizing profit, however, in the real Cloud computing market, these are unknown. Therefore, a SaaS provider should use *ProfPD*, however, *ProfRS* is a better choice for a SaaS provider in

comparison with *ProfminVM*. In addition, the *ProfPD* is effective in maximizing profit in heavy workload situations.



(a). Total profit

(b). Average response time

(c). Number of initiated VMs

(d). Number of accepted users

Figure 3.6 Overall algorithms' performance during variation in number of user requests

Figure 3.6b shows that our algorithms' trends of response time increase from 1000 users to 5000 users because of increasing in processing of user requests per VM. When there is smaller number of requests, the difference between different algorithm's response times becomes significant. For example, with 1000 requests, *ProfPD* gives users 16% lower response time than *ProfminVM* and *ProfRS*, and even accept more requests. This is because *ProfPD* scheduled less number of users per VM, thus user's experience less delay. In other scenarios the reason for lower response time is smaller initiation time. *ProfminVM* provides the lowest response time compared to others, because it can serve a new user with new VMs.

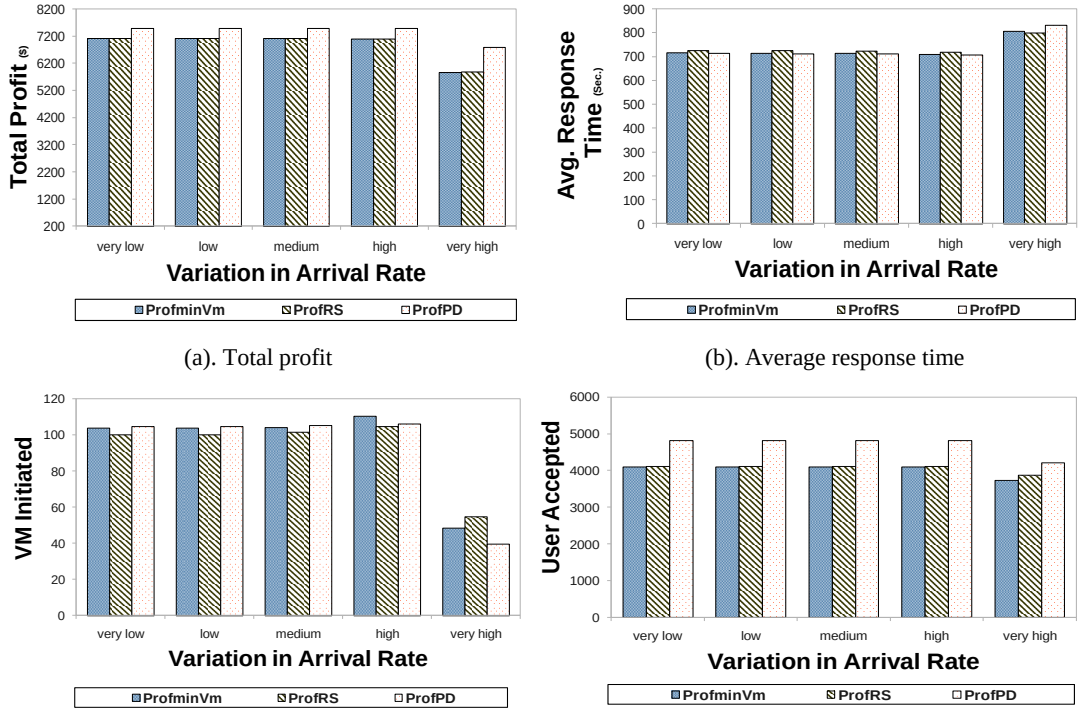
Impact of QoS parameters

In the following sections, we examine various experiments by varying both user and resource provider side's SLA properties to analyse the impact of each parameter.

1) Impact of variation in **arrival rate**

To observe the impact of arrival rate in our algorithms, we vary the arrival rate factor, while keeping all other factors such as deadline, budget as the same. All experiments are conducted with 5000 user requests. It can be seen from *Figure 3.7* that when arrival rate is "very high", the performance of *ProfminVM*, *ProfRS*, and *ProfPD* are affected significantly. The overall trend of profit is decreasing and the response time is increasing because when there is more user arrival per second, the service capability is decreased due to fewer new VM instantiations.

Figure 3.7a shows that the *ProfPD* achieves the highest profit (maximum 15% more than *ProfminVM* and *ProfRS*) by accepting (45%) more users and initiating the least number of VMs (19% less than *ProfminVM*, 28% less than *ProfRS*) when arrival rate increases from "very small" to "very large". This is because *ProfPD* accept users with existing machines with penalty delay. In the same scenario, *ProfminVM* and *ProfRS* achieve similar profit, but *ProfRS* accepts 4% more requests with 13% more VMs than *ProfminVM*. Therefore, in this scenario *ProfPD* is the best choice for a SaaS provider. However, when arrival rate is "very large", and the number of VM is limited, *ProfRS* is a better choice compared to *ProfminVM* because although it provides similar profit as *ProfminVM*, it accepts more requests, leading to market share expanding.



(a). Total profit

(b). Average response time

(c). Number of initiated VMs

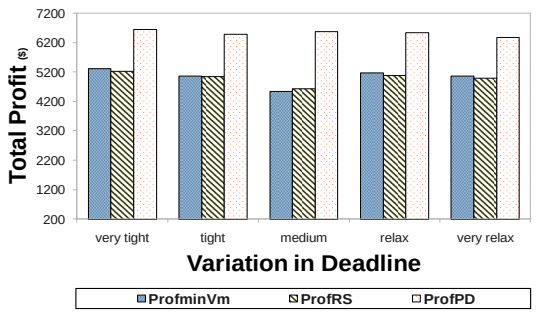
(d). Number of accepted users

Figure 3.7 Impact of arrival rate variation

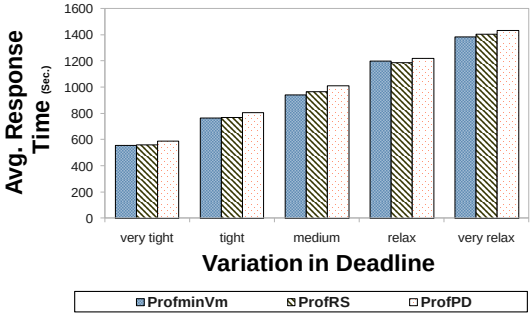
Figure 3.7b shows that the *ProfPD* achieves in the smallest response time and accepted more number of users with less number of VMs except when arrival rate is very high. Even in the case of high arrival rate, the difference between response time from *ProfPD* and its next competitor is just 3%. *ProfminVM* and *ProfRS* have similar response times. However, there is a drastic increase in response time when the arrival rate is very high because more requests are accepted per VM which delays the processing of requests. It is safe to conclude that even considering the response time constraints from users, the first choice for a SaaS provider is still the *ProfPD*.

2) Impact of variation in **deadline**

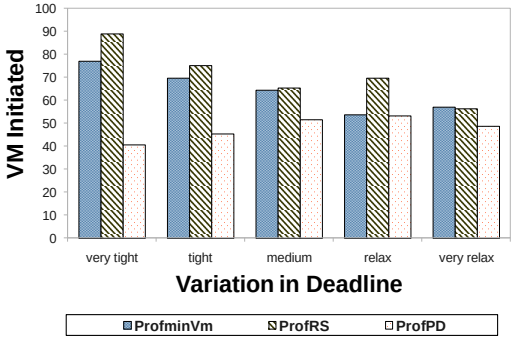
To investigate the impact of deadline in our algorithms, we vary the deadline, while keeping all other factors such as arrival rate and budget fixed. *Figure 3.8a* shows that the *ProfPD* achieved the highest profit (45% over *ProfminVM* and 41% over *ProfRS*) by accepting 33% more user requests (*Figure 3.8d*) and initiating 52% less VMs (Fig. 8c)". In some scenarios, *ProfminVM* provides higher profit than *ProfRS*, for example, when deadline is "very tight", because *ProfRS* accepted requests with larger service time, which occupy the space for accepting other requests.



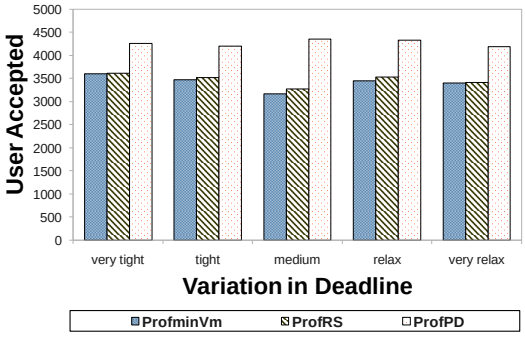
(a). Total profit



(b). Average response time



(c). Number of initiated VMs



(d). Number of accepted users

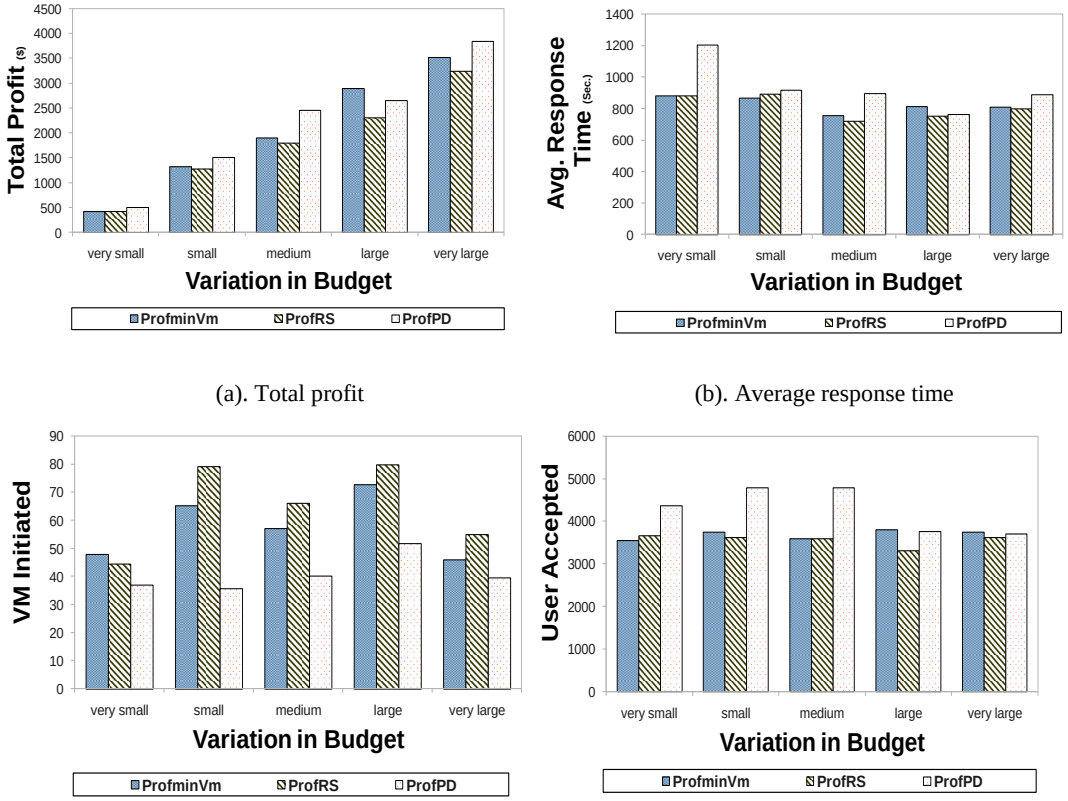
Figure 3.8 Impact of deadline variation

Figure 3.8b shows that when deadline is relaxed, *ProfPD* results in 4% higher average response time than in the case of *ProfminVM* and *ProfRS*. The *ProfPD* has larger response time because of the two factors governing response time, i.e., request's service time and VM initiation time. It can be seen from *Figure 3.8d* that *ProfPD* always requires less VMs, to process more requests. Thus, when service time is comparable to the VM initiation time, the response time will be lower. When the VM initiation time is larger than the service time, the response time is affected by the number of initiated VMs.

3) Impact of variation in **budget**

Figure 3.9 shows variation of budget impacts our algorithms, while keeping all other factors such as arrival rate and deadline fixed. *Figure 3.9a* shows that when budget is varies from "very small" to "very large", in average the total profit by all the algorithms has increased, and response time has decreased since less requests are processed using more VMs. From *Figure 3.9a*, it can be observed that *ProfPD* gains the highest profit for SaaS provider except when budget is "large". In case of scenario when budget is "large", *ProfminVM* provides the highest profit (20%) over other algorithms by accepting similar number of requests while initiating more VMs without penalty delay. This is due to an increase in the *Penalty Delay Rate (β)* (Equation15) with the budget raise. Between *ProfminVM* and *ProfRS*, *ProfminVM* provides more profit in all scenarios. Therefore, in this scenario a SaaS provider should consider *ProfPD*, *ProfminVM* compared with *ProfRS*.

In the case of response time (*Figure 3.9b*), *ProfPD* on average delayed the processing of request for the longest time (e.g. 33% bigger response time for "very small" budget scenario) even though it processed more user requests and initiated less VMs. However, when budget is "large", the response time provided by *ProfminVm* is the longest even though it accepts similar number of users as *ProfPD*. This anomaly caused by the contribution of VM initiation time which becomes very significant when *ProfRS* initiated large number of VMs.



(a). Total profit

(b). Average response time

(c). Number of initiated VMs

(d). Number of accepted users

Figure 3.9 Impact of budget variation

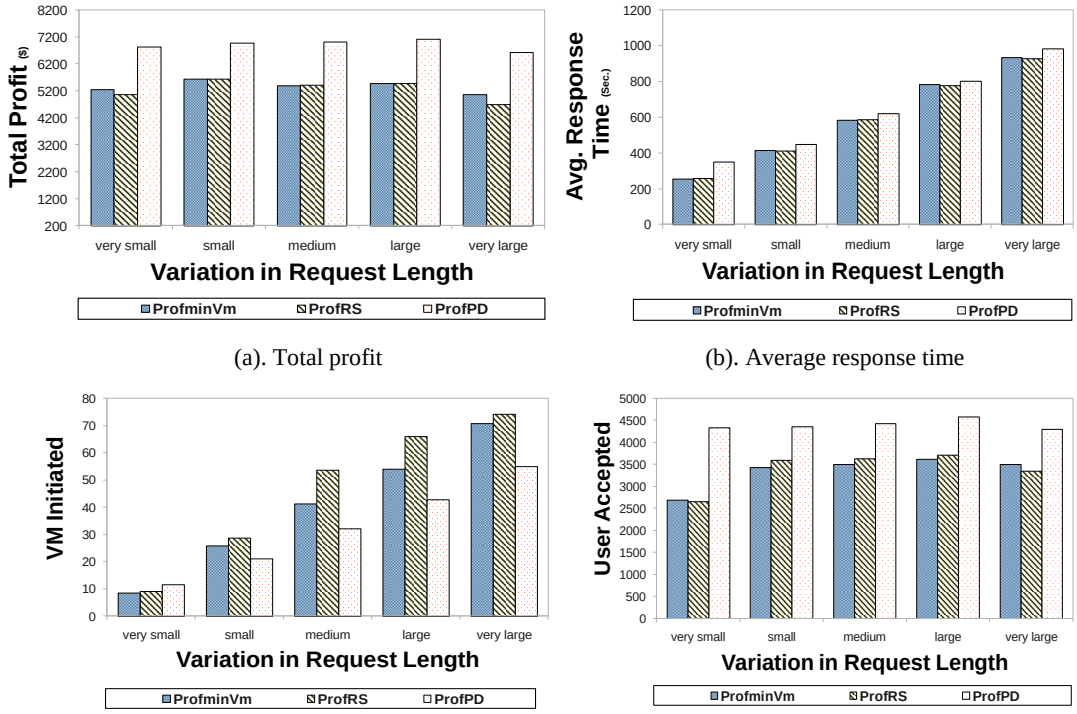
4) Impact of variation in **service time**

Figure 3.10 shows how service time impacts our algorithms, while keeping all other factors such as arrival rate and deadline as the same. In order to vary the service time, five classes of request length (*MI*) are chosen from "very small" (10^6MI) to "very large" ($5x10^6$MI).

Figure 3.10a shows that the total profit by all algorithms has slightly decreased but response time increased rapidly when the request length varies from "very small" to "very large". *ProfPD* achieves the highest profit among other algorithms. For example, in the case of "very large" request length scenario, *ProfPD* generated about 30% more profit than other algorithms by accepting 24% more requests (*Figure 3.10d*) and initiating 32% (*Figure 3.10c*) less VMs. In addition, *ProfminVM* and *ProfRS* achieve similar profit in most of the cases. Therefore, the *ProfPD* is the best solution for any size of requests.

In addition, it can be observed from Fig. 10b that *ProfPD* provides only a slightly higher response time (almost 6%) than others except when the request size is very small. When

request size is very small, the response time provided by *ProfPD* becomes 27% bigger than others, because it accepts 63% more user requests with 22% more VMs, leading to more requests waiting for processing on each VM.



(a). Total profit

(b). Average response time

(c). Number of initiated VMs

(d). Number of accepted users

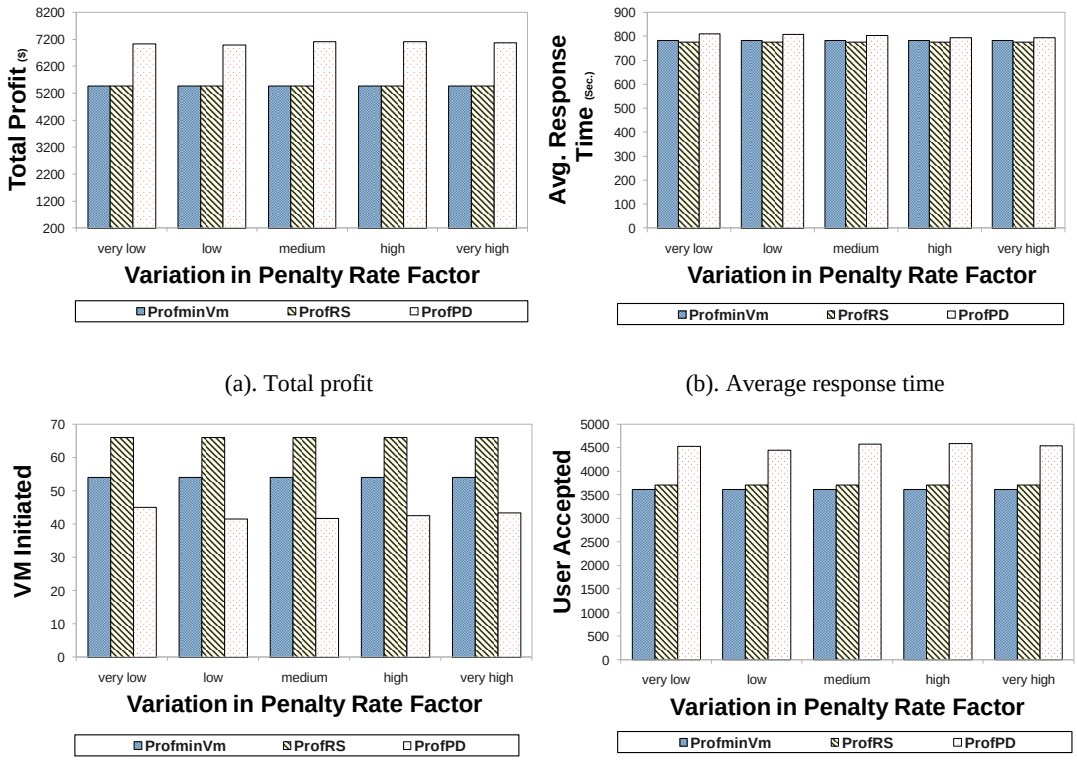
Figure 3.10 Impact of request length variation

5) Impact of variation in **penalty rate**

In this section, we investigate how penalty rate (β) impacts our algorithms. The penalty rate (Equation 3.15) depends on how long user is willing to wait (r), which is defined as ***penalty rate factor*** in our chapter. Therefore, when the penalty rate factor (r) is large, the penalty rate is small. All the results are presented in *Figure 3.11*.

In can be observed from *Figure 3.11* that only *ProfPD* shows some effect of variation in penalty rate since this is the only algorithm which uses Penalty Delay strategy to maximize the total profit. The total profit (*Figure 3.11a*) and average response time (*Figure 3.11b*) are only slightly decreased when the (r) is varied from "very low" to "very high". In almost all scenarios, *ProfPD* achieves 29% more profit over others by accepting 22% more requests and initiating 30% less VMs. In addition, when the penalty rate varies from "very low" to very high", the response time slightly decreased. This is because *ProfPD* accepts a little bit

less requests with similar number of VMs. Thus, the number of requests waiting in each VM becomes smaller, leading to faster response time for each request.



(a). Total profit

(b). Average response time

(c). Number of initiated VMs

(d). Number of accepted users

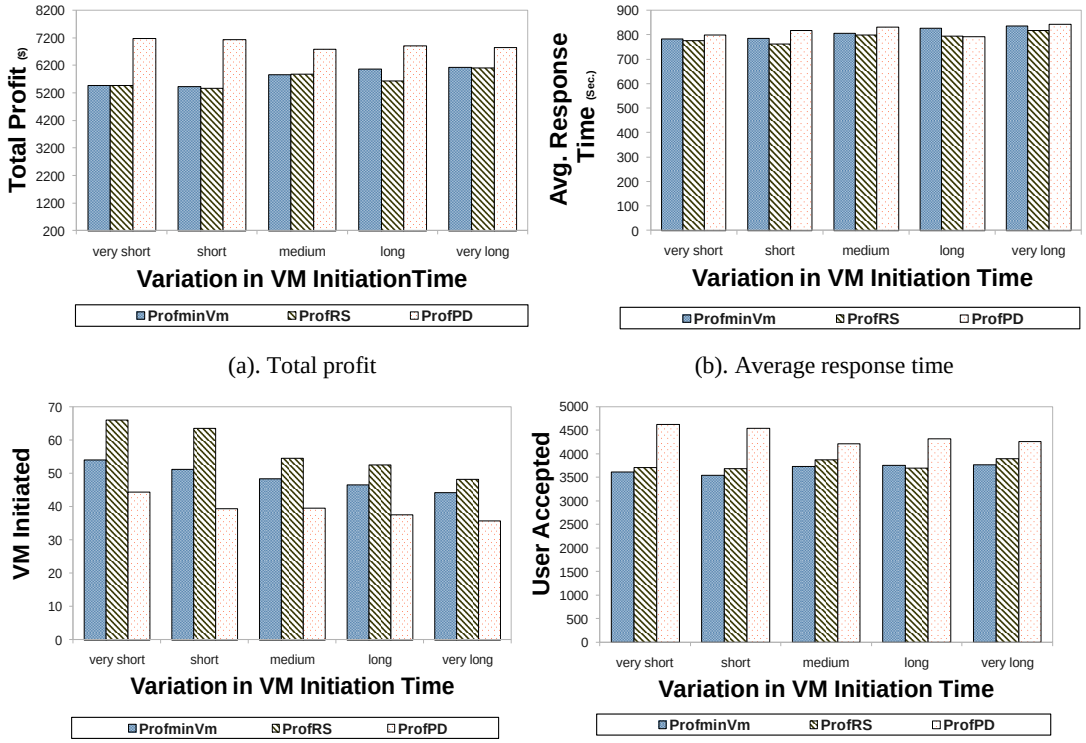
Figure 3.11 Impact of penalty rate factor variation

6) Impact of variation in **Initiation Time**

In this section, we analyse the variation of initiation time impacts our algorithms. *Figure 3.12a* illustrates that with increase in initiation time the total profit achieved by all the algorithms decreases slightly while response time has increased a little bit. Due to increase in initiation time, the number of initiated VMs (*Figure 3.12c*) has decreased rapidly due to the contribution of initiation time in SaaS providers cost (spending). In all the scenarios, *ProfPD* achieves highest profit over others by accepting 17% more requests (*Figure 3.12d*) and with 37% less initiated VMs. Therefore, *ProfPD* is the best choice for a SaaS provider in this scenario.

The response time offered by *ProfPD* is slightly higher than others in most of cases, because it accepted more users with less number of VMs, in other word, a VM required to serve more number of users, leading to delay in request processing. The response time of *ProfPD* is the lowest in this scenario; because of large initiation time of VM, the response time is also

increased with each initiated VM. However, the contribution to delay in processing of requests, due to more number of requests per VM also increases. This leads to higher response time in the scenario when the initiation time is "very long".



(a). Total profit

(b). Average response time

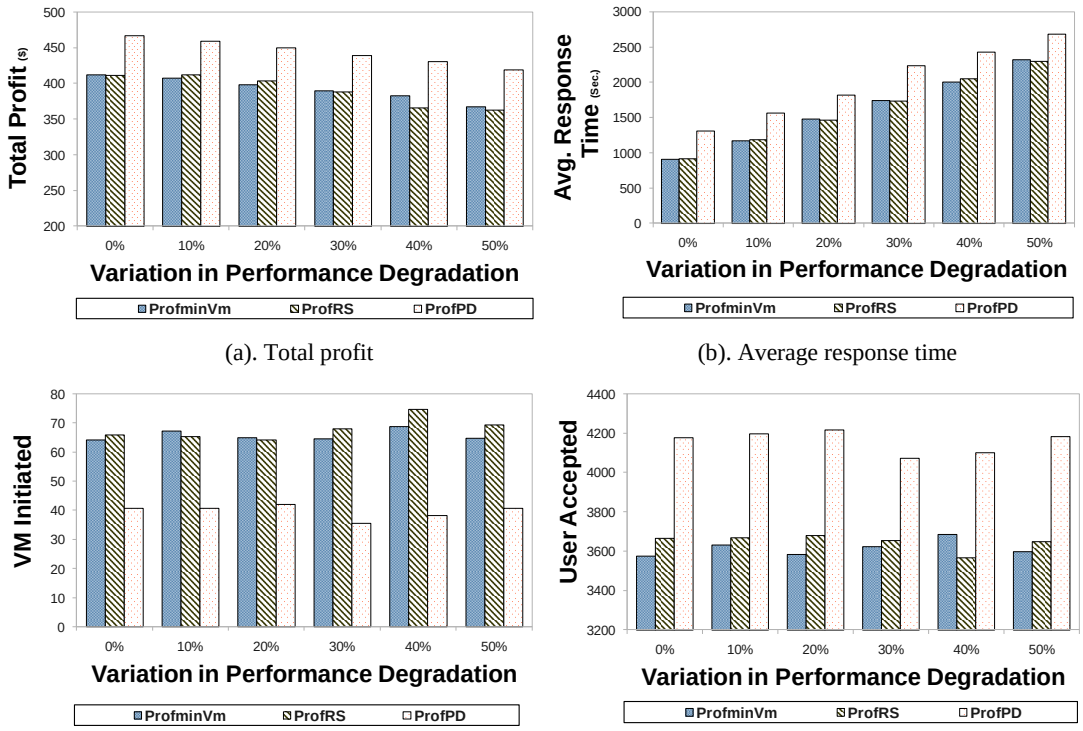
(c). Number of initiated VMs

(d). Number of accepted users

Figure 3.12 Impact of initiation time variation

Robustness Analysis

In order to evaluate the robustness of our algorithms, we run some experiments by reducing the actual performance of VMs in the SLA(R) promised by IaaS providers. This performance degradation has been observed by previous research study in Cloud computing environments [98]. This experiment is conducted also to justify the inclusion of compensation (penalty) clauses in SLAs which is absent in current IaaS providers' SLAs [93]. We modelled the reduced performance using a normal distribution with average variation between mean varies 0% and 50%.



(a). Total profit

(b). Average response time

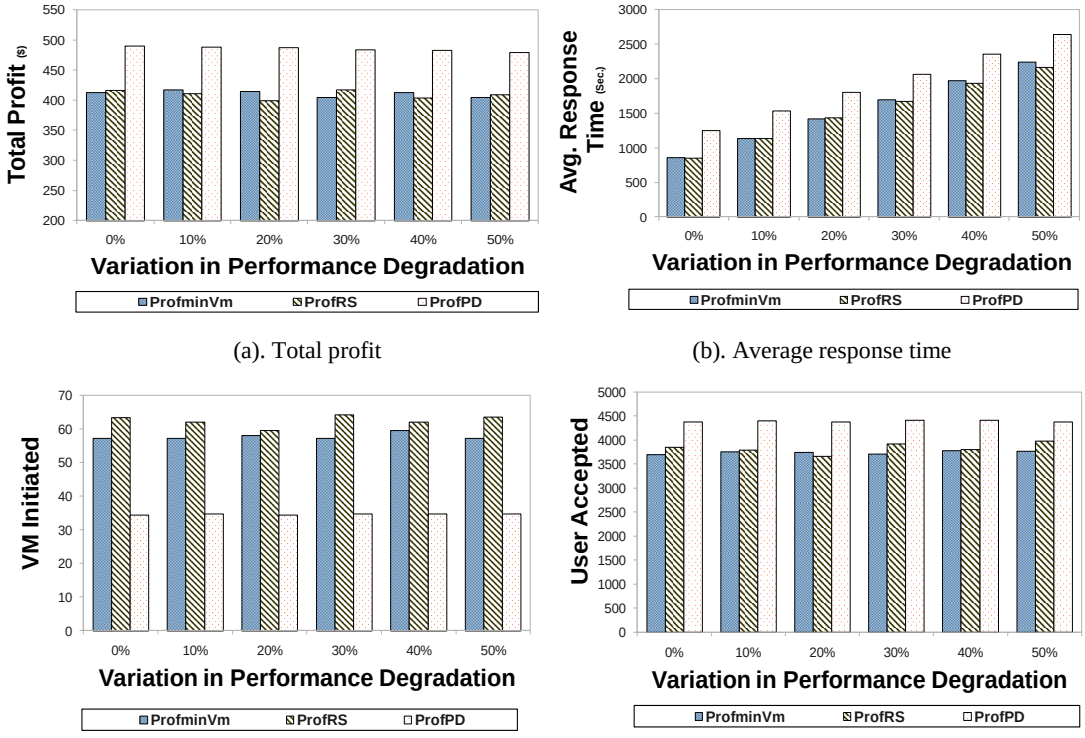
(c). Number of initiated VMs

(d). Number of accepted users

Figure 3.13 Impact of performance degradation variation

Figure 3.13 shows that during the degradation of VM performance, the average total profit (*Figure 3.13a*) has reduced 11% and average response time (*Figure 3.13b*) has doubled with the increase in performance degradation of initiated VMs. This is because of the performance degradation of VMs has not been accounted in SLA(R). Therefore, a SaaS provider does not consider this variation during their scheduling, but it impacts significantly on the total profit and average user requests response time.

Two solutions to handle this VMs performance degradation are: first, utilization of the penalty clause in SLA(R) to compensate for profit loss; second, considering the degradation as a potential risk. Therefore, during the scheduling process a (300 seconds) slack time is added in estimated service processing time and it can be seen from *Figure 3.14*, that the latter solution reduces considerably (from 0% to 50%, profit decreased only by 2%). Thus, if there is a risk for a SaaS provider to enforce SLA violation with an IaaS provider, an alternative solution to reduce risk is by considering a slack time during scheduling.



(a). Total profit

(b). Average response time

(c). Number of initiated VMs

(d). Number of accepted users

Figure 3.14 Impact of performance degradation variation after considering slack time

3.5 Related Work

Research on market driven resource allocation and admission control has started as early as 1981 [72][69]. Most of the market-based resource allocation methods are either non-pricing-based [6] or designed for fixed number of resources, such as FirstPrice [48] and FirstProfit [70]. In Cloud, IaaS providers focusing on maximize profit and many works [89][6][42] proposed market based scheduling approaches. For instance, Amazon [92] introduced spot instance way for customers to buy those unused resources at bargain prices. This is a way of optimizing resource allocation if customers are happy to be terminated at any time. However, our goal is not only to maximize profit but also satisfy the SLA agreed with the customer.

At platform category, Projects such as InterCloud [77], Sky Computing [79], and Reservoir [78] investigated the technological advancement that is required to aid the deployment of cloud services across multiple infrastructure providers. However, research at the SaaS provider level is still in its infancy, because many works do not consider maximizing profit and guaranteeing SLA

with the leasing scenario from multiple IaaS providers, where resources can be dynamically expanded and contracted on demand.

As we focus on developing admission control and scheduling algorithms and strategies for SaaS providers in Cloud, we divide related work into two sub-sections: admission control and scheduling.

3.5.1 Admission Control

Yeo and Buyya presented algorithms to handle penalties in order to enhance the utility of the cluster based on SLA [65]. Although they have outlined a basic SLA with four parameters in cluster environment, multiple resources and multiple QoS parameters from both user and provider sides are not explored.

Bichler and Setzer proposed an admission control strategy for media on demand services, where the duration of service is fixed [74]. Our approach allows a SaaS provider to specify its expected profit ratio according to the cost, for example; the SaaS provider can specify that the service request which can increase the profit in 3 times will be accepted.

Islam et al. investigated policies for admission control that consider jobs with deadline constraints and response time guarantees [90][91]. The main difference is that they consider parallel jobs submitted to a single site, whereas we utilize multiple VM from multiple IaaS providers to serve multiple requests.

Jaideep and Varma proposed learning-based admission control in Cloud computing environments [67]. Their work focuses on the accuracy of admission control but does not consider software service providers' profit.

Reig G. et al contributed on minimizing the resource consumption by requests and executing them before their deadline with a prediction system [86]. Both the works use deadline constraint to reject some requests for more efficient scheduling. However, we also consider the profit constraint to avoid wastage of resources on low profit requests.

3.5.2 Scheduling

Chun et al. built a prototype cluster of time-sharing CPU usage to serve user requests [75]. A market-based approach to solve traffic spikes for hosting Internet applications on Cluster was studied by Coleman et al. [76][75]. Lee et al. investigated a profit-driven service request scheduling for workflows [42]. These related works focus on scenarios with fixed resources, while we focus on scenarios with variable resources.

Liu et al. analysed the problem of maximizing profit in e-commerce environment using web service technologies, where the basic distributed system is Cluster [83]. Kumar et al. investigated two heuristics, HRED and HRED-T, to minimize business value but they studied only the minimization of cost [99]. Garg et al. also proposed time and cost based resource allocation in Grids on multiple resources for parallel applications [89]. However, our current study uses different QoS parameters, (e.g. penalty rate). In addition, our current study focuses on Clouds, where the unit of resource is mostly VM, which may consist of multiple processors.

Menasce et al. proposed a priority schema for requests scheduling based on user status. The algorithm assigns higher priority to requests with shopping status during scheduling to improve the revenue [84]. Nevertheless, their work is not SLA-based and response time is the only concern.

Xiong et al. focused on SLA-based resource allocation in Cluster computing systems, where QoS metrics considered are response time, Cluster utilization, packet loss rate and Cluster availability [87]. We consider different QoS parameters (i.e., budget, deadline, and penalty rate), admission control and resource allocation, and multiple IaaS providers. Netto et al. considered deadline as their only QoS parameter for bag-of-task applications in utility computing systems considering multiple providers [88]. Popovici et al. mainly focused on QoS parameters on resource provider's side such as price and offered load [70]. However, our work differs on QoS parameters from both users' and SaaS providers' point of view, such as budget, deadline, and penalty rate.

In summary, this chapter is unique in the following aspects:

- The utility function is time-varying by considering dynamic VM deploying time (aka initiation time), processing time and data transfer time.

- Our strategies adapt to dynamic resource pools and consistently evaluate the profit of adding a new instance or removing instances, while most previous work deal with fixed size resource pools.

3.6 Summary

We presented admission control and scheduling algorithms for efficient resource management to maximize profit and market share by accepting more profitable user requests with minimum number of resources for SaaS providers. Through simulation, we showed that the algorithms work well in a number of scenarios. Simulation results show that in average the *ProfPD* algorithm gives the maximum profit (in average save about 40% VM cost) among all proposed algorithms in all scenarios varying all types of QoS parameters. If a user request needs fast response time, *ProfRS* and *ProfminVM* could be chosen depending on the scenario. The summary of algorithms and their ability to deal with different scenarios is shown in *Table 3.2*.

In this work, we assumed that the estimated service time is accurate since existing performance estimation techniques (e.g. analytical modelling **Error! Reference source not found.**, empirical, and historical data [83]) can be used to predict service times on various types of VMs. However, still some error can exist in this estimated service time [98] due to variable VMs' performance in Cloud. The impact of error could be minimized by two strategies: first, considering the penalty compensation clause in SLAs with IaaS provider and enforce SLA violation; second, adding some slack time during scheduling for preventing risk.

The next chapter generalizes the problem and presents customer requirements-driven algorithms to achieve SaaS providers' objectives by dedicating personalized attention to customers. These algorithms take into account customer profiles (such as their credit level) and multiple Key Performance Indicator (KPI) criteria.

Table 3.2 Summary of heuristics of comparison results (Profit)

Algorithm	**Time Complexity**	**Overall Performance**						
		Arrival Rate	Deadline	Budget	Request Length	Penalty Rate Factor	VM Initiation Time	Data Transfer
ProfminVM	*O(KIJ+KI)*	Good (low -high)	Good (low-high)	Good	Good (very low & very high)	No effect	Okay	Good (very low & very high)
ProfRS	$O(KIJ+IK^2)$	Okay (very high)	Okay (very high)	Okay (very low)	Okay	No effect	Good (low-high)	Okay
ProfPD	$O(KIJ+IK^2)$	Best	Best	Best	Best	Best	Best	Best

4 SLA-based Resource Provisioning for SaaS Applications

This chapter proposes customers' requirements-driven resource provisioning algorithms to achieve SaaS providers' objectives. The proposed provisioning algorithms consider customer profiles and providers' quality parameters (e.g. response time) to handle dynamic changes in customer requirements and infrastructure level heterogeneity for SaaS providers that lease enterprise software. We also take into account customer-side parameters (such as the proportion of upgrade requests), and infrastructure-level parameters (such as the service initiation time) to compare algorithms. Simulation results show that our algorithms reduce the total cost up to 54% and the number of SLA violations up to 45%, compared with the previously proposed best algorithm.

4.1 Introduction

Research related to SLA-based cost minimization and Customer Satisfaction Level (CSL) maximization for SaaS providers are still in their preliminary stages, and current research on Cloud computing [42][6][89] focus mostly on market oriented models for IaaS providers. Many authors do not consider customer driven resource management, where resources have to be dynamically reallocated according to the customer's on-demand requirements.

CSL can be reduced by SLA violations while it also can be improved by delivering services better than expected. For example, if actual service response time is higher than the one specified in SLA, it causes SLA violations and customer will be unsatisfied. On the other hand, if the response time is smaller than the one specified in the SLA, the customer satisfaction level will be improved.

This chapter proposes customer driven algorithms to minimize the total cost and maximize CSL by resource provisioning. These algorithms also take into account customer profiles (such as their

credit level) and multiple Key Performance Indicator (KPI) criteria. A holistic way to quantify the customer experience is by considering KPIs from seven categories: Financial, Agility, Assurance, Accountability, Security and Privacy, Usability and Performance [115]. To improve a SaaS application's performance quality rating, we consider three KPIs, including one from provider's perspective: cost (part of the Financial category) and two from customers' perspective: service response time (part of the Performance category) and SLA violations (related to Assurance):

- Cost: the total cost of resource usage including VM and penalty cost.
- Service response time: how long it takes for users to receive a response.
- SLA violations: the possibility of SLA violations creates a risk for SaaS providers. In this chapter, SLA violations are caused by elapse in the expected response time, and whenever a SLA violation occurs, a penalty is charged.

To satisfy customer requests in order to minimize the total cost and SLA violations for SaaS providers, the following key questions are addressed:

- How to manage dynamic customer demands? (such as upgrading from a standard product edition to an advanced product edition or adding more accounts)
- How to reserve resources by considering the customer profiles and multiple KPI criteria?
- How to map customer requirements to infrastructure level parameters?
- How to deal with infrastructure level heterogeneity (such as different VM types and service initiation time)?

The key contributions of this chapter are:

- Design of a resource provisioning model for SaaS Clouds considering customer profiles and multiple KPI criteria. These considerations are important for resource reservation strategies to improve the CSL.
- Development of innovative scheduling algorithms to minimize the total cost and number of SLA violations.
- Extensive evaluation of the proposed algorithms with new QoS parameters such as credit levels.

4.2 System Model

The SaaS model for serving customers in the Cloud is shown in *Figure 4.1*. The SaaS provider uses a three layered Cloud model, namely the application layer, the platform layer and the infrastructure layer, to satisfy the user requests. The *application layer* manages all the secured application services, such as the Customer Relationship Management (CRM) or Enterprise Relationship Package (ERP) applications, that are offered to customers by the SaaS provider. The *platform layer* is responsible for application development and deployment (such as Aneka [106], Google App Engine [135], Spring framework). In our model, the function of this layer also includes mapping and scheduling policies for translating the customer side QoS requirements to infrastructure level parameters. The mapping policy considers customer profiles and KPI criteria to measure the SaaS provider's QoS.

The *infrastructure layer* includes the virtualization VM management services (such as VMWare [137], Hyper-V [136]) and controls the actual initiation and termination of VMs resources, which can be leased from IaaS providers, such as Amazon EC2, S3 [106] or own private virtualized clusters. In both cases, the minimization of the number of VMs will deliver savings for the providers.

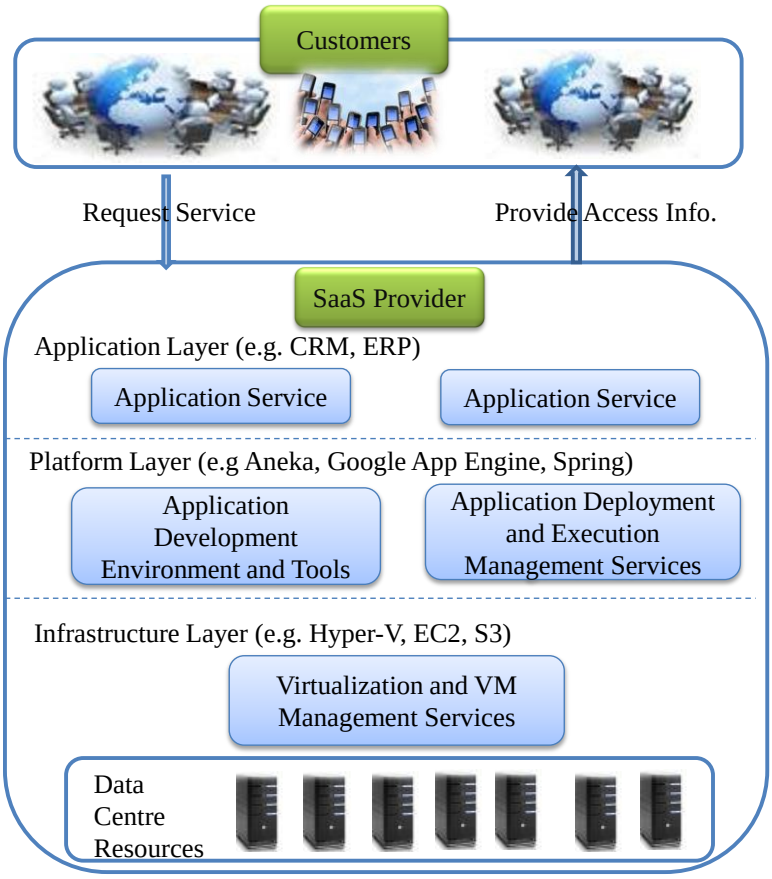


Figure 4.1 A system model of SaaS layer structure

4.2.1 Actors

The actors involved in our system model are described below along with their objectives, activities and constraints.

SaaS Providers

SaaS providers lease web-based enterprise software as services to customers. The main objective of SaaS providers is to minimize cost and SLA violations. We achieve this objective by proposing customer-driven SLA-based resource provisioning algorithms for Web-based enterprise applications. In our context, a SaaS service provider *X* offers CRM or ERP software packages with three product editions (for example, Standard, Professional and Enterprise) and each product edition with a fixed price. The current SaaS providers, such as 'Compiere ERP', use a similar service model [107]. In this service model, when a customer *Company Y* submits its 'first time rent' request with a product edition (*Standard*), and additional number of accounts, the SaaS provider needs to allocate resources and then provides the login information to the customer. *Company Y* may require an upgrade in their service by adding

additional user accounts or an upgrade of the software edition. In this case, sometimes a new VM is created and the content from the previous VM is migrated to the new one. In practice, the provider has to handle these on-demand customer requests in line with the SLA. The SLA properties including the provider's pre-defined parameters and the customer specified QoS parameters are as follows:

- Product Edition (*p*): It is defined as the software product package that is offered to customers. For example, *SaaS X* offers Standard, Professional, and Enterprise product editions.
- Request Type *(j)*: This defines the type of customer request, which may be a 'first time rent' or a 'service upgrade' request. 'First time rent' means the customer is renting a new service from this SaaS provider. A 'service upgrade' includes two types of upgrade, which are 'add account' and 'upgrade product'. To downgrade a service, first, the customer needs to terminate the current contract, and then processing of this downgrade request will be treated as a new request.
- Contract Length (*cl*): How long the customer is going to use the software service.
- Number of Accounts *(a*): The actual number of user accounts that a customer wants to create. The maximum number of accounts is related to and restricted by the type of product edition.
- Number of Records (*n*): The average number of records that a customer is able to create for each account during a transaction and this may impact the data transfer time during the service upgrade (The value of this parameter is predefined in the SLA).
- Response Time (*respT*): It represents the time taken by the provider to process a particular customer request. For example, An SLA violation occurs when the actual response time is longer than it was defined in the SLA. We consider four types of response time: (a) first time renting (*ftr*) of the service - *respT(ftr),* (b) upgrading the service(*upServ*) by adding additional accounts (*addAcc*) - *respT(upServ,addAcc)* (c) upgrading the product (*upProd*) - *respT(upServ,upProd),* and (d) the service usage (*useServ*), such as for saving a document (the value of each type of response time is different and predefined in the SLA).
- Penalty Conditions: For each SLA violation the SaaS provider needs to pay a penalty, which is based on the delay in the response time to the customer. For each request type there is a different penalty (detailed in the cost model on Section 2.2.2). Penalty rate is the monetary cost incurred to the provider for unit time delay in serving the customer request.

The *infrastructure layer* (*Figure 4.1*) uses VM images to create instances on their physical infrastructure according to mapping decisions. The following infrastructure layer properties are important for mapping:

- VM types (*l*): The type of VM image that can be initiated. For instance, there may be three types of VMs: large, medium, and small. The three types of VMs have different capability to serve different numbers of accounts and records since different requests may consume different memory and storage. Therefore, for a particular type of VM, price, and the maximum capabilities are listed in Table 4.1.
- Service Initiation Time (*iniT*): How long it takes to initialize the service, which includes the VM initiation time and application deployment and installation time.
- Service Processing Time (*procT*): It is defined as the time taken to process an operation of SaaS service. For example, how long it takes to generate a report, or save a transaction record.
- VM Price (*VMPrice*): How much it costs for the SaaS provider to use a VM for the customer request per hour. It includes the physical equipment, power, network and administration cost.
- Data Transfer Time (*DTT*): How long it takes to transfer one Gb record from one VM to another. This depends on the network bandwidth.

Customers

When customers register on the SaaS provider's portal, their profile information is gathered. In practice, this happens via forms that customers fill during the registration process. To categorize customers, high level information such as company size in range is collected. For example, when the number of information workers, who may be the potential users, are between 5 to 10. The following items are considered:

- Company Name (*compName*): The legally registered trade name.
- Company Size (*compSize*): The number of information workers (staffs who may use the software service) in the company.
- Company Type (*compType*): The classification of a customer's company based on the number of employees and revenue. Customer companies are categorized into three divisions, i.e., small, medium, and large.
- Future Interest Expression (*futureInterest*): The customer's expected future upgrade requirements. Such as the need for additional user accounts. This allows the SaaS

provider to plan for possible offering of discount as it helps them in making resource reservation decisions. The provider's reduced cost due to advance booking is shared with customer by offering them a discounted price. Such practice is quite commonly used by current industries and service providers. Therefore, we believe that this model will work well for Cloud computing.

Moreover, in the service market, there are two types of sales models, which are one off and long term relationships. The entire sales process is based on relationship building and trust [129]. In addition, the application type we provide is enterprise application, which is used as a pay-as-you-go and most of time with the customer repeatedly using the service. For instance, Company Y may need to use the invoice and report services only a few times a month, but they will use these services repeatedly over the long term. Therefore, we focus on the relationship model but not the once off model (e.g. spot pricing).

4.2.2 Mathematical Models

Customer Profile Model

Credit Level (*creditLevel*): It measures the creditability and loyalty of a customer, which depends on the value of the company type and credit level factor (Equation 4.1).

$$\text{creditLevel} = \text{compTypeValue} \times \sigma \tag{4.1}$$

The *CompTypeValue* indicates the company type, which is categorized based on the range of company size. In practice, the company size can be verified during the registration identity and security verification process. The *CompTypeValue* for small, medium and large company types are 1, 2 and 3 respectively. The reason we use the values 1, 2 and 3 rather than say 10, 20, 30 or other sets of values, because the trend of other value sets are found to be the same during the evaluation. The company type is considered when calculating the credit level, because having larger companies as customers adds more value to the SaaS provider's market share. The credit level factor (σ) is determined by the customer's historical upgrade requests and the actual upgrade action. The actual upgrade is a boolean value. If an actual upgrade happened, the actual upgrade is true, and otherwise it is false. The value of actual upgrade (*actualUpgradeValue*) is the actual value, such as number of account, that service upgrades requested. The credit level factor (σ) is the ratio of the *actualUpgradeValue* and *futureInterestValue* (which cannot be 0) (Equation 4.2).

$$\sigma = \frac{actualUpgradeValue}{futureInterestValue} \quad (4.2)$$

For example, Company Y expresses a future interest to add 2 user accounts before the contract expiry date. In this case the future interest is 'add user accounts' and the value of the future interest (*futurInterestValue*) is 2. If they do not come back to request more user accounts (the actual upgrade is false, and the *actualUpgradeValue* is 0), its credit level factor (σ) is 0; but if it adds one user account (the actual upgrade is true, and the *actualUpgradeValue* is 1), the credit level factor (σ) is 1/2 =0.5 (Equation 4.2). If it adds 3 user accounts (the *actualUpgradeValue* is 3), the credit level factor is 3/2 = 1.5. If there is no history about previous actions or user does not specify the future interest value, then σ is 0 (in this case the 'future interest value' is not used for new requests). The customers have to specify the future interest every time they submit requests.

This model is used to adjust the inaccuracy or ensure information from the customer using the actually verified and historical data. However it is necessary for providers to keep gathering future interest data from customers, since customers supplied high level "future" expectations/ requirements guided in the initial planning and helps resource providers to plan about possible incentives they may offer to their "high" value customers.

Cost Model

Let *C* be the number of customer requests and *c* indicates a customer request id. At a given time *t*, a customer submits a service request *c* to the SaaS provider. The customer specifies a product edition, contract length, and number of accounts after agreeing with the pre-defined SLA clauses (response time). After the SLA establishment, the SaaS provider will reserve the requested software services which are translated at the infrastructure level to match the VM capacity.

Let *Cost* be the total cost incurred to the SaaS provider to serve all customer requests *C* and as described in Equation (4.3). It depends on the VM cost and the penalty cost.

$$Cost = VMCost + PenaltyCost \quad (4.3)$$

Let *I* be the number of initiated VMs, and *i* indicates the VM id. The VM cost is the total cost for all VMs and is expressed by Equation (4.4):

$$VMCost = \sum_{i=1}^{I}(VMCost_i) \quad i \in I \tag{4.4}$$

The Penalty cost is the total penalty cost for all customer requests *C* and is expressed by Equation (4.5):

$$PenatyCost = \sum_{c=1}^{C} PenaltyCost_c \quad c \in C \tag{4.5}$$

For each VM *i*, the VM cost depends on the VM price of type *l (VMPrice$_l$)*, the time slot when the VM is on (s_i), and the time slot when the VM is off (f_i) and the set up time of the VM *i* (ts_i) and it is expressed by Equation (4.6):

$$VMCost_i = VMPrice_l \times (f_i - s_i + ts_i) \qquad i \in I, l \in L \tag{4.6}$$

Let *c'* be the previous request from the same customer. The time spent on a VM set up is expressed by Equation (4.7) and it depends on the request type *j*, VM initiation time *iniT$_i$*, total data transfer time for *c'* (totalDTT$_{c'}$). If *j* is 'first time rent' then the data transfer time is zero. Only when *j* is 'service upgrade' and requires data migration, the data transfer time occurs.

$$ts_i = \text{iniT}_i + \text{totalDTT}_{c'} \quad i \in I, l \in L, c' \in C \tag{4.7}$$

The total data transfer time depends on the number of accounts ($a_{c'}$) that previously were requested by the same customer, the data records created by previous request *c'*, the storage size per record ($rs_{c'}$) and data transfer time per size ($DTT_{c'}$). *N* indicates the total number of records and *n* is the record id.

$$totalDTT_{c'} = a_{c'} \times \sum_{n=1}^{N} rs_{c'} \times DTT_{c'} \quad n \in N, c' \in C \tag{4.8}$$

The SLA violation penalty (*Penalty*) model is similar to the models used in the related publications [65][48][68] and is modeled as a linear function. The penalty model is shown in Equation (4.9). The constant factor α is used to make sure the minimum penalty is always greater than 0. β is the penalty rate and *td* indicates delay time. β is based on the request type, and each type of request incurs the same range of penalty rate. This is a similar model to credit card penalty, in which the late payment for a particular type of card will have the same range of penalty [132].

$$Penalty = \alpha + \beta \times td \tag{4.9}$$

The penalty function penalizes the service provider by increasing the cost. According to the penalty model, the penalty cost equation for each customer request c is depicted as follows where the customer request c is of request type j and td_c indicates the delay time for customer request c.

$$PenaltyCost_c = \alpha + \beta_j \times td_c \quad j \in J, c \in C \tag{4.10}$$

The delay time *td* is the variation between the value of the response time defined in the SLA and the actual experienced response time. There are four situations in which a penalty delay can occur (*Table 4.1*). If the request type is 'first time rent', the delay (violation) can occur due to a long service initiation time. If the request type is 'upgrade service', the delay can be caused by adding accounts or upgrading the product edition. Moreover, during the service usage, the delay can be caused by machine performance degradation, which is out of the scope of this chapter.

Average performance can be calculated based on a per-user (macroaverage) or per-request (microaverage). Macroaverage performance treats all users equally, although some users will be more active and generate more traffic than others. In contrast, microaverage performance emphasizes the requests made by highly active users. Authors claimed that "we don't always build per-user predictive models. Individual models of behavior require more space, and tend to be less accurate because they see less data than a global model. Thus for comparison, we will report only per-request average" [139]. In addition, we consider penalties caused by service preparation response time which are once-off activities without moving average and thus it is based on per-request.

Table 4.1 The summary of penalty delay time according to request types

Response Time	**First Time Rent-ftr**	**Upgrade Service**	
		Add account-addAcc	**Upgrade product-upServ**
Defined in SLA	respT (ftr)	respT(upServ, addAcc)	respT (upServ, upPro)
Actual Time	iniT	iniT + totalDTT	iniT +totalDTT

The service initiation time varies subjected to the physical machine's capability

$$td_c = \begin{cases} iniT_i - respT_j & \text{where, } j = \text{first time rent} \\ iniT_i + totalDTT - respT_j & \text{where, } j = \text{upgrade service} \end{cases} \tag{4.11}$$

4.2.3 Mapping of products to resources

In our work, the infrastructure layer focuses on the VM and the host level. The mapping between a host and hosted VMs is depicted in *Figure 4.2*. Our VM to physical machine 'Mapping configuration' supports heterogeneous physical machines. Homogeneous physical machines are depicted just for easy comparison and presentation of results.

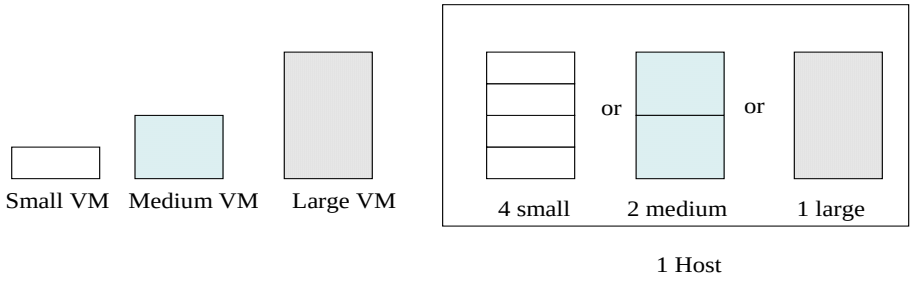


Figure 4.2 Mapping between VMs and a Host

Table 4.2 The summary of mapping between requests and resources

VM Type	VM Capacity and Price	Product Edition	Max Account #	Min Account #
Small	1 CPU Unit, 2Gb RAM, 160 G Disk. $0.12 per hour	Standard	M	1
Medium	2 CPU Unit, 4Gb RAM, 850 G Disk. $0.48 per hour	Standard, Professional	2m	m+1
Large	4 CPU Unit, 8Gb RAM, 1690 G Disk. $0.96 per hour	Standard, Professional, Enterprise	10m	2m+1

We use a similar record model as 'Salesforce.com' to restrict each account to create the maximum number of records. This configuration is chosen to avoid/minimize the SLA violations due to service response delay. Because the VM performance can degrade after a certain number of VMs are hosted on the same server due to using shared resources, such as CPU. An example of a mapping strategy between customer requests and VM resources is shown in *Table 4.2*.

4.2.4 Problem description

Let a SaaS provider have *I* VMs initiated in a data center, and *C* is the number of requests currently arriving to the SaaS provider. The SaaS provider charges a fixed service price from

customers for an application based on their request parameters. The request parameters include request type (j), product edition (p), contract length (cl), and the number of accounts (a). The SaaS provider has a dual objective, i.e., minimizing the cost and improving CSL. The objective functions and constraint functions are explained below with input parameters and variables:

Input Parameters

1. L: Set of VM type.
2. η: Time-slot size.
3. I: Set of VM has been initiated from time 0 to time T. T is divided in slots of size η
4. $VMPrice_l$: The cost of VM of type l , $l \in \{1, 2, 3\}$.
5. c: The particular request. The parameter of this request includes the number of accounts, when the contract starts, when the contract finishes, which type of request it is, and what type of product it is requesting.
6. C: Set of customer requests received from time 0 to T.
7. β_j: Penalty rate that is associated with request type j.
8. A_l: Maximum number of accounts that can be allocated for VM type l.
9. a_c: The number of accounts requested by request c.
10. s_c: The time slot when this customer request contract started.
11. f_c: The time slot when this customer request contract finished.

Variables

12. $y_{il} = 1$, if VM i is of type l, otherwise = 0.
13. z_{cj} : For request c, $z_{cj}=1$, if request c is of request type j.
14. f_i: The time slot when the VM is off.
15. s_i: The time slot when the VM is on.
16. $x_{cit}= 1$, if request c is served by VM i at time slot t.
17. td_c: The time delayed to serve request c.
18. ts_i: The time spent in setting up the VM i.

Objective Functions In our model we are interested in minimizing the total cost and SLA violations. Consequently, cost minimization can be described by the following function:

$$\text{Minimize (Cost)} = \text{VMCost} + \text{PenaltyCost} \tag{4.12}$$

Where

(4.13)

$$VMCost = \sum_{i=0}^{I}\left(\sum_{i=1}^{3} VM\ \mathrm{Pr}ice_l y_a \times (f_i - s_i + ts_i)\right)$$

$$PenaltyCost = \sum_{c=1}^{c}\left(\alpha + \sum_{j=1}^{3}(z_{cj} \times \beta_j)td_c\right) \quad (4.14)$$

In Equation (4.12), the minimization of cost depends on the VM Cost, and Penalty Cost due to SLA violations. In Equation (4.13), the VM Cost depends on the type of VM *l* and the time period VM is on, which is calculated by ($f_i - s_i$). VM Cost is the cost of all initiated VM of type *l* during the time period when the VM is on. In Equation (4.14), the Penalty Cost depends on the, penalty rate β_j of request type *j* and time delayed to serve request c (td_c).

The other objective function is to maximize of the CSL by minimizing the SLA violations, which is expressed below:

Minimize (SLA violations) (4.15)

The number of SLA violations impacts CSL, so we consider minimizing the number of SLA violations as the objective function for maximizing CSL.

Constraints: The SaaS provider needs to ensure that the customer requested product edition, and the number of accounts are allocated before a threshold time (refer to *Table 4.1*) to minimize the penalty delay. To this end, we define the following set of constraint functions:

$$\sum_{c=0}^{C} x_{cit} a_c \le \sum_{l=1}^{3} A_l y_{tl} \quad (4.16)$$

$$s_i = \min_{\forall c}\{x_{cit} s_c\} \quad (4.17)$$

$$= f_i = \max_{\forall c}\{x_{cit} f_c\} \quad (4.18)$$

$$0 \le a_c \le \sum_{l=0}^{I}(\sum_{l=1}^{3} x_{cit} y_{il} A_l) \quad (4.19)$$

The Equation (4.16) restricts the number of accounts requested by all customers on VM *i* which should be within the maximum capability of the VM of type *l* (the VM capability is listed in

Table 4.2). In Equation (4.17), s_i represents the minimum time when customer contract started. In Equation (4.18), f_i represents the max time when customer contract finished. In Equation (4.19), the number of accounts (a_c) should be less than or equal to the maximum capability of the VM of type *l*, which is serving the customer request c.

The objective functions (4.12) and (4.15) of the SLA based resource provision problem are to minimize cost and SLA violations for a SaaS provider. The constraints ensure that the customer requirements of an application are met. However, it is difficult to allocate the exact number of accounts to a VM to avoid space wastage within the response time, because customer requests have different parameters, require different types of VMs, and have dynamic arrival rates [133]. Moreover, this problem maps to the 2-dimensional bin-packing problem which is NP-hard [134] (proof as below), hence we propose various algorithms to heuristically approximate the optimum.

Proof of 2-dimensional bin-packing problem

Definition 1. Let X=(x1, xij, …., xn) be a given list of n items with a value of xij , and B= b1, … bm be a finite sequence of m bins each of unit capacity. The 2-dimensional bin-packing problem is to assign each xij into a unique bin, with the sum numbers in each bj not exceeding one, such that the total number of used bins is a minimum (denoted by L*) [43].

Proposition 1.The optimization problem described in Equation (13) and (14) is an NP-hard problem

Proof. The proposition can be proven by reducing the problem to the (2-dimensional) bin-packing problem [43], which is a well-known NP-hard problem. The number of bins m is equal to the available N VMs. The dimensions of an application request c consist of two parameters: the number of accounts (ac) and the contract length (fc-sc). However, to serve the request on a particular VM depends on these two parameters with objective that total number of VMs is minimum. By Definition 1, it is a 2-dimensional bin-packing problem defined by the number of accounts and contract length.

4.3 Resource Provisioning Algorithms

As discussed on the provider side, the main objective of our work is to minimize cost and SLA violations using resource provisioning strategies to achieve SaaS providers' objectives. We use the best algorithm (*ProfminVMMinAvaiSpace*) proposed in our previous chapter [114] as a benchmark algorithm (renamed to *BestFit*) and propose two new algorithms: *BFResvResource* and *BFReschedReq*, which consider customer profiles and provider KPI criteria.

4.3.1 Base Algorithm: Maximizing the profit by minimizing the cost by sharing the minimim available space VMs (BestFit).

A SaaS provider can maximize its profit by minimizing the resource cost, which depends on the number and type of initiated VMs. Therefore, this algorithm is designed to minimize the number of VMs by utilizing the same already initiated one for serving other user requests as well. The algorithm avoids SLA violations of existing requests by not allocating new request to the initiated VM if the new request can cause an SLA violation to existing customers.

The strategy of this algorithm is illustrated in *Figure 4.3*, where the gray space indicates unavailable space, *x* axis indicates the id of VM, which has the same VM type and is deployed with the same type of product as customer *c* requested; y axis indicates the number of accounts a VM can hold.

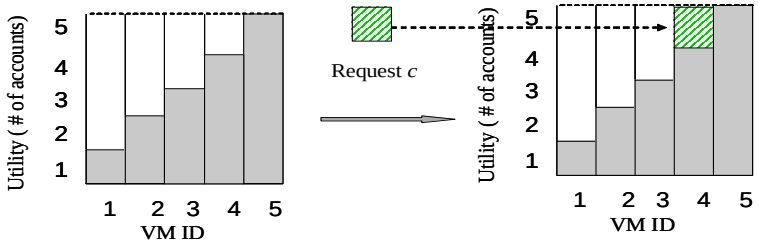


Figure 4.3 Best Fit Strategy

Base Algorithm Pseudo-code for *BestFit*	
Input	request *c* with QoS parameters
Output	Boolean
Functions	FirstTimeRent (), Upgrade ()

```
First Time Rent (c)
1   Let p be the product edition and a_c be the number of accounts required by request c
2   Let L be type of VM which can serve c after applying mapping strategy.
3   Foreach VM i of type 'l' from 'L' to 'Large'
    {   //get list of VMs of type l which can serve the request 'c'
4       Let vmList=GetVMlist(l, p , a_c)
5       If(vmList is empty)
6           continue;
7       Else
        {
8           Allocate capacity of VM_min with minimum available space in vmList to request c
9           update the available capacity of VM_min to (VM_min's available capacity – a_c)
```

```
10                  break;
            }
    }
11  If(request c is still not served)
    {
12      Initiate a new VM of type L and deploy the product type p on the VM
13      Allocate capacity of the new VM to request c
14      Update the available capacity of the new VM to (available capacity – a_c)
    }
Upgrade(c)
1   If (upgrade type is 'add account')
    {
2       Get VM_il which is processing the previous request from the same customer as c
3       If ( VM_il has enough space to serve request c and can guarantee SLA objectives of existing requests)
        {
4           Process request c using VM_il
        }
5       Else
        {
6           Let a_c' be the number of account that are already rented by the customer.
7           Let new a_c be the number of more accounts requested by the customer
8           Using similar process as of the function First Time Rent(c) search a newVM_il which can serve request
            with
            ( a_c'+new a_c) accounts
9           Transfer data from VM_il to newVM_il
10          Release the space in old VM_il
        }
11  If (upgrade type is'upgrade service')
    {
12      Get the VM_il which processed the previous request from the same customer as c
13      Using similar process as of the function First Time Rent (c) search a newVM_il which can serve the request
14      Transfer data from VM_il to newVM_il
15      Release the space in old VM_il
    }
```

Customer request *c* is the input of the algorithm, which includes the request type, product edition, and the number of accounts. The algorithm involves two main request types: a) first time rent and b) upgrade service.

If the request type is 'first time rent', the algorithm gets the VM type *L* using a mapping table similar to *Table 4.2* (Line 1). Then, it checks and gets the list of all initiated VMs of type *L* (Line 2) that can serve the request 'c' (Line 4). If there is no such initiated VM, it will find space in other types of VMs which are larger in size (Line 5-6). Otherwise, the request *c* is assigned to the VM from 'vmList' that has minimum available space (Line 8). The available capacity of VM_{min} is updated (Line 9-10) (it is illustrated in *Figure 4.3*). If there is no initiated VM, which can serve the request, then it initiates a new VM according to the mapping strategy and deploys the requested product on this VM (Line 13).

If the request type is 'upgrade', then it checks the type of upgrade. If upgrade type is 'add account', the algorithm gets the id (*i*) and type (*l*) of VM, which has placed the previous request from the same customer as *c'* (Line 2). If VM_{il} has enough space to place the new request *c*, the algorithm schedules *c* to VM_{il} (Line 3, 4). Otherwise, the algorithm searches for a newVM_{il} using a similar way as given in First Time Rent (Line 6-8).Then, the algorithm transfers data stored on the old VM to the new VM and releases space on the old VM (Line 9, 10). On the other hand, if a customer requests an upgrade to a more advanced product edition, the new request is placed to a suitable VM by using the First Time Rent() function, and then the customer's old data is migrated to the new VM and the space occupied by the old request on the old VM is released (Lines 11-15). The time complexity of this algorithm is *O(IK+I)*, where *I* represents the total number of initiated VMs and *K* represents the total number of existing requests.

The "*BestFit*" algorithm minimizes the number of initiated VMs in order to minimize cost. However, the disadvantage is that it can increase the cost in some cases due to delay penalties. For example, when a new customer requests to add more accounts on the VM which has been fully occupied by other requests, initiating a new VM may be more expensive than the delay penalty.

4.3.2 Proposed Algorithms

- Minimizing the cost by minimizing the penalty cost through resource provisioning based on the customer's credit level (*BFResvResource*).
- Minimizing the cost by rescheduling the existing requests (*BFReschedReq).*

Algorithm 1 : Minimizing the cost by minimizing the penalty cost through resource provisioning based on the customer's credit level (BFResvResource)

The base algorithm can cause upgrade penalties in the situations when a customer requests to add more accounts and the available space is filled by other requests, because this could

trigger the initialization of a new VM. To optimize the cost caused by adding new accounts, *Algorithm 1* provisions more resources than requested based on the customer's credit level (which is driven by customer's actual requirements, the credit level is 0 when the request type is new). When a request's credit level is greater than the provider's expected value, more resources will be provisioned in order to minimize the time spent on adding user accounts. The algorithm is designed to minimize penalty cost due to the addition of new accounts to the system by reserving resources according to the customer requirements (Line 11). Penalty cost is caused by SLA violations; therefore the reduction of penalty cost will automatically reduce SLA violations. The algorithm also reserves resources according to the historical record and customer estimation to reduce VM cost. Therefore, the total cost (based on VM cost and penalty cost) are minimized.

The customers may be unsure about their future interest, so we design two types of reservation strategies (dynamic and fixed) to figure out how much resources should be reserved. Dynamic reservation (*dynamicR*) strategy reserves resources for customer request *c* depending on its credit level ($creditLevel_c$), the number of accounts (a_c(futureInterest)) specified in the future interest and provider's expected value for credit level (its value is '1' in the experiments) using Equation (4.20). Fixed reservation strategy uses a fixed percentage (e.g. 20%) customer specified future interest value instead of credit level.

$$dynamicR = \begin{cases} creditLevel_c \times a_c(futureInterest), & \text{if } creditLevel_c \geq \text{provider expected value} \\ 0, & \text{otherwise} \end{cases} \quad (4.20)$$

The *ReservationStrategy* is depicted in *Figure 4.4* (The pattern with horizontal line indicates the reserved resources for the same customer; gray space, *x* axis and *y* axis are the same as *Figure 4.3*). The other lines are the same as those in the base algorithm. The time complexity of this algorithm is *O(IK+I)*, where *I* represents the total number of initiated VMs and *K* represents the total number of existing requests.

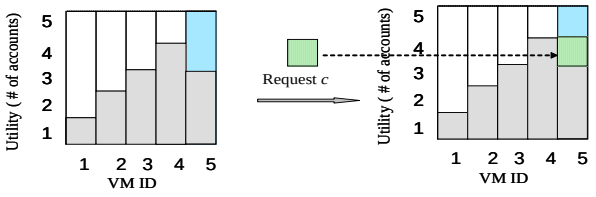


Figure 4.4 The Reservation Strategy

Algorithm 1. Pseudo-code for *BFResvResource*

Input request *c* with QoS parameters

Output Boolean

Functions: FirstTimeRent (), Upgrade ()

```
First Time Rent (c)
1   Let p be the product type and a_c be the number of accounts required by request c
2   Let L be type of VM which can serve c after applying mapping strategy.
3   Foreach VM i of type 'l' from 'L' to 'Large'
    {
4       Let vmList=GetVMlist(l, p , a_c)//get list of VMs of type l which can serve request 'c'
5       If (vmList is empty)
6           continue;
7       Else
        {
8           Allocate capacity of VM_min with minimum available space in vmList to request 'c'
9           CreditLevel = getCreditLevel(Profile Information)
            //get the credit level for request 'c'
10          If (CreditLevel ≥ Threshold)
11              update the available capacity of VM_min to (VM_min's available capacity – a_c(futureInterest))
12          Else
13              update the available capacity of VM_min to (VM_min's available capacity – a_c)
14          break;
        }
    }
15  If (request c is still not served)
    {
16      Initiate a new VM of type L and deploy the product type p on the VM
17      Allocate capacity of the new VM to request c
18      update the available capacity of the new VM to (available capacity – a_c)
    }
Upgrade(c)
1   If (upgrade type is 'add account')
    {
2       Get VM_il which is processing the previous request from the same customer c
3       If ( VM_il has enough space to serve request c and can guarantee SLA objectives of existing requests)
        {
4           Process request c using VM_il
        }
5       Else
        {
6           Let a_c' be the number of account that are already rented by the customer.
```

7			Let *new a_c be the number of more accounts requested by the customer*
8			Using similar process as of the function First Time Rent (*c*) search a newVM$_{il}$ which can serve request with *(a_c+new a_c) accounts*
9			Transfer data from VM$_{il}$ to newVM$_{il}$
10			Release the space in old VM$_{il}$
		}	
	}		
11	If (upgrade type is 'upgrade service')		
	{		
12			get the VM$_{il}$ which processed the previous request from the same customer *c*
13			Using similar process as of the function First Time Rent (*c*) search a newVM$_{il}$ which can serve the request
14			Transfer data from VM$_{il}$ to newVM$_{il}$
15			Release the space in old VM$_{il}$
	}		

Algorithm 2: Minimizing the cost by rescheduling existing requests. (BFReschedReq).

Algorithm 1 prevents the penalties caused by adding accounts but does not prevent penalties caused by upgrading the product edition. Algorithm 2 further minimizes the product edition upgrade penalty by rescheduling accepted requests, which leads to a reduction of SLA violations and total cost (Line 11-26).

The strategy of this algorithm is depicted in *Figure 4.5* (The pattern with horizontal line indicates the reserved resources for the same customer; gray space, *x* axis and *y* axis are the same as *Figure 4.3*). The time complexity of this algorithm is $O(IK+I^2)$ where *I* represents the total number of initiated VMs and *K* represents the total number of existing requests.

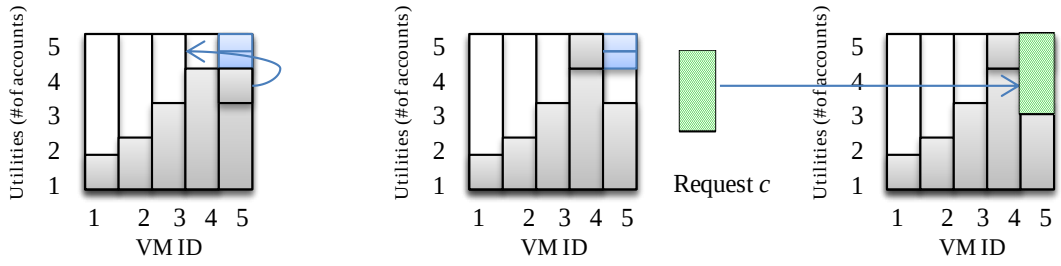


Figure 4.5 The Reschedule Strategy

This algorithm is designed in a way that all VMs are deployed with the full software package to reduce the resource discovery and content migration time for rescheduling accepted requests. If the request type of *c* is 'service upgrade', the algorithm checks the available

space of VM_i which has served the previous request *c'*. If the available space of VM_i is less than the *c* required and there is an existing request c_e, which causes a lower (or zero) penalty than the current request *c*, then request *c* is scheduled on VM_i and the c_e is migrated to another available and capable VM (Upgrade (*c*)). The request c_e is rescheduled to the cheapest VM. The rest of the lines are the same as those in *Algorithm 1* except that *Algorithm 2* does not differentiate VM types, because all VMs are deployed with the full package. When the customer requests more accounts than the reserved fixed percentage for upgrade, the upgrade function will take care of the exception (Lines 13-26). Briefly, the algorithm checks if the current VM has enough available resources to fit the extra accounts. If yes, the extra accounts will be allocated to the same VM. If no, we will search for the same type of VM with minimum available but enough capability. If there is no suitable VM, *Algorithm2* need to check if a new VM can be initiated. This may require content migration and incurs penalty cost.

Algorithm 2. Pseudo-code for *BFReschedReq*

Input	request *c* with QoS parameters
Output	Boolean
Functions:	FirstTimeRent (), Upgrade ()

```
First Time Rent (c)
1     Let p be the product type and a_c be the number of accounts required by request 'c'
4     Let vmList=GetVMlist(p , a_c)//get list of VMs of which can serve request 'c'
5     If (vmList is not empty){
8       Allocate capacity of VM_min with minimum available space in vmList to request 'c'
9       CreditLevel = getCreditLevel(Profile Information)
        //get the credit level for request 'c'
10      If (CreditLevel ≥ Threshold )
11            update the available capacity of VM_min to (VM_min 's available capacity – a_c(futureInterest))
12      Else
13            update the available capacity of VM_min to (VM_min 's available capacity – a_c)
        }
14    Else
        {
15      Initiate a new VM of type L and deploy the product type p on the VM
16      Allocate capacity of the new VM to request c
17      Update the available capacity of the new VM to (available capacity – a_c)
        }
Upgrade(c){
1     If (upgrade type is 'add account')
        {
```

```
2       Get VM_il which is processing the previous request from the same customer as c
3       If ( VM_il has enough space to serve request c and can guarantee SLA objectives of existing requests)
        {
4           Process request c using VM_il
        }
5       Else
        {
6           Let a_c' be the number of account that are already rented by the customer.
7           Let new a_c be the number of more accounts requested by the customer
8           Using similar process as of the function First Time Rent (c) search a newVM_il which can serve request
            with ( a_c'+newa_c) accounts
9           Transfer data from VM_il to newVM_il
10          Release the space in old VM_il
        }
    }
11  If (upgrade type is 'upgrade service')
    {
12      get the VM_il which processed the previous request from the same customer as c
13      If ( the available space of VM_il is less than request c required in VM_il ) {

15          If ( migrating c'generates minimum penalty cost || after trying to migrate all requests,
16      available space in VM_il is still less than request c required) {
17              Find or initiate the VM where new and previous requests generate minimum penalty
18      cost
19              Migrate c' and assign c to the VM found or initiated in last step.
                Transfer all the data to this VM.
            }
20          Else {
21              Find or initiate the VM where migrating other requests generate minimum penalty cost
22              Migrate these requests to the VMs found or initiated in last step.
23              Transfer all the data to this VM.
            }
24          Release the space in old VM_il
        }
25      Else {
26          Allocate c to VM_il;
        }
    }
```

4.3.3 Lower Bound

Due to the NP hardness of the SLA-based resource provisioning problem described in the system model section, it is difficult to find the optimal solution in polynomial time. Thus, to estimate the performance of our algorithms, we present a lower bound for the cost. The lower bound is derived from the scenario when we can get the minimum cost in case all requests are allocated to the VM to minimize the VM space wastage, penalty cost and number of SLA violations.

The constraint of the request and VM mapping relationship depends on the number of accounts, product edition, and request type. For the sole purpose of deriving the lower bound, we relax these constraints to minimize the VM space wastage and penalty cost by initiating the large VM to deploy and install the full package (e.g. enterprise edition) on them. Take the product edition as an example, when the type of the old VM is small, but the customer requests to upgrade product edition to enterprise, which requires the VM of type large but the existing large VMs may do not have enough space for the new request, which causes the penalty. Because all VMs have the same capability, when one VM does not have enough space, we can allocate some accounts to other VMs to minimize VM space wastage. In addition, to relax the dynamic request constraint, the incoming customer requests are known in advance. This forms the ideal lower bound scenario, where all incoming requests are known in advance without any request constraint. c denotes the individual customer request and C denotes the total number of customer requests arrived at time t. a_c denote the number of accounts requested by customer request c. The maximum number of accounts can be accepted by the large VM is defined as M. According to Equation 4.3, the equation for lower bound is expressed by:

$$\text{Minimize (Cost)} = \text{VMCost} + \text{PenaltyCost} \quad ; \quad \text{Where} \quad \text{PenaltyCost} = 0 \tag{4.21}$$

$$\text{PerUnitTimeVMCost}_{\text{large}} = \text{VMPrice}_{\text{large}} \times \text{Min}(\text{VM}_{\text{large}}) = \frac{\sum_{c=1}^{C} a_c}{10 \times (VM_{small})} \tag{4.22}$$

$$\text{VMPrice}_{\text{large}} \times \frac{\sum_{c=1}^{C} a_c}{M\,(VM_{large})}$$

However, this lower bound solution is the ideal solution, whereas in real dynamic and constraint Cloud environment we cannot achieve the lower bound but can optimize proposed algorithms to be as close as possible to the lower bound. The reason for initiating the large VM to minimize the total cost is proved as below.

Proof for using large VM in lower bound

For VM of type *l*, the base number of accounts can be accepted is *m*, the VM capability of type *l* in regard to the maximum number of accounts can be accepted is expressed by:

$$M(VM_l) = \alpha \times m \begin{cases} \alpha = 1, when\ l = small \\ \alpha = 2, when\ l = medium \\ \alpha = 10, when\ l = large \end{cases} \quad (4.23)$$

Let *I* denote the total number of initiated VMs. Let *c* denotes the individual customer request and let *C* denote the total number of customer requests arrived at time *t*. Let a_c denote the number of accounts requested by customer request *c*. The maximum number of accounts can be accepted by the large VM is defined as *M*.

The minimum number of VMs required to allocate all requests, *Min(VM)* can be expressed by:

$$\text{Min}(VM_l) = \frac{\sum_{c=1}^{C} a_c}{M(VM_l)} \quad (4.24)$$

According to the Equation (4.23) and (4.24), the minimum number of VMs required to allocate the same number of all requests for the small VM, medium and large VM are expressed by (4.25), (4.26) and (4.27):

$$\text{Min}(VM_{small}) = \frac{\sum_{c=1}^{C} a_c}{M(VMs_{mall})} \quad (4.25)$$

$$\text{Min}(VM_{medium}) = \frac{\sum_{c=1}^{C} a_c}{2 \times M(VMs_{mall})} \quad (4.26)$$

$$\text{Min}(VM_{large}) = \frac{\sum_{c=1}^{C} a_c}{10 \times M(VMs_{mall})} \quad (4.27)$$

The price cost by using the small, medium and large type VM for initiating minimum number of VMs are expressed below:

$$\text{VMCost}_{small} = \text{Min}(\text{VM}_{large}) \times 0.12 = 0.12 \times \frac{\sum_{c=1}^{C} a_c}{M(VMs_{mall})} \quad (4.28)$$

$$\text{VMCost}_{\text{medium}} = \text{Min}(\text{VM}_{\text{medium}}) \times 0.48 = 0.48 \times \frac{\sum_{c=1}^{C} a_c}{2 \times M(VMs_{mall})} = 0.24 \times \frac{\sum_{c=1}^{C} a_c}{M(VMs_{mall})} \quad (4.29)$$

$$\text{VMCost}_{\text{large}} = \text{Min}(\text{VM}_{\text{large}}) \times 0.96 = \frac{\sum_{c=1}^{C} a_c}{10 \times M(VMs_{mall})} \times 0.96 = 0.096 \times \frac{\sum_{c=1}^{C} a_c}{M(VMs_{mall})} \quad (4.30)$$

Clearly, the above Equations prove that by initiating large VMs the total cost is minimized, when the total number of accounts requested is greater than the capability of a large VM. In addition, for a SaaS provider the total number of accounts requested by customers should always be greater than a large VM's capability, otherwise it means that this SaaS provider does not have market share. Therefore, the lower bound minimize cost is achieved by initiating large VMs to serve all requests without space wastage and penalty cost.

4.4 Performance Evaluation

We present the performance results obtained from an extensive set of experiments comparing the proposed algorithms with the best algorithm introduced in our previous chapter [114]. We discuss the experiment methodology along with performance metrics and detailed QoS parameters. Our analysis of results shows the impact of (1) reservation strategies and (2) QoS parameters: customer's QoS parameters (request arrival rate, proportion of upgrade requests, and credit level) and SaaS provider's parameters (service initiation time and penalty rate).

4.4.1 Experimental Methodology

We used CloudSim Toolkit [80] to model and simulate the proposed algorithms for resource provisioning. We simulated a data center with 500 physical machines whose configuration resembles are Amazon EC2 large image. A number of VMs of different types that are mapped to a physical machine is shown in *Figure 4.2*. Configuration details of three different types of VMs (small, medium and large) are given in *Table 4.2*. The bandwidth of the network connecting physical machine is 10 Gb. The general scheduling policy is time shared scheduling. We have extended the existing Cloud environment and added our algorithm for SLA-based resource provisioning. We model the execution time (i.e. service processing time) based on what we measured from dynamic CRM 4.0 system on a VM with Windows Server 2008R2 OS and 10Gb bandwidth over 2 weekdays and a weekend. For an operation of 303 items records, the mean time

for query response time was 2.0 second with a standard deviation of 0.2 second.

We observe the performance of the proposed algorithms by considering performance criteria from both customers' and SaaS providers' perspectives. From customers' perspective, CSL improvement is considered as reducing SLAs violations (from provider's perspectives this is KPI Assurance) and improving service quality (from provider's perspectives this is KPI Performance) in the experiment section. Although in the proposed algorithms only minimization of SLA violations is considered. The number of SLA violations is defined as the number of requests which experience slower response time than the specified in the SLA. Service Quality Improvement (SQI) for an algorithm in the system model is defined as how much faster the actual response time *respT (actual)* than the SLA pre-defined response time *respT (SLA).*

$$SQI = respT(SLA) - respT(actual) \quad (4.23)$$

In experiments, how much the response time of a proposed algorithm is faster than the base algorithm and is calculated as below:

$$ServiceQualityImprov. = SQI\ (\text{base algorithm}) - SQI\ (\text{proposed algorithm}) \quad (4.24)$$

From SaaS providers' perspective, how much the total cost is reduced by minimizing the number of VMs is observed. Therefore, there are four performance measurement metrics: the total cost, number of initiated VMs, percentage of SLA violations, and service quality improvement.

In this chapter, experiments are designed from the following three high level considerations:

1) Impact of reservation strategies: The credit level is defined by multiple parameters including (1) company type, which is based on company size (2) customer actual requirements (3) customer expressed future interest. We look into different resource reservation strategies to analyse how dynamic (based on credit level) and fixed reservation strategies impact on performance metrics.

2) Impact of QoS parameters: Which algorithm performs better in which situation by varying arrival rate, proportion of upgrade requests, credit level, service initiation time and penalty rate?

3) Performance Analysis under Uncertainty Future Interest Value: To evaluate the performance of our algorithms in handling the uncertainty in the future interest value.

All the parameters used in the simulation study are given in the following sections.

4.4.2 QoS parameters

Customers' Side

From the customers' side, three parameters (request arrival rate, proportion of upgrade requests, and credit level) are varied to evaluate their impact on the performance of our proposed algorithms. Requests arrival rate follows a Poisson distribution as suggested by previous publications [100][138]. We use a normal distribution (standard deviation = (1/2) x mean) to model all parameters, because there is no available workload specifying these parameters.

Five different types of request arrival rate are used by varying the mean from 200 to 650 simulated customers per second. The probability of a customer to have small, medium and large company type is equal.

Five different variations in the proportion of upgrade requests are used by varying the mean proportion of upgrade requests from 20% to 80%.

Five scenarios varying the proportion of customers having a credit level factor ≥ 1. This proportion is varied from 10% to 90% ('very low' to 'very high' proportion of companies having high credit level).

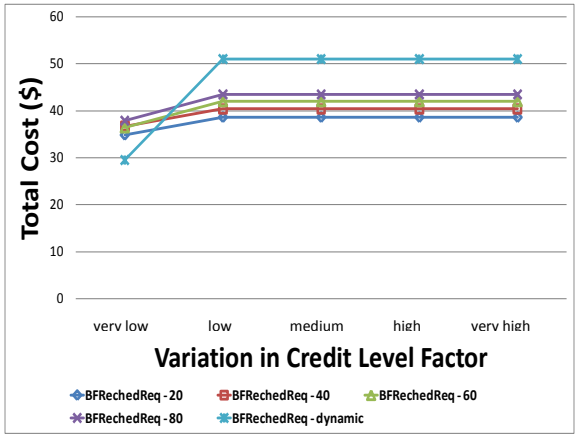
SaaS Providers' Side

A SaaS provider offers three product editions (*Table 4.2*). Due to unavailability of the public data of the SaaS provider's spending on VMs, we have used the price schema of Amazon EC2 [106] to estimate the cost per hour of using a hosted VM. It is a reasonable assumption, since today many SaaS providers lease resources from IaaS providers rather than maintaining their own resources. Resource price and capabilities, which are used for modeling VMs, are shown in *Table 4.2*.

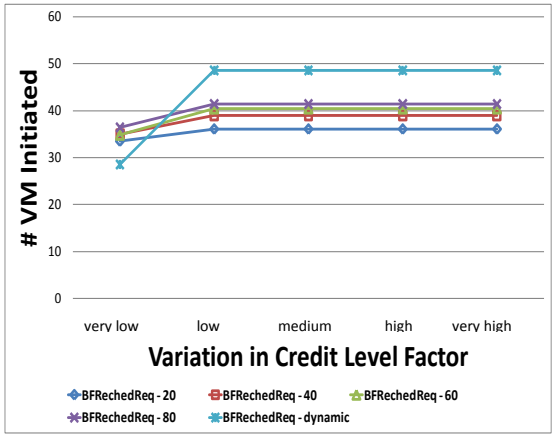
- Five different types of service initiation time (mean value varies from 5 to 15 minutes) were used in the experiments. The mean of initiation time is calculated by conducting real experiments of 60 samples on Amazon EC2 [106] over four days (2 week days and a weekend) by deploying different editions of products.
- The penalty cost is modelled by Equation (10) and it depends on the request type. The mean of penalty rate (β) varies from \$3 per second (very low) to \$12 per second (very high).

4.4.3 Results Analysis

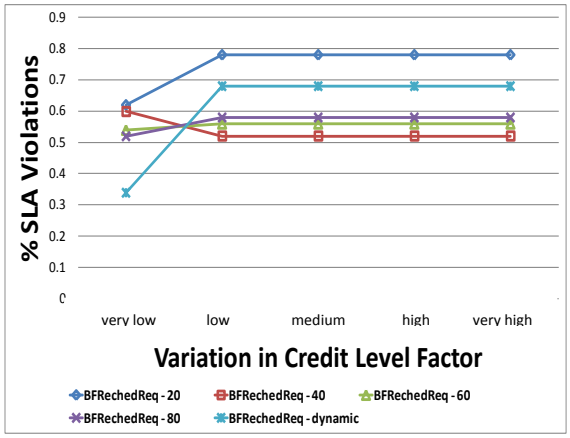
We evaluate our proposed algorithms – BFResvResource and BFReschedReq by examining the impact of QoS parameters on the providers' KPIs. For all results, we present the average obtained from 5 experiment runs. In the following sections, we examine various experiments by varying both customers' and SaaS providers' SLA properties to analyze the impact of each parameter. The mean response time which governs SLA violations is set at 5 seconds for 'first time rent' requests, 10 seconds for 'upgrade product' requests and 3 seconds for 'add account' requests.



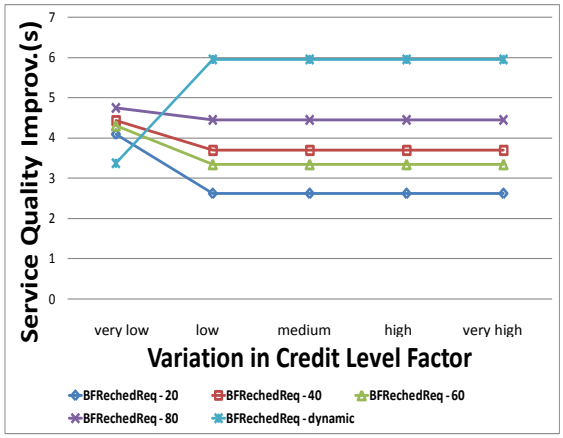
(a). Total Cost



(b). Number of initiated VMs



(c). Percentage of SLA Violations



(d). Service Quality Improvement

Figure 4.6 Impact on reservation strategy during the variation in proportion of customers with high credit level

Impact of Reservation Strategies

In this set of experiments a dynamic and four fixed (20%, 40%, 60%, and 80%) reservation

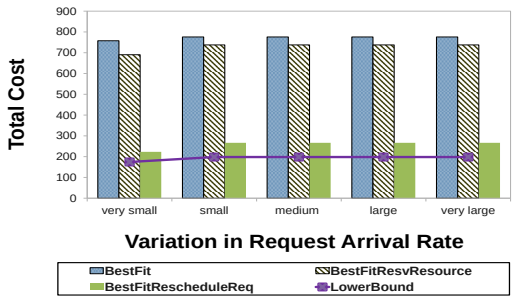
strategies are examined by varying the proportion of high credit level customers, for instance, 20% reservation strategies mean reserve 20% more space during resource reservation.

In *Figure 4.6*, the variation in credit level (x-axis) indicates the variation in the proportion of customers having high credit level. For instance, the 'very low' credit level indicates that most customers have very low credit level. Fixed (20%) reservation strategy costs the least (about 20% higher) by utilizing the least number of VMs, but responses slowest (about 60% slower) when the credit level is not very low. The dynamic strategy performs the best with respect to the response time but costs the most, because it initiates the largest number of VMs, when the credit level is high.

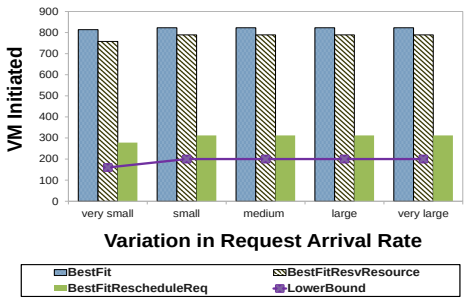
In regard to the customer satisfaction level, there are two aspects: (1) how many requests experience violations (*Figure 4.6c*), and (2) the service quality improvement *(Figure 4.6d).* In conclusion, during the service type variation experiments, dynamic reservation gives the best service quality improvement, but the fixed reservation saves the most cost. Varying the credit level has the greatest impact on the results, although the overall conclusions are the same as those obtained from the experiments which varied the other parameters, such as upgrade frequency. On the other hand, when the credit level is very low, the dynamic strategy saves the largest amount of cost and incurs the smallest number of SLA violations.

Impact of QoS parameters

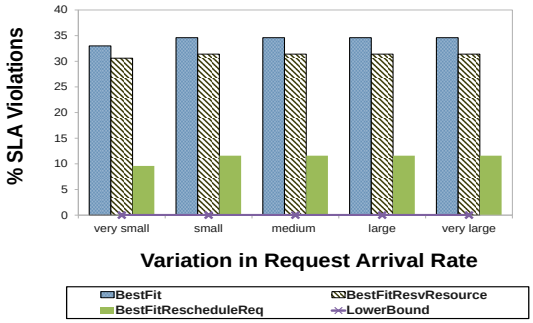
a) Impact of arrival rate variation



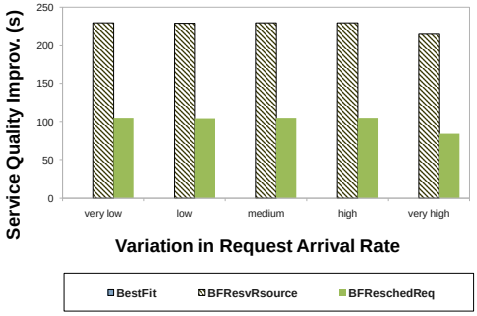
(a). Total cost



(b). Number of initiated VMs



(c). Percentage of SLA Violations

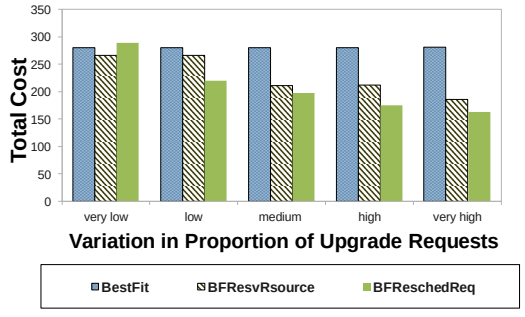


(d). Service Quality Improvement

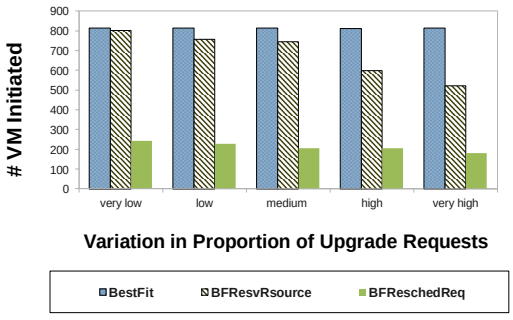
Figure 4.7 Impact of request arrival rate variation

In this section, we present the performance results of our proposed algorithms in different scenarios. In each experimental scenario, we varied one QoS parameter and set others as constant. For instance, the scenario considered for credit level is 'medium', which indicates the medium proportion of companies with high credit level. The reason for presenting the 'medium' is to minimize the impact of other factors during the evaluation of reservation strategies. For all experiments, only dynamic reservation strategy is used in algorithms, since it performs best among other evaluated reservation strategies.

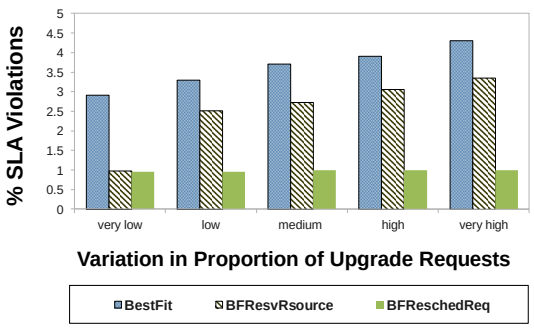
The impact of arrival rate on our algorithms is depicted in *Figure 4.7* with the following parameter settings: 'low' upgrade frequency, 'low' initiation time, and 'medium' for all rest parameters. The lower bound is plotted in line chart. The *BFReschedReq* is the closest to the lower bound, and it is 18 times and 13 times closer than the *BFResvResource* and the *base* algorithm respectively.



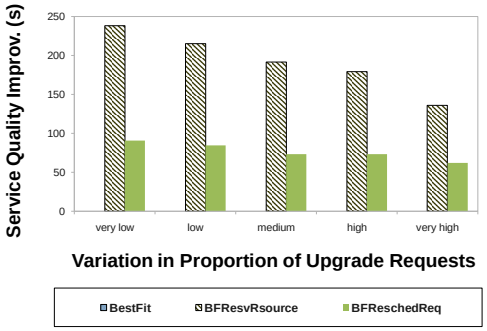
(a). Total cost



(b). Number of initiated VMs



(c). Percentage of SLA Violations



(d). Service Quality Improvement

Figure 4.8 Impact of proportion of upgrade requests variation

On average, the *BFReschedReq* performs the best by saving about 50% of the cost and reducing 60% of the SLA violations by using approximately half the number of VMs compared with the base algorithm. As *Figure 4.7c* shows, when the request arrival rate is 'very high', the *BFResvResource* causes more SLA violations than other algorithms, because when a large number of concurrent requests arrive, they increase the response time for upgrading the services (*Figure 4.7d*). However, the total cost generated by this algorithm is lower than the by the base algorithm due to a lower VM cost. It can be seen from Figure 6d that *BFReschedReq* has a smaller improvement in service quality compared with other algorithms, because of the additional time consumed by request rescheduling in transferring data and initiating new VMs. In addition, *Figure 4.7 a* and *d* show that as the service quality improves but costs more. Therefore, during the variation of the arrival rate, the *BFReschedReq* performs best in respect to the total cost, the number of initiated VMs and causes the least number of SLA violations.

b) impact of proportion of upgrade requests variation

We investigate the strengths and weaknesses of the algorithms by varying the proportion of upgrade requests from 'very low' to 'very high'. In *Figure 4.8*, 'very low' is when there is no product upgrade but low level of 'add account' upgrade. 'low' is when there is low proportion of both 'product upgrade' and 'add account upgrade'. 'medium', 'high', and 'very high' is when there is 'medium', 'high', and 'very high' proportion of both upgrades respectively. Other parameter settings are: 'very high' for request arrival rate, 'low' for service initiation time, and 'medium' for the rest of parameters. As it can be seen from *Figure 4.8*, the proportion of upgrades increases, the total cost of the base algorithm slightly increases because of more SLA violations while utilizing the similar number of initiated VMs. In

contrast, the total cost that is generated by two proposed algorithms decreases, because less number of VMs are initiated by utilizing reserved resources. In the worst case scenario, our proposed algorithms deliver results similar or close to the Best-fit algorithm. When the proportion of upgrade requests is 'very low', *BFResvResource* saves more cost than the *BFReschedReq*, because *BFReschedReq* uses large VMs, which cost more than the small and medium VMs. However, when the proportion of upgrade requests varies from 'low' to 'very high', the *BFReschedReq* saves cost over the *BFResvResource*, because *BFReschedReq* takes care of product upgrade penalty (SLA violations) and utilizes less VMs to serve an increasing number of product upgrade requests.

To compare with the base algorithm, on average *BFReschedReq* reduces the cost more than 27% when the proportion of upgrade requests varies from 'very low' to 'very high', because it initiates about 30% of the number of VMs (*Figure 4.8b*) and SLA violations reduces to about 1% (*Figure 4.8c*). The overall trend of SLA violations is increasing (*Figure 4.8c*). Nevertheless, when the upgrade frequency varies from 'low' to 'very high', the *BFReschedReq* causes more SLA violations than the *BFResvResource*, because the *BFReschedReq* cannot prevent SLA violations caused by product upgrade.

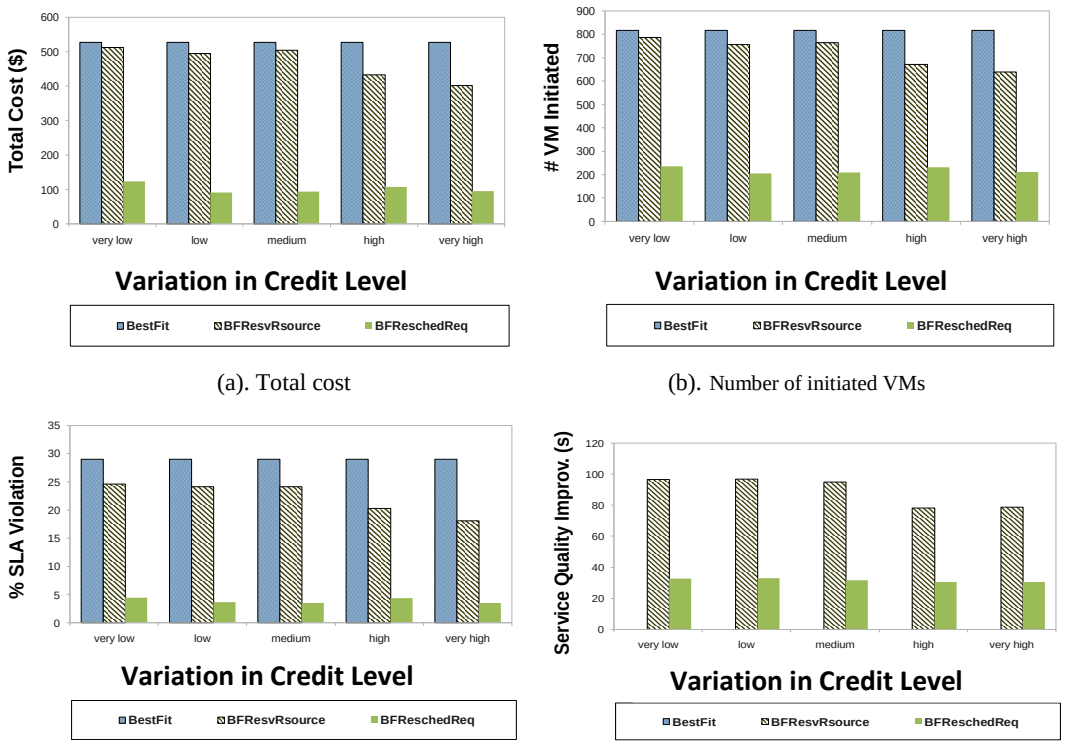
In regard to the service quality improvement, the *BFReschedReq* takes more time for rescheduling and the *BFResvResource* provides better service quality, because the *BFResvResource* takes about half of the time than that the *BFReschedReq* takes to respond to the customers' requests (*Figure 4.8d*).

c) Impact of credit level variation

To investigate the impact of customer profiles, we investigate how the proportion of high credit level customers impacts the performance of our algorithms. In *Figure 4.9*, the variation in credit level (x-axis) indicates the variation in the proportion of customers with high credit level. Parameter settings are: 'very high' value of requests arrival rates, 'very high' value of upgrade proportion, and 'medium' value of all rest parameters. It can be seen from *Figure 4.9* that there is no influence on the base algorithm, which does not consider customer profiles. However, our proposed algorithms are affected during the variation of proportion of high credit level customers, because our algorithms reserve resources according to the credit level.

When the proportion of high credit level customers varies from 'very low' to 'very high', proposed algorithms generates less cost than the base algorithm by initiating up to 12% less number of VMs (*Figure 4.9b*) and violating up to 6% less SLA violations (*Figure 4.9c*). This is because the wastage of reserved resources is lower, when the credit level increases. The

service quality improvement decreases for both proposed algorithms (*Figure 4.9d*), because it takes longer to serve the same number of requests using fewer VMs.



(a). Total cost

(b). Number of initiated VMs

(c). Percentage of SLA Violations

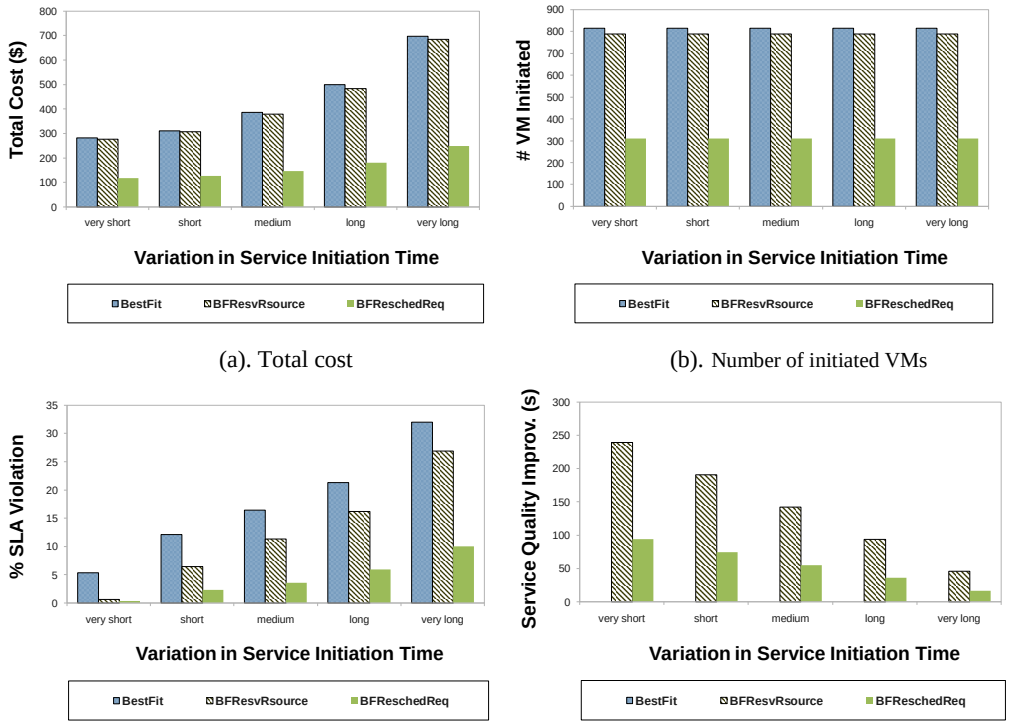
(d). Service Quality Improvement

Figure 4.9 Impact of credit level variation

d) Impact of service initiation time variation

Figure 4.10 shows how service initiation time variation impacts the SaaS provider's total cost. Parameter settings are: 'very high' value of requests arrival rate, and 'medium' value of all rest parameters. When the initiation time varies from 'very short' to 'very long', the trend of the total cost generated by all algorithms increases about 1.5 times, because it causes penalty delays (SLA violations) resulted in new service initiation. The base algorithm is affected more when service initiation time varies from 'long' to 'very long', because it initiates more VMs. The service quality improvement falls down during the enlargement of service initiation time, because the service initiation time includes the time for deploying software services.

e) Impact of penalty rate variation



(a). Total cost

(b). Number of initiated VMs

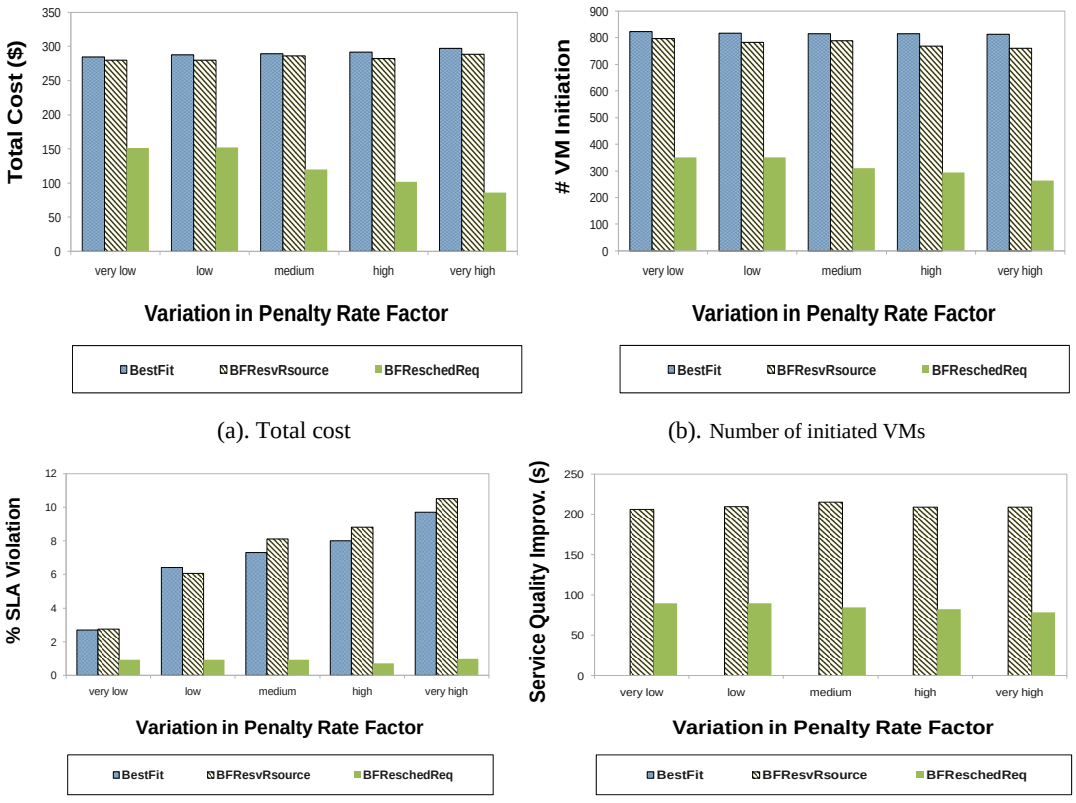
(c). Percentage of SLA Violations

(d). Service Quality Improvement

Figure 4.10 Impact of service initiation time variation

How the penalty rate (β) impacts our algorithms is investigated. Parameter settings are: 'very high' requests arrival rate, 'low' value of service initiation time, and 'medium' value of all rest parameters. It can be observed from *Figure 4.10* that all algorithms are affected during the variation of the penalty rate, because requests are scheduled with shared resources. When penalty rate varies from 'very low' to 'very high', the base and the *BFResvResource* algorithms cost more because of more SLA violations. However, the *BFReschedReq* saves cost and causes very small number of SLA violations (the maximum percentage is less than 1%).

When penalty rate varies from 'medium' to 'very high', the *BFResvResource* initiates less VMs by using reserved resources, which causes more SLA violations. Because the *BFResvResource* may delay first time rent requests to serve upgrade requests. In summary, *Figure 4.11* shows that the *BFReschedReq* minimizes the total cost, although penalty cost grows during penalty rate variation.



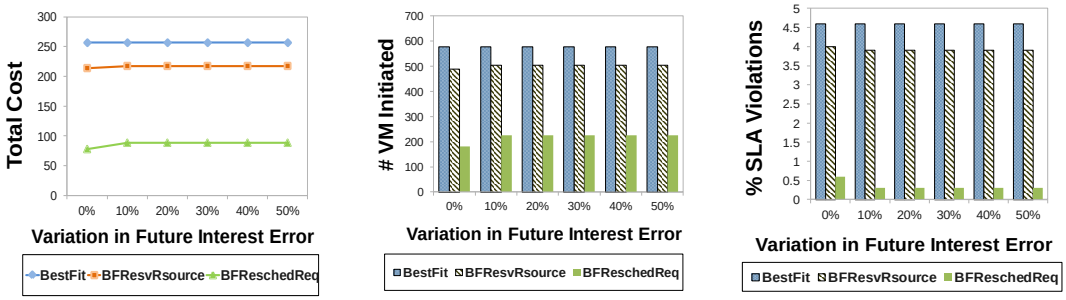
(a). Total cost (b). Number of initiated VMs

(c). Percentage of SLA Violations (d). Service Quality Improvement

Figure 4.11 Impact of penalty rate factor variation

Performance Analysis under Uncertainty Future Interest Value

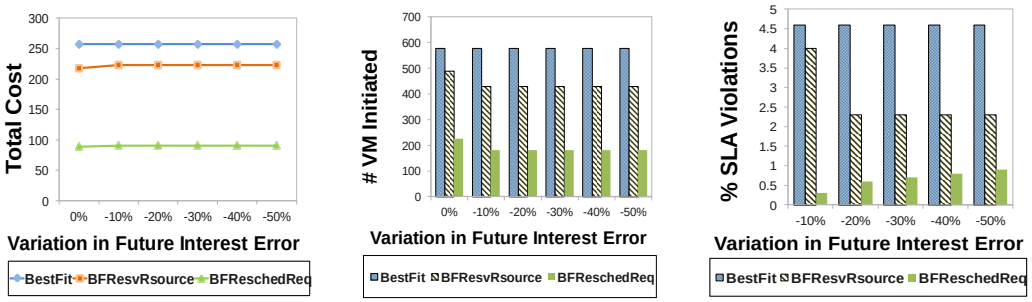
Since customer may be uncertain about their future interest value, they may under-claim or over-claim the value. To evaluate the performance of our algorithms in handling the uncertainty in the future interest value, we carried out two sets of experiments by varying the (1) future interest from 10% to 50% over-claim (*Figure 4.12*). (2) future interest from 10% to 50% under-claim (*Figure 4.13*). The base algorithm (BestFit) is not impacted since it does not consider resource reservation.



(a). Total Cost (b). Number of initiated VMs (c). Percentage of SLA Violations

Figure 4.12 Impact of Future Interest Error (Over-Claim)

Figure 4.12 shows that during the over-claim of customers' specified future interest value, the total cost (*Figure 4.12a*) increases for both proposed algorithms (upto 10%). This is because more VMs are initiated for resource reservation. However, the SLA violations have decreased due to availability of more reserved resources than required.



(a). Total Cost (b). Number of initiated VMs (c). Percentage of SLA Violations

Figure 4.13 Impact of Future Interest Error (Under-Claim)

Figure 4.13 shows that during the under- claim of the future interest, the total cost (*Figure 4.13a*) is increasing for both proposed algorithms (upto 2%). This is because of more SLA violations, which is due to under allocation of required resources.

The summary of heuristic comparison results regarding to total cost to show on which condition each algorithm can get best and worst results are presented in *Table 4.3*.

Table 4.3 The summary of best and worst results (cost) comparison

Algorithms	**Time Complexity**	**Overall performance**				
	O(IK+K)	**Arrival Rate**	**Proportion of Upgrade Requests**	**Credit Level**	**Service Initiation Time**	**Penalty Rate Factor**
BestFit	$O(IK+I^2)$	Best (very small) Worst (very large)	Best (no upgrade) Worst (very high)	No effect	Best (very short) Worst (very long)	Best (very high) Worst (very low)
BFResv Resource	$O(IK+I^2)$	Best (very small) Worst (very large)	Best(only add account upgrade) Worst (very high proportion of product upgrade)	Best (very high) Worst(very low)	Best (very short) Worst (very long)	Best (very high) Worst (very low)
BFReschedReq		Best (very small) Worst (very large)	Best (very high proportion of product upgrade) Worst (no product upgrade)	Best (very high) Worst(very low)	Best (very short) Worst (very long)	Best (very high) Worst (very low)

4.5 Related Work

Research on market driven resource allocation was started in early 80s [69][72]. Most market-based resource allocation methods [6] are designed for fixed number of resources [48][104] [118][119]. Our work is related to user driven SLA-based economic-oriented resource provision with dynamic number of resources. In the following sub-sections, we present related publications in Grid and Cloud computing that focus on the area of resource allocation and SLA management. In addition, the resource usage patterns and usage prediction are related areas to our work. The discipline of Web Usage Mining (WUM) has grown rapidly in the past few years, despite the crash of the e-commerce boom of the late 1990s. WUM is the application of data mining

techniques to Web clickstream data in order to extract usage patterns [139]. In the current WUM area, the data has been classified as content, structure, usage and user profile [139]. The first three data categories are related to the usage of Web sites but not the e-commerce transactions. Current three types of usage prediction algorithms, which are history-based, sequence-based and MARKOV-based algorithms [139][142] are mainly used in the first three data categories. Thus, in this chapter as our first attempt we consider user profile and using history-based method as the basis of the transaction-based enterprise system usage prediction to calculate the credit level.

In the following sub-sections, we present related publications in Grid and Cloud computing that focus on the area of resource allocation and SLA management.

4.5.1 Grid

Harnscher et al. discussed typical scheduling strategies in computational Grids [116]. They have considered scientific tasks, which run for short term, whereas we consider transaction based applications, which run for long term. Moreover, customer driven scenarios are out of their scope. In addition, the evaluation metrics are different, because they focused on the response time and utilization, while we focus on the cost and the number of SLA violations.

Gomoluch et al. proposed market-based resource allocation algorithms for Grid computing [117]. The common points between their and our chapter are: firstly, the consideration of state-based and pre-emptive strategies. The state-based strategy indicates all resource allocation based on the current service/system state. The pre-emptive strategy means tasks assigned to a resource, and they are allowed to be migrated to other resources for some advantageous purposes. Secondly, both chapters focused on market-based resource allocation. Nevertheless, their work considered independent tasks with input data, deadline as QoS parameters using fixed number of resources. In our case, a customer requests the enterprise applications with multiple QoS parameters using dynamic and flexible resources.

He et al. introduced a QoS guided task scheduling algorithm in Grid [128]. The bandwidth was considered as one of the major QoS parameters; and their strategy was based on the earliest completion time, while our chapter focuses on minimizing the cost by considering QoS parameters on both customer and provider side.

Reig et al. contributed to minimizing the resource consumption for serving requests and executing them within the deadline with a prediction system [105]. Their prediction system enables the scheduling policies to discard the service of a request, if the available resource cannot complete the request within its deadline. However, in our work, we consider the data intensive transaction based application, which run for long term, whereas they considered compute intensive independent application, which are relatively short term. Moreover, the QoS parameters we considered are different from the ones in their work. In addition, our model considers penalty and market oriented targets which do not exist in their work.

Fu et al. proposed an SLA-based dynamic scheduling algorithm of distributed resources for streaming [112]. Moreover, Yarmolenko et al. evaluated various SLA-based scheduling heuristics on parallel computing resources with two evaluation metrics: resource (number of CPU nodes) utilization and income [113]. Nevertheless, our work focuses on scheduling enterprise applications on VMs in Cloud computing environments (the minimum unit of resources in our work is the number of VMs).

4.5.2 Cloud

As virtualization is a core technology of Cloud computing, the VM placement has become crucial [123][124][125] in the resource management and scheduling, while the virtualization at the operating system (such as, VMware [119]) and storage (such as [120]) level is entering the mainstream. For instance, Grit et al. investigated various algorithms for assignment of VMs[123]. Similarly, Van et al. proposed the resource provisioning and VM placement [124]. Hermenier et al. designed a dynamic consolidation mechanism for homogeneous resources [125]. However, these related publications [123][124][125] did not consider monetary cost or uncertainty of future demand. Bobroff proposed a dynamic heuristic-based VM placement methodology that did not focus on customer-driven scenario to minimize the total cost for SaaS providers [126].

Kimbre et al. proposed an allocation algorithm to minimize the number of VM migrations during resource reallocation [121]. Khannaet al. pursued the goal to minimize the number of VM migrations and the number of physical machines [122]. In contrast, the objective of our work is to minimize the total cost and number of initiated VMs by considering request migrations instead of VM migrations.

Popovici et al. mainly considered QoS parameters on the resource provider's side, such as price and offered load in Cloud computing [104]. Lee et al. investigated the profit driven service requests scheduling for dependent tasks without user-driven consideration [42]. In contrast, our work focuses on SLA driven QoS parameters on both user and provider sides; and solves the challenge of assigning dynamically varying customer requests to minimize the cost and number of SLA violations.

Chaisiri et al. proposed optimisation of resource provisioning cost in Cloud computing by applying stochastic programming approach in multiple phases [127]. They minimized the cost by considering the uncertainty which is only a part of our objective. In the context of the resource allocation algorithms for enterprise applications, Yang et al. used Genetic Algorithm (GA) in their chapter [110]. As GA-based algorithms create a pre-planing schedule, they will not be able to deal with dynamic environment such as Cloud. Therefore, this approach is not suitable for SLA-based resource provisioning in dynamic Cloud computing environments. This chapter improves our previous work [114] by proposing two extended algorithms and considering additional QoS parameters such as credit level. We also propose resource provisioning and request migration strategies to optimize the total cost and SLA violations.

In summary, our work is unique in the following ways:

- It manages the CSL based on the customer QoS requirements by minimizing the SLA violations.
- The utility function is time-varying that considers dynamic VM deployment time (service initiation time).
- It considers KPI criteria as a decision making approach for scheduling.
- Scheduling algorithms consider the customer profiles to minimize penalty cost.
- It adapts to dynamic resource pools and consistently evaluates the cost of adding new instances, while most of the previous chapters deal with a fixed size of resource pool.

4.6 Summary

This chapter focused on customer driven SLA-based resource provisioning for SaaS providers with the explicit aim of cost minimization while maximizing CSL to achieve SaaS providers' objectives. To achieve this goal, we answered questions raised in the Section 4.1 by considering customer profiles and KPI criteria while using mapping and scheduling mechanisms to deal with the dynamic demands and resource level heterogeneity. We implemented two customer driven

algorithms that consider various QoS parameters (such as arrival rate, service initiation time, and penalty rate) from both customers' and SaaS providers' perspectives using respectively resource reservation and request rescheduling strategies. In addition, in order to find out how many resources should be reserved to further optimize the solution, for each QoS parameter, we implemented five sets of reservation strategies (one dynamic and four fixed percentage reservation strategies).

The analysis of our evaluation focused on customers' and SaaS providers' perspectives to maximize various KPI criteria, including total cost, number of initiated VMs, percentage of SLA violations, and service quality improvement. Simulation results showed that on average, the *BFReschedReq* results in maximum cost savings and the lowest number of SLA violations compared with the other evaluated algorithms. In general, both proposed algorithms improved service quality to a level higher than that specified in the SLAs and the *BFResvResource* improved most in regard to the service quality. The lower bound is the ideal solution and the *BFReschedReq* is the closest to the ideal solution. The dynamic reservation strategy performed best during the scenarios of service type variation with respect to the total cost, number of initiated VMs and percentage of SLA violations in general.

The CRM application scenario is a good representative example of many enterprise applications. In addition, the scenario can also be applied to HPC (High Performance Computing) and scientific applications by mapping VM capabilities to QoS requirements. The package upgrade scenario may not be required by HPC applications, which simplifies the scenario compared to enterprise web applications. Therefore, techniques and algorithms proposed in our chapter can support a wide range of applications from many domains.

In addition, the upper bound of our proposed algorithms can be explored in the future, such as the worst case scenarios for SaaS providers are that 1). All requests concurrently come together and minimized number of requests can share the same VM. 2). All scheduled requests need to be migrated to more expensive VMs.

As customer may not always like the standard offers made, and thus need more flexible offers from provider, more sophisticated mechanism is needed to accept users. Thus in the next chapter, the SLA negotiation framework is introduced.

5 Automated SLA Negotiation Framework

This chapter propose an automatic negotiation framework to help SaaS providers to attract more customers in a more flexible and profitable way. The chapter includes negotiation framework components, negotiation policies, protocols, strategies and decision making heuristics that take into account time, market constraints and trade-off between QoS parameters. The negotiation heuristics are evaluated by extensive experimental studies of our framework using data from a real Cloud provider.

5.1 Introduction

A service level agreement (SLA) is a legal contract between providers and consumers that define the Quality of Service (QoS) that is achieved through a negotiation process [142]. Negotiation processes in Cloud are essential because participating parties (customers and SaaS providers) are independent entities with different objectives and QoS requirements. Through negotiation, players in the Cloud marketplace [159] are given the opportunity to maximize their return-on-investment.

Currently, SLAs are defined by service providers without providing customers with sufficient negotiation opportunity. Moreover, current preliminary research work [158] on automated SLA negotiation frameworks in Cloud is minimal and generally does not consider, in combination, the following two factors: 1) the dynamic nature of the Cloud, as service cost and quality are constantly changing and consumers have varying needs, and 2) time and market oriented resource allocation, as any delay incurred in waiting for a resource assignment is perceived as an overhead [145]. These two factors make answering the following questions in the design of a negotiation framework for Cloud a challenging task: 1) how to balance the trade-off between multiple QoS parameters 2) how to make a decision for acceptance and rejection of the proposal, and 3) how to generate a counter offer?

To address these questions, our proposed negotiation framework integrates a decision making system considers the current Cloud market situation, time constraints, and multiple QoS parameters. In the dynamic Cloud market, opportunities and competition between service providers can have a considerable impact on strategies and decision making processes. For example, when the competition increases or the opportunity decreases, the counter offer generation strategy is to concede faster. SaaS providers aim to accept more profitable customers with the objective of maximizing profit and market share considering the cost, market and time constraints. For SaaS customers: 1) to choose the best provider, a SaaS broker is introduced on behalf of customers to negotiate with multiple providers simultaneously in order to select the best offer, and 2) multiple QoS parameters are balanced through prioritization, which is based on customer preferences. The best offer is selected based on different objectives of the parties involved in the negotiation.

5.1.1 Motivations

Our work is motivated by: 1) the emergence of the SaaS broker model [155], and 2) the lack of automated negotiation frameworks along with decision making systems and strategies to maximize profit and improve CSL in Cloud.

The broker model has been used mainly in utility markets. Due to lack of detailed information about different providers and current market, customers prefer using brokers, which provide fast and economical solutions. Similarly, in Cloud, customers face the problem of identifying the best provider, as the number of providers is dramatically increasing. Therefore, the SaaS broker model in Cloud provides a one-stop-shop for guaranteed customer service.

Currently, in the Cloud market, brokers like ViTLive [155], Cordys [167], only provide a portal listing of different providers. However, they do not select or negotiate with providers to maximize profit and improve customer satisfaction. If negotiation is required, specialist knowledge is sourced to manage the process which incurs additional direct costs. In addition, the existing negotiation framework may not be automated [149], or suitable for Cloud specific negotiations [154].

We propose an automated Cloud negotiation framework, counter offer generation strategies, and decision making heuristics considering time and market factors to achieve various objectives for

different parties. In this way, the parallel negotiation process can be set up to maximize profit or the CSL for SaaS broker and SaaS provider. Our proposed negotiation framework can be extended for any layer (e.g. Platform-as-a-Service, and Infrastructure-as-a-Service) in Cloud.

5.1.2 Contribution

The key contributions of this chapter are: 1) a novel negotiation framework for Cloud along with decision making heuristics to achieve different objectives and strategies considering both time and market factors for counter offer generation, and 2) a prototype of our framework which is implemented proposed decision making heuristics and strategies, and compared with the latest best approach proposed by Zulkernine and Martin [152]. The experimental results demonstrate that our approach generates up to 50% increased profit and about a 60% customer satisfaction level (CSL) improvement for brokers over the base heuristic.

5.2 Automated Negotiation Framework

In order to design an automated negotiation framework in Cloud, it is important to define negotiation objectives, processes, and strategies.

5.2.1 Framework Components

The main components in our negotiation framework are: Customer Agent (CA), Broker Coordinator Agent (BCA), Provider Agent (PA), IaaS Provider, SLA Generator, Directory, Policy Database (PD), and Knowledge Base (KB).

Customer Agent: Represents a customer that submits requests for software services and registers their QoS requirements into PD.

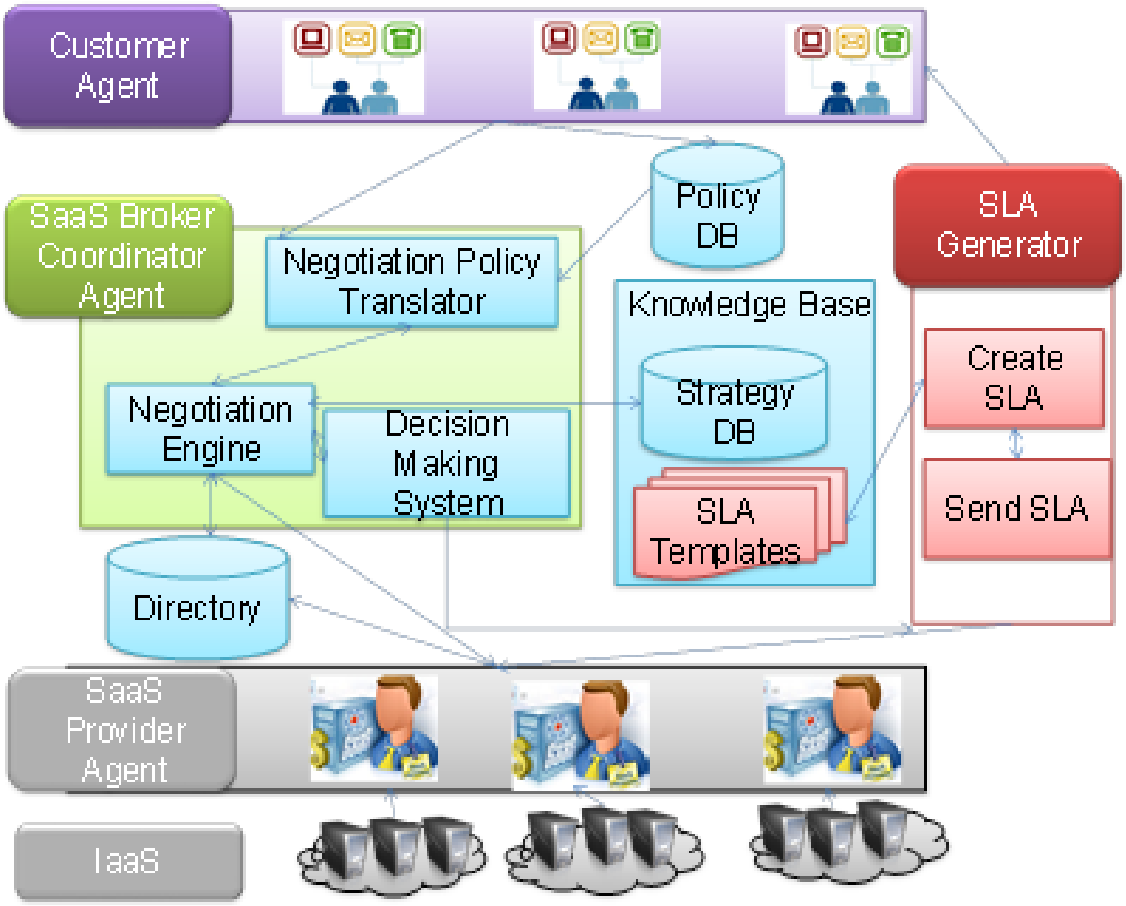


Figure 5.1 Negotiation Framework High Level Architecture

Broker Coordinator Agent: Represents the broker by receiving customer requests and negotiates with providers to achieve business objectives. It includes **Negotiation Policy Translator (NPT), Negotiation Engine (NE),** and **Decision Making System (DMS)**.

Negotiation Policy Translator: Maps customer's QoS parameters to provider level parameters.

Negotiation Engine: Includes workflows which use negotiation strategies during the negotiation process.

Decision Making System: Uses decision making heuristics to update the negotiation status.

Provider Agent: Represents the provider. PA could include the third party monitoring system to update the provider's dynamic information. Although out of the scope of this chapter, systems and processes can be implemented to monitor and measure provider capabilities.

The SLA Generator: When the negotiation has been successfully completed, the SLA Generator creates an SLA between the customer and the provider using templates retrieved from the KB. The template includes specified Service Level Objectives (SLOs) according to the QoS (SLA excludes any general legal terms and conditions).

The Directory: The repository stores the providers' registered service information.
The Policy DB: The repository stores QoS terms that both providers and customers understand.
The Knowledge Base: The repository stores negotiation strategies and SLA templates.

This chapter focus on two main components: the NE, by proposing strategies considering both time and market, and the DMS, by proposing heuristics for different objectives.

5.2.2 System Scenario

We consider three entities: consumers, SaaS brokers and SaaS providers. Each consumer *c* submits a service request to the SaaS broker, who leases software services from SaaS providers. The **customer** *c* requests services with the following attributes:

- Budget B_c: the maximum price a customer can afford.
- Software service set $SR_{b:}$ the service editions.
- The service start time t_{ss}: the latest service available time for a customer *c*.
- The contract length indicates the period of service usage *conLength*, so that customer *c* must be able to use software service within the contract term.
- The service refresh time t_r: time it takes a query operation to be executed in a software service.
- The service process time $t_{p:}$ the maximum time for a consumer *c* to wait for completing a transaction.
- The service availability *avai:* the minimum availability that the customer requires.
- The expected discount percentage for budget σ: the percentage a customer can save from their actual budget.
- The preference level of each QoS parameter γ: the absolute importance level which varies (0, 1].

The broker receives the customer request and calculates the expected budget, expected refresh time, process time, and availability. These expected values are the **best** values that the broker expects to provide to the customer and they will be proposed to providers in the quote request process. If providers cannot fulfil these expected values, the broker will adjust the expected value up to the customer requested value during the negotiation process. The broker always seeks to secure the expected value from provider.

Each **provider** offers the same or different types of services. The provider can host or lease infrastructure services from 3rd party IaaS providers.

5.3 Negotiation Objectives

In sophisticated markets, the negotiation objective is not only price but also other elements such as quality, reliability of supply, or the creation of long-term relationships. We consider multiple objectives including cost, refresh time, process time and availability. The main objectives for a customer, a SaaS broker and a provider are:

- **Customer**: minimize price and guaranteed QoS within expected timeline.
- **SaaS Broker**: maximize profit from the margin between the customer's budget and the providers' negotiated price.
- **SaaS Provider**: maximize profit by accepting as many requests as possible to enlarge market share.

5.3.1 Mathematical Models

SaaS Broker

The broker's actual budget $maxB_c$ for serving a customer c depends on the customer's budget B_c and the customer expected discount percentage σ for budget.

$$maxB_c = B_c \times (1-\sigma) \tag{5.1}$$

The initial budget proposed to all providers is the expected budget $expB_c$, which is based on the $maxB_c$ and the broker's expected margin $margin_c$:

$$expB_c = \max B_c \times (1-margin_c) \tag{5.2}$$

The profit of broker b gained from serving customer c depends on the B_c and the best provider's price $price_p$.

$$Prof_b = maxB_c - price_p \tag{5.3}$$

In the following sections, a QoS parameter shall also be referred to as an "Issue". The δ_i represents the expected improvement percentage for an issue. Therefore, the *CSL* is reflected by these Issues, which are service refresh time, process time and availability.

The expected refresh time $expT_r$ depends on the customer requested refresh time t_r and the improvement percentage for refresh time δ_r. The $expT_r$ changes during the negotiation process up to t_r.

$$expT_r = t_r \times (1 - \delta_r) \tag{5.4}$$

The customer requested service process time t_p and the improvement percentage for process time δ_p impact the expected process time $expT_p$ and varies during the negotiation process up to the t_p.

$$expT_p = t_p \times (1 - \delta_p) \tag{5.5}$$

The expected availability *expAvai* depends on the customer requested service availability *avai* and the improvement percentage of availability δ_a.

$$expAvai = avai \times (1 + \delta_a) \tag{5.6}$$

The CSL of an individual Issue csl_i depends on the variation between the current proposed value from provider $currentV_i$ and the broker expected value $\exp V_i$. The parameter υ is a value to guarantee that csl_i lies in the interval [0, 1].

$$csl_i = \frac{currentV_i - \exp V_i}{\exp V_i} \times \upsilon \tag{5.7}$$

The total customer satisfaction level CSL_c, where *i* represents the individual issue, *I* indicates all Issues, γ_i indicates the importance level of the Issue *i*, and the csl_i.

$$CSL_c = \sum_{i=0}^{I} \gamma_i \times csl_i \tag{5.8}$$

SaaS Provider

The provider's service price is based on the provider's cost $cost_p$ and expected margin $expMagin_p$. Different providers calculate price differently. The general equation for a provider to calculate price is proposed below.

$$price_p = \cos t_p + \exp Magin_p \tag{5.9}$$

The $cost_p$ depends on the base cost $baseCost_p$ (such as infrastructure cost, admin cost, software cost) and the relevant cost of satisfying each Issue *i, where* $i \in I$. Take availability as an example. To provide a higher availability than what currently exists, it may cost extra for the provider to buy another server as a mirror server. This extra cost is the relevant cost for satisfying availability.

$$cost_p = baseCost_p + \sum_{i=0}^{I} \cos t(i) \tag{5.10}$$

5.4 Negotiation Policy Specification

The negotiation policy specifications are used to specify QoS parameters, which are to be negotiated and the acceptable range of them to reach the mutual agreement [157]. In this section, we propose the QoS model and policy specification.

5.4.1 QoS Model

Different participants' using different terms is one of the critical challenges in SLA negotiation [166]. In our framework, a QoS model is used to provide shared knowledge about QoS attributes among negotiating participants. A QoS model defines a set of QoS dimensions. Each QoS dimension represents a specific quality aspect of a service, such as refresh time, availability, and price. In our QoS model, a quality dimension is defined using: a title, a category, a name, a description, and a metric. The QoS model is shared among service consumers and service providers. Thus, they have a common understanding on the QoS attributes about how they are defined, how they are measured, and so on. In this chapter, we consider the following QoS dimensions – price, refresh time, process time and availability. These dimensions are the ones that are mostly used and they are domain-independent. Our QoS model can be easily extended to include other QoS dimensions.

Before negotiation, both participants specify the rule of QoS parameter in a policy specification. The policy usually refers to a high-level description of goals to be achieved and actions to be taken in different situations.

5.4.2 Policy Specification

Our policy specification is inspired by WS-Policy and XACML. WS-Policy is a XML-based specification, in which assertions are basic blocks [167]. Each assertion defines domain specific constrains, capabilities, and requirements. However, the WS-Policy framework does not provide any assertion, and therefore users of this framework need to develop their own assertions. XACML is a XML-based language which is standardized by OASIS and has been successfully used widely as access-control policy languages [157]. With XACML, the QoS parameter constraints can be domain-independent, because XACML is based on generic data type. However, both of them are only machine-readable but not human-readable, especially for non-IT background users. Therefore, based on the concept of constraints and goals in WS-Policy and

XACML, we design our domain-independent policy in a both human-readable and machine-readable manner by providing web user interface to register constraints (rules) and goals.

The main concepts of our policy specification are rules and goals:

- The rules: are used to specify the QoS parameters and the acceptable range of these parameters (*Figure 5.2*).
- The goals: are non-negotiable rules.

Moreover, in order to take care of different policy rules from different agents we provide a rule register to extend policy flexibly.

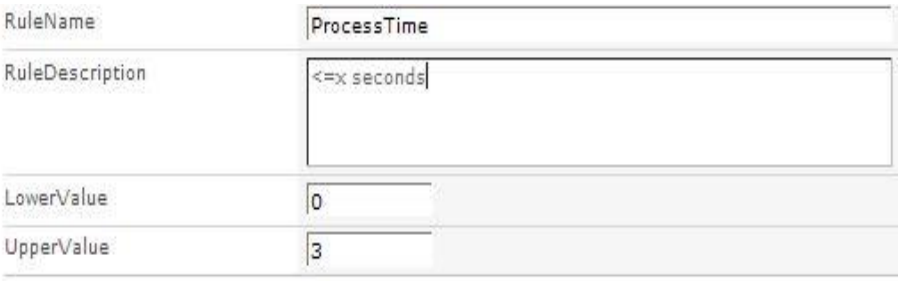


Figure 5.2 Negotiation Rule Register Web Form

In *Figure 5.2*, the rule names are QoS parameters. The lower value and upper value fields are lower and upper bounds of the rule value. If a rule does not exist, there is another interface to register new rule names. Any policy and rule registered by providers are stored in Policy DB component of the framework. The NPT component matches these policies with customer QoS parameters during the negotiation.

5.5 Negotiation Protocol

The negotiation protocol refers to a set of rules, steps or sequences during the negotiation process, aiming at SLA establishment. It covers the negotiation states (e.g. propose offer, accept/reject offer, and terminate negotiation). It is common to characterize negotiations by their settings: bilateral, one-to-many, or many-to-many. We focus on the one-to-many bargaining setting, where we consider three types of agents (CA, BCA and PA). A BCA negotiates with many PAs in a bilateral fashion.

During the negotiation process, the negotiation status is updated using negotiation states described in *Table 5.1*.

Table 5.1 The Negotiation States and Description Summary

States	**Description**
Propose	The agent propose initial or counter offer to the opponent agent.
Reject	The agent does not accept the offer proposed by the opponent agent.
Accept	The agent accepts the offer proposed by the opponent agent.
Failure	System failure, trigger renegotiation.
Terminate	Negotiation is terminated due to timeout or no mutual agreement.

In our framework, the sequential negotiation process is described as follows and depicted in *Figure 5.3*:

Phase 1: *CA submits requests*: CA requests services on behalf of the customer to the Broker.

Phase 2: The BCA requests initial proposals from all providers, who are registered in the Directory. The values sent from BCA to PAs are expected values.

Phase 3: *PAs propose initial offer:* All PAs propose initial offers based on their current capabilities and availability to fulfil BCA's requirements.

Phase 4: *Negotiation Process with PAs:*

a). If there are providers who can fulfil all requirements, then the BCA selects the best vendor.

b). If there is no provider that can fulfil all requirements, then the BCA starts the negotiation process with PAs.

*Step 1***:** BCA selects the best initial offer from all offers that are proposed by all providers according to the objective.

*Step 2***:** BCA adjusts its initial offer according to the offer selected in **Step 1** to generate new counter offer and propose it to all providers.

*Step 3***:** A PA evaluates BCA's counter proposal.

Step 4: If the counter offer proposed by BCA cannot be accepted, PA proposes a counter offer.

*Step 5***:** Terminate negotiation. There are three termination conditions: First, when negotiation deadline expires. Second, when the offer is mutual agreed by both the

CA and the PA. Third, when BCA is not able to accept any counter offer proposed by all providers within the negotiation deadline.

Phase 5: *SLA Generation:* Initiate SLA creator to generate SLA for customer and provider respectively using SLA templates stored in KB.

Phase 6: *Send SLA to all participants:* The generated SLA will be sent to the customer and provider respectively by the SLA creator.

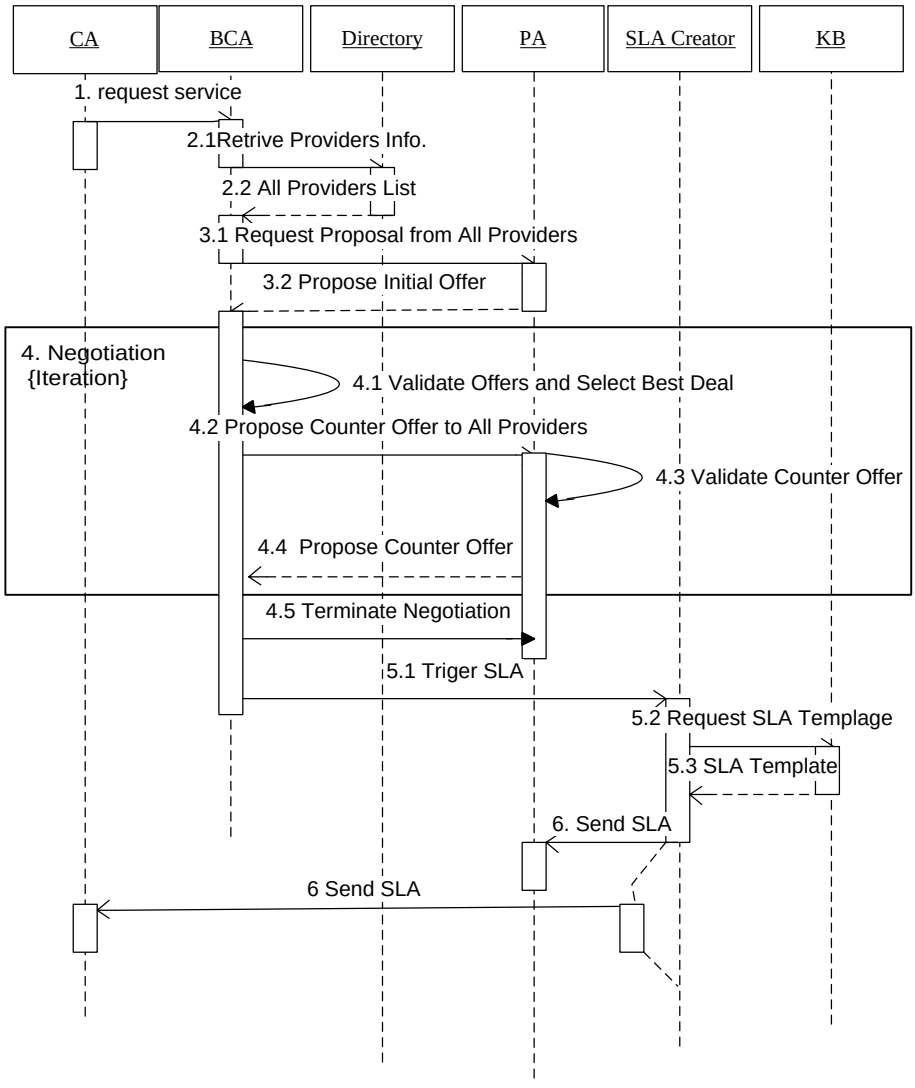


Figure 5.3 The Interaction between Components during Negotiation Process

5.6 Decision Making System

In the negotiation process, the action that a participant performs is determined by a decision making system. In the decision making system, three main questions need to be answered: 1) how to evaluate the offer; 2) what actions to take: accept, reject or generate counter offer; and 3) how to generate counter offer? We design negotiation heuristics to answer them from the broker and provider's perspectives.

5.6.1 Broker

After **BCA** requests quotes from all PAs, each PA proposes an initial offer to the BCA, which selects the best offer and makes a decision. If the decision is to propose a counter offer, then the new counter offer will be proposed to all PAs. The best offer is selected based on different objectives. We consider cost-benefit objectives as follows:

- **Minimum cost**: selects the offer with the lowest price first and then the highest cumulative CSL for all QoS.
- **Maximize CSL**: selects the offer with the highest cumulative CSL for all QoS first and then the lowest price.

Table 5.2 The Mincost Heuristic

Conditions	**Within BCA's expB**	**Exceed BCA's expB**
All QoS parameters are satisfied	If deadline condition is urgent, agree. Otherwise decrease expB.	If expB is less than actual budget, then increase expB. Otherwise reject.
Not all QoS are satisfied	Satisfy all parameters and reduce expB.	Satisfy all parameters by negotiating on minimal (not desired) values.

Table 5.3 The *Maxcsl* Heuristic

Conditions	**Within BCA's expB**	**Exceed BCA's expB**
all QoS parameters are satisfied	If deadline condition is urgent, agree. Otherwise decreases the least preference parameter to decrease expB.	Decreases the value of parameters, which are better than expected to decrease price.

Not all QoS are satisfied	Satisfy all parameters and increases expB.	Increases expB.

After selecting the best offer, the broker needs to decide how to deal with the selected best offer. One of three actions can be adopted: accept, reject or generate counter offer according to negotiation heuristics. We design two broker negotiation heuristics (*mincost* heuristic and *maxcsl* heuristic) to decide which action to take according to different objectives.

In these two heuristics (*Table 5.2* and *5.3*), cost and other Issue values are calculated using negotiation strategy functions, where the most desired and the minimal acceptable values for each Issue are considered for the broker.

In both decision making heuristics, two criteria is used to evaluate the offer: 1) weather offer is within BCA's expected budget: whether the service price offered by provider $price_p$ is less than the broker's expected budget *expB*, and 2) whether all QoS parameters are satisfied.

The above two criteria generate four combined conditions. For each condition, the decision making heuristics guide the broker to make different decisions on which Issue requires adjustment. There are two factors that require consideration when making adjustments. Firstly, trade-off between cost and QoS parameters depends on the objective. Secondly, when the broker must concede on QoS parameters, it always adjusts the least preferred parameter. After the broker decides which Issue to adjust, the new value of the Issue is calculated. The time complexity of these heuristics is *O(CPI)* depending on the number of customers (*C*), the number of providers (*P*) and the number of Issues (*I*).

5.6.2 Provider

Table 5.4 Provider's Decision Making Heuristic

Conditions	Within BCA's expB	Exceed BCA's expB
All QoS parameters are satisfied	If deadline condition is urgent, agree. Otherwise decrease the least preference parameter to decrease expB.	If expB is less than actual budget, increase expB. Otherwise decrease the QoS value.
Not all QoS are satisfied	Satisfy all parameters and increase price.	Increase price.

The provider's objective is to maximize profit by accepting as many requests as possible. Therefore, the provider does not reject requests but continues to negotiate with each broker until negotiations have ended. *Table 5.4* shows the provider's decision making heuristic.

5.7 Negotiation Strategy

The negotiation strategy underpins the counter offer generation process using various strategy functions which guide to what degree the agent concedes or bargains considering time and market factors.

The strategy functions control whether an agent concedes on certain Issues, or in the alternative, negotiates very hard in each negotiation until the deadline is reached.

The new value $newv^{i}_{a\to\wedge a}$ proposed by agent *a* (e.g. broker) to opponent ^a (e.g. provider) for Issue *i* depends on the current value of Issue *i* proposed by the opponent agent $cv^{i}_{\wedge a}$, the best expected value $bestv^{i}_{a}$ and a strategy function.

$$new^{i}_{a\to\wedge a} = cv^{i}_{\wedge a} + \alpha^{i}_{a}(\chi_1, \chi_2 \ldots \chi_n)(bestv^{i}_{a} - cv^{i}_{\wedge a}) \tag{5.11}$$

The strategy function $\alpha^{i}_{a}(\chi_1, \chi_2 .. \chi_n)$ guides the speed of adjustment, where χ_n indicates different factors (such as time, market related factors), which will be explained below.

Opportunity: At time *t*, the probability that an agent is ranked as the most preferred candidate is defined using the condition of opportunity $C_o\,(c_t, p_t)$. At time *t*, c_t indicates the number of competitors, and p_t indicates the number of partners[157].

$$C_o\,(c_t, p_t) = 1 - \left(\frac{c_t - 1}{c_t}\right)^{p_t} \tag{5.12}$$

Competition: At time *t*, the competition $C_c\,(c_t, p_t)$ in the market depends on the demand and supply ratio (equation 5.13). At time *t*, c_t indicates the number of customers, and p_t indicates the number of providers. The resource/market competition has the largest effect on the equilibrium price [157].

$$C_c\,(c_t, p_t) = \frac{c_t}{p_t} \tag{5.13}$$

Time: At time *t* the negotiation deadline condition $C_{dl}(t)$ of an agent depends on the deadline t_{nd} and negotiation start time $t_{ns.}$

$$C_{dl}(t) = \frac{t - t_{ns}}{t_{nd} - t_{ns}} \tag{5.14}$$

The negotiation period is the variation between negotiation start time t_{ns} and negotiation deadline t_{nd}. As deadline is a time-based condition, the well-adopted time-dependent result of functions, such as Linear (L), Boulware (B) or Conceder (C) are generally used to model how an agent varies its offer with time. These time-based functions are often used in negotiation systems because of their simplicity [153][154]. In this chapter, we use a similar model and consider time, market (opportunity and competition) conditions to design new strategy functions for negotiation.

For the broker, we propose the strategy function for a particular issue by considering opportunity, competition and time constraints in equation 5.15:

$$\alpha(t, c_t, p_t, \gamma) = e^{(\gamma \times C_{dl}(t)^{(\frac{C_O(c_t, p_t)}{C_C(c_t, p_t)})^{\beta}}} \times \ln k \tag{5.15}$$

For the provider, we propose strategy function for a particular issue by considering opportunity, competition and time constraints in equation 5.16:

$$\alpha(t, c_t, p_t) = e^{(C_{dl}(t)^{(\frac{C_O(c_t, p_t)}{C_C(c_t, p_t)})^{\beta}}} \times \ln k \tag{5.16}$$

In equations 5.15 and 5.16, the function $\alpha(.)$ varies from 0 to 1 and guides the changes in the values of an Issue in the subsequent counter offers from its current value to the maximum allowable value within the negotiation deadline. The *k* determines the initial offer.

In equation 5.15, γ indicates the preference of the Issue considered by the customer. The degree of compensation depends on a parameter β and reflects the conceding nature of the broker. The higher value of β (>1) results in a steeper curve, i.e., faster increment in α with time indicating a more conceding attitude of the negotiating party. The lower value of β (<1) represents the restrictive or boulware attitude. The reason for us to design our strategy using exponential and not polynomial models, is because the polynomial concedes faster at the beginning than the exponential one, even though both behave similarly on a whole level. For a small value of β the exponential waits longer than the polynomial model before it starts conceding. The objective of broker is to maximize profit by waiting as long as possible to start conceding.

5.8 Performance Evaluation

We present the performance results obtained from an extensive set of experiments by comparing our proposed heuristics with the most recently proposed heuristic (referred as *base*) [152]. The performance of each proposed heuristic *depends on three factors: time, cost and market constraints*. Therefore, to analyse how these heuristic can achieve customer, broker and provider's objectives, the following experimental scenarios are considered

- ***Impact of negotiation deadline (time factor)***: The impact of 4 sets of negotiation timeframes from the customer's perspective is observed; we use number 1 to 4 to represent the variation from 'very urgent' to 'very relaxed'.
- ***Impact of broker expected margin (cost factor)***: The impact of 4 sets of initial broker expected margins (varying from 20% to 50% over budget), are observed.
- ***Impact of market factor***: The impact of 4 sets of market factors (varying the ratio in relation to the number of providers and customers from less than 10%, 30%, 70%, and more than 90%), are observed. Numbers 1 to 4 are used to represent each set.

5.8.1 Reference Heuristic

For comparing our proposed heuristics, we used the most recent work related to our context on automated negotiation proposed by Zulkernine and Martin [9], who developed a time-based Sigmond function in their negotiation process for generating counter offers. We however, consider both time and market functions in Clouds. To compare their negotiation strategy, we have implemented their heuristics and Sigmond function with the objective of cost minimization.

5.8.2 Experimental Methodology

We implemented a prototype of the framework considering both time and market factors using real data shared with us by cloud provider CA Technologies. CA Technologies offers a number of enterprise software solutions to customers delivered as SaaS. The data provided included the response, refresh and processing times of an enterprise solution hosted on VMs, as measured by the quality assurance team. Availability data is collected from CloudHarmony benchmarking system [156], which provides real data from Cloud providers. These data are collected over 4 days including weekdays, weekends and Easter public holiday.

- ***Availability***: Varies from 98.654% (Colosseum) to 100% (Amazon EC2) as derived from Cloud Harmony.
- ***Process Time***: The mean 5.243 ($\pm$ 2.043) s.

- ***Refresh Time***: The mean 1.581 (± 1.383) s.
- ***Cost***: Cost is considered similar to Windows VMs from 3rd party IaaS providers, which varies from $0.34 per hour (VCloud Express) to $0.46 per hour (Amazon EC2).

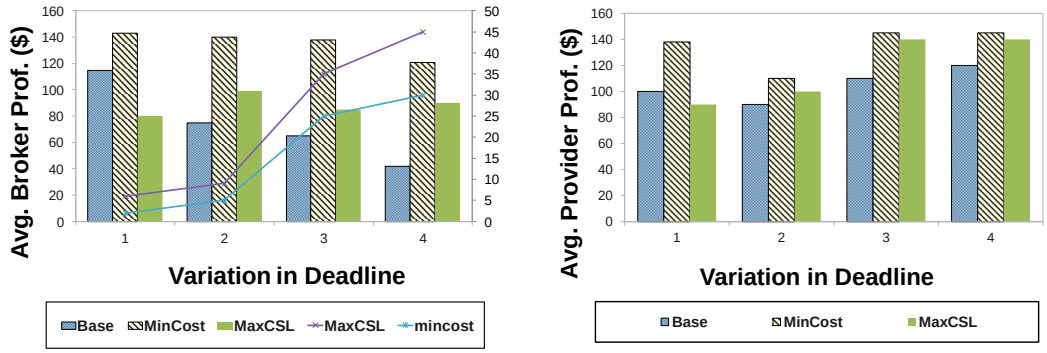
We conducted experiments considering 50 concurrent users based on the CA provided data, which is designed according to their customer historic data. The summary of customer data is:

- ***Availability***: uniformly distributed and varies from 99.95% to 100%.
- ***Process Time***: normally distributed mean 1.5 (±1) s.
- ***Refresh Time***: normally distributed mean 2 (± 1) s.
  Software service set**:** consists of 3 editions.
- ***The expected discount percentage***: normally distributed with mean value 30% (variation ± 20%).
- ***The preference level of each QoS parameter***: uniformly distributed between 0 and 1.
- ***Budget:*** normally distributed with mean $40 (± $10).

5.8.3 Result Analysis

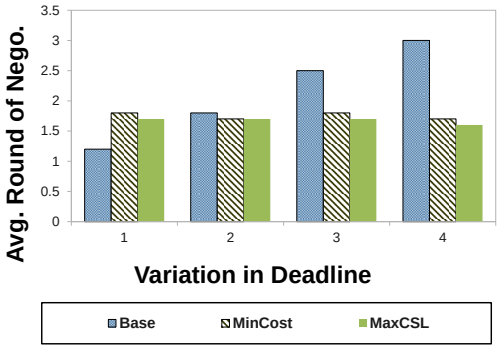
The following performance metrics are considered for evaluation based on the objectives of the negotiating parties:

- ***Average broker's profit*****:** The broker's average profit from accepted customers.
- ***CSL improvement***: The average CSL improvement over base.
- ***Average provider's profit***: The average provider's profit for accepting customers.
- ***Average round of negotiation:*** The average number of negotiations conducted during the negotiation process to reach mutual agreement.
- ***Number of successful negotiations:*** The number of successful negotiations reaching mutual agreement.

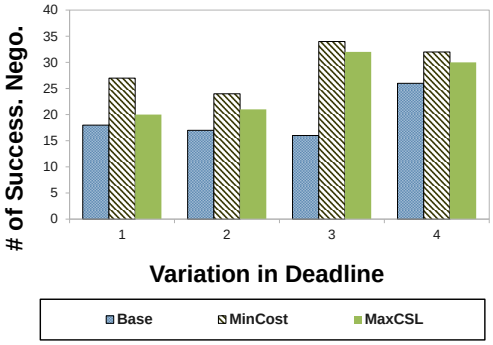


(a). Average Broker Prof.($)

(b). Avg Provider Prof.($)



(c). Avg Round of Nego.



(d). # of Success. Neg.

Figure 5.4 Impact of Deadline Variation

Variation of negotiation deadline

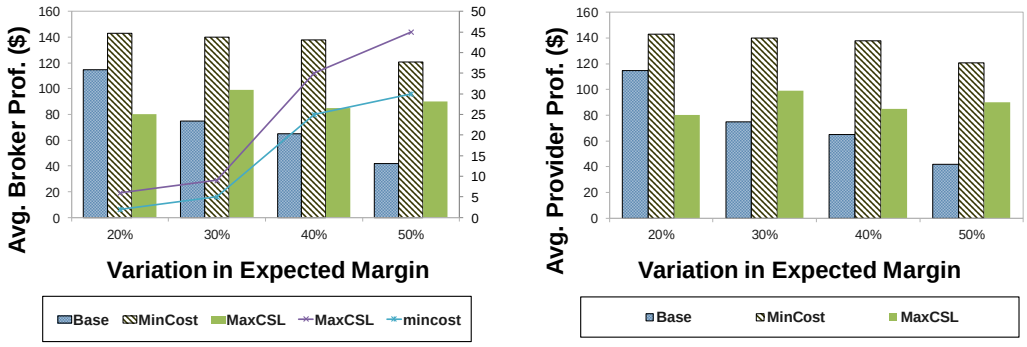
The experiment is designed to evaluate mincost and maxcsl during negotiation deadline variations.

The bar chart in *Figure 5.4a* represents average broker profit while the line chart represents the CLS improvement over base heuristic. For all the negotiation deadline variations, mincost generates the highest profit (up to 400%) for the broker over maxcsl and base. The reason for such a trend is that the broker concedes less or bargains harder for more profit. In terms of CSL improvement, maxcsl results in the highest improvement (up to 15%) over base, since it is designed to sacrifice profit for a higher CSL.

From the providers' perspective (*Figure 5.4b*), on average maxcsl generates more profit for providers, because the maxcsl aims at satisfying all Issues within the broker's budget, which leaves more profit for providers.

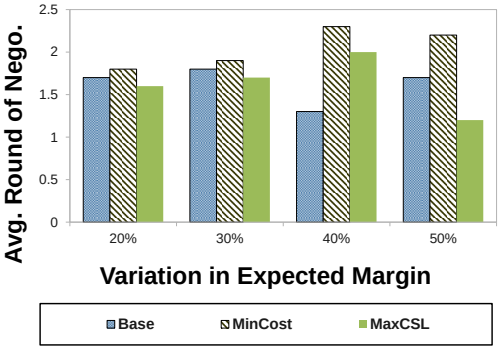
Figure 5.4c shows the average negotiation round for base increases dramatically when deadlines are varied (as base is only time dependent), whereas our proposed heuristics increases slightly (less than 2 rounds), as market factors also impact on the negotiation process. In terms of the number of successful negotiations (*Figure 5.4d*), when the deadline becomes relaxed, our proposed heuristic performs better and increases in trend, as there is more bargening time.

In summary, mincost generates more broker profit while maxcsl generates improved CSL and increased provider profit by increasing the number of successful negotiations with similar negotiation rounds.

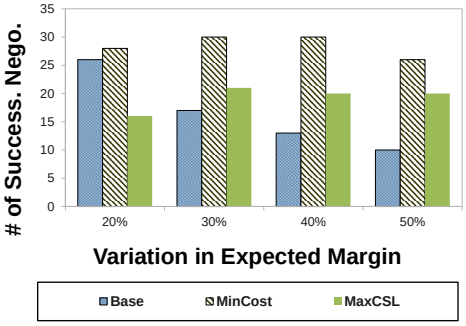


(a). Average Broker Prof.($)

(b). Avg Provider Prof.($)



(c). Avg Round of Nego.



(d). # of Success. Neg.

Figure 5.5 Impact of Variation in Expected Margin

Variation of initial expected margin

As increase in expected margin leads to reduced initial broker budget (cost), the experiment is designed to evaluate mincost and maxcsl heuristics during the varition of broker costs. The expected margin varies from 20% to 50%, since after 50% the observed trend is similar.

Figure 5.5a bar chart depicts that the mincost generates the highest profit for the broker, which is up to 200% more than the base. The line chart shows that the maxcsl has improved CSL by up to 15% over the mincost.

Figure 5.5b shows that the maxcsl generates a higher profit for providers when the broker negotiates for higher levels of CSL.

Generally, the average round of negotiations increases for all heuristics when the expected margin increases (*Figure 5.5c*), because when time and market factors are constant, the broker is required to negotiate more rounds with less budget to achieve the objectives and reach agreement.

In summary, during expected margin variations, the mincost generates more profit for the broker, whereas maxcls achieves more profit for the provider as the broker sacrifices cost for securing improved CSL.

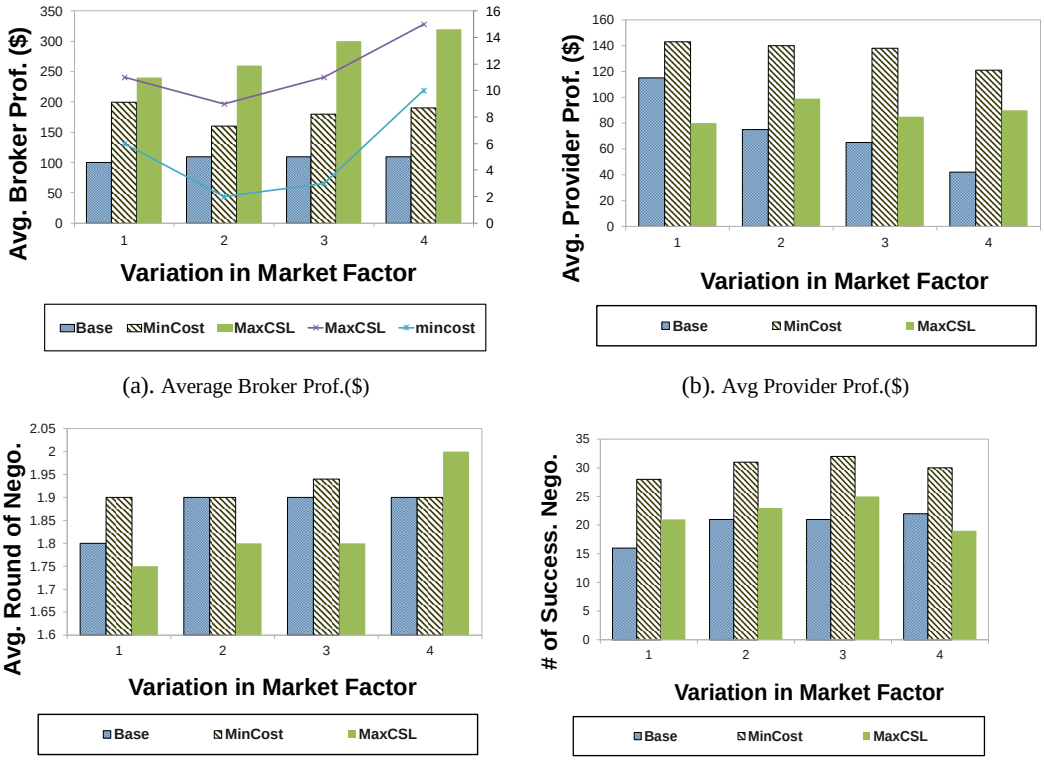
Variation of the market factor

The experiment is conducted to evaluate the proposed heuristics during the variation of market factors. When market factors vary from 1 to 4, which represents an increase in market competition, the mincost generates up to twice the profit than the base (*Figure 5.6a* bar chart) and the maxcsl improves up to 4 times more CSL compare to mincost (*Figure 5.6a* line chart). The broker's profit generated by base only changes slightly during market factor variations, as base does not consider market conditions.

Figure 5.6b *illustrates that the provider's profit decreases due to an increase in market competition. The* maxcsl *generates more profit for providers than* mincost *and* base*, as* maxcsl *considers the CSL as the highest priority, which leaves more profit for providers.*

When competition increases, more negotiation rounds are required to reach agreement (Figure 5.6c), as participants bargain harder and the number of opportunities to reach agreement increases (Figure 5.6d).

To conclude, the experiment demostrates that mincost *produces more profit while the* maxcsl *achives better CSL for the broker and more profit for providers.*



(a). Average Broker Prof.($)

(b). Avg Provider Prof.($)

(c). Avg Round of Nego.

(d). # of Success. Neg.

Figure 5.6 Impact of Market Factor Variation

5.9 Related Works

With the advancement of web technology, various approaches of resource allocation have been developed for distributed systems [160]. Current literature indicates that research focusing on resource allocation is rapidly growing. However questions remain as to whether multi-agent systems can be adopted in the domain of resource allocation. In this context several multi-agent approaches were developed to leverage the wide applicability and efficient adoption of multi-agent systems for the heterogeneous domain [161]. However, these approaches have some limitations when applied to Cloud. For example, most popular strategies such as Game theory [162], Reinforcement Learning [163] and Markov Decision Process (MDP) [164] require either expensive storage of each status or that every agent is required to expose tactics to opponents. Therefore, these approaches are not applicable for Cloud where private information such as the number of utilized resources is not advertised.

Faratin et al. presented a formal model of negotiation between autonomous agents in service-oriented environments [146]. Chhetri, et al. proposed an agent-based negotiation architecture for coordinated negotiation in service composition [147]. Comuzzi and Pernici proposed a negotiation broker framework to support semi-automated or fully automated negotiation of QoS for service selection [153]. Similarly, Zulkernine et al. proposed a policy based negotiation broker middleware framework for automated negotiation of SLA's [152]. Dastjerdi and Buyya proposed negotiation strategies for Infrastructure layer in Cloud which depends on provider resource capabilities [166]. These approaches have not considered elements such as CSL objectives, broker's profit, and market factors in their algorithms.

5.10 Summary

In Cloud computing, the SLA is a legal contract between the consumer and provider to guarantee the QoS. Negotiation is essential for both participants to feel comfortable about meeting their objectives prior to SLA finalization. In this chapter, we proposed a novel negotiation framework which included strategies and decision making heuristics by considering factors such as time, market constraints, and trade-offs.

Our two proposed heuristics have been evaluated by using real data from a cloud-hosted enterprise software solution provided by CA Technologies. Results showed that our proposed heuristics minimize cost or maximize CSL in comparison to the most recently proposed base heuristic.

Up to now we have demonstrated the efficiency of our algorithms through extensive simulation studies. In the next chapter, we develop a prototype of the system considered and show how our proposed strategies can be used in practical scenarios.

6 An SLA-based Resource Management System for SaaS Providers

To demonstrate the usefulness of key algorithms and techniques proposed in this thesis, we implemented a software prototype system, called SLA-based Resource Management System (SLARMS). This chapter presents SLARMS for adapting dynamic customer demands using cloud infrastructure resources. It covers the system architecture, and implementation are described. It concludes with a case study in enterprise software applications.

6.1 Motivation and Requirements

With the advancement of Cloud technologies, a large number of applications are delivered through software as a service (SaaS) model in Cloud computing environments. Although several existing works (noted in Chapter 2) have explored SaaS model, capabilities such as support for adapting dynamic customer demands using Cloud resources to achieve business objectives are required by many SaaS providers.

In addition, to meet requirements of SLA-based resource provisioning of Cloud applications (in Chapter 1), future efforts should focus on design, development, and implementation of software systems based on novel SLA-based resource allocation models exclusively designed for data centres.

The resource provisioning within these Cloud data centres will be driven by market-oriented principles for efficient resource management depending on customer QoS targets. In the case of a Cloud data centre as a commercial offering to enable crucial business operations of companies, there are many critical QoS parameters to consider in a service request, such as response times. In particular, QoS requirements cannot be static and need to be dynamically updated over time due to continuing changes in business operations and operational environments. In short, there

should be greater importance on customers since they pay for accessing services in data canters. Therefore, our thesis presented various SLA based customer requirements driven resource management techniques for SaaS providers to achieve their objectives. In the following sections, the realization of this vision about SLA-based resource management system is presented that includes implementation of proposed customer driven resource management techniques with evaluation of a prototype system in an operational data centre.

6.2 System Architecture

In order to fulfil the aforementioned requirements, a SaaS model for serving customers in Cloud is shown in *Figure 6.1*. A customer sends a request for software services offered by a SaaS provider, who uses three layers, namely application layer, platform layer and infrastructure layer, to satisfy the customer's request. The application layer manages all application services that are offered to customers by the SaaS provider. In the platform layer, the request monitor is used to monitor requests including new and upgrade requests. Whenever a customer changes QoS requirements, the mapper and decision maker are invoked. The mapper is responsible for translating the customer's QoS requirements to infrastructure level parameters and the decision maker is used to make decision on if the request can be accepted and where to schedule the acceptable request. In addition, the resource allocator is responsible for initiating or allocating Virtual Machines (VMs) to serve the request. Moreover, the SLA manager is used to track SLA violations according to actual resource information. Based on SLA terms, the market manager updates the final cost and profit accordingly. The infrastructure layer includes data centres where VMs are hosted.

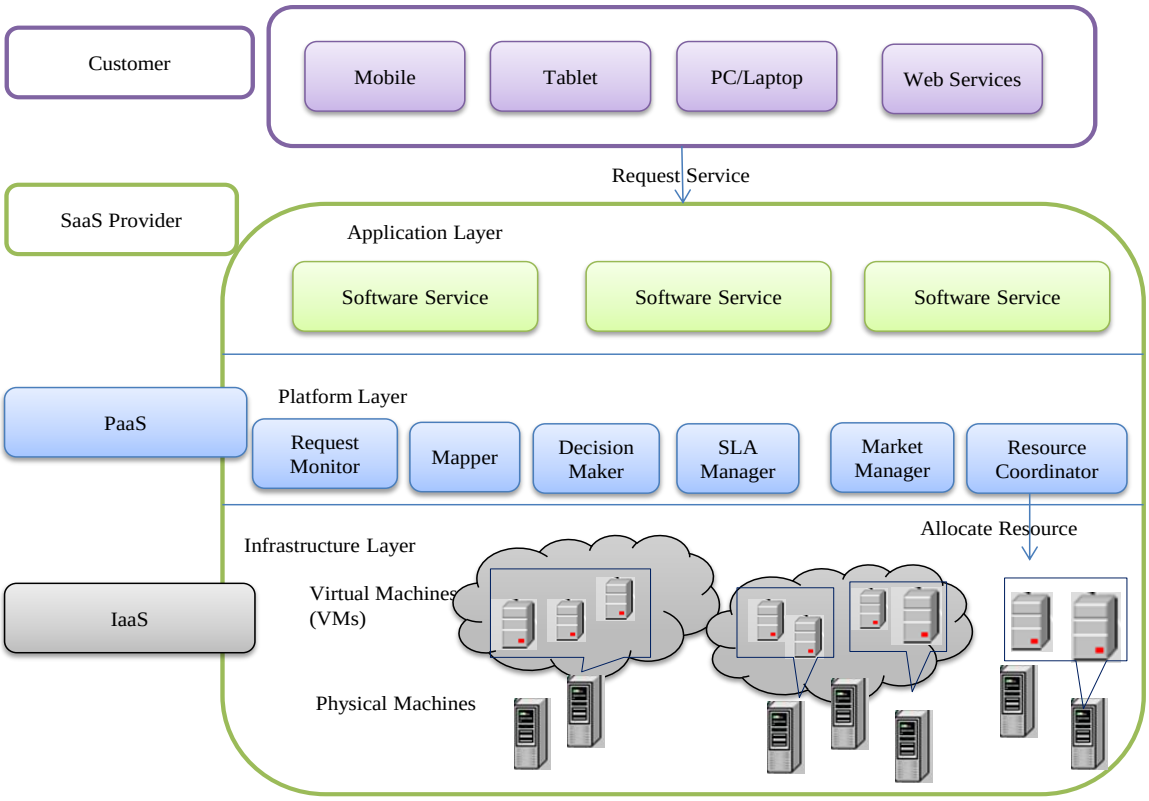


Figure 6.1 the SLA-based resource management system high level architecture

6.2.1 Details

In this section, we provide finer details related to fundamental classes of the SLA-based resource allocation system, which are also the building blocks of the system. The overall Class, Sequence, and States design diagrams are shown below.

Class Diagram

The main components of class diagram are described below:

- **(QoS) Request Monitor:** When a customer submits a new request or changes an existing request for the service, this class monitors changes and then invokes Mapper and DecisionMaker classes to reschedule the request.
- **Mapper**: This class maps customer QoS requirements to a suitable type of resource by method getVMTypebyServiceType(servType).
- **SLA Service Setting**: This class provides functions to access and operate the SaaS provider's predefined service characteristics. For example, getServiceResponseTime(servType) is used to retrieve the predefined service response time.

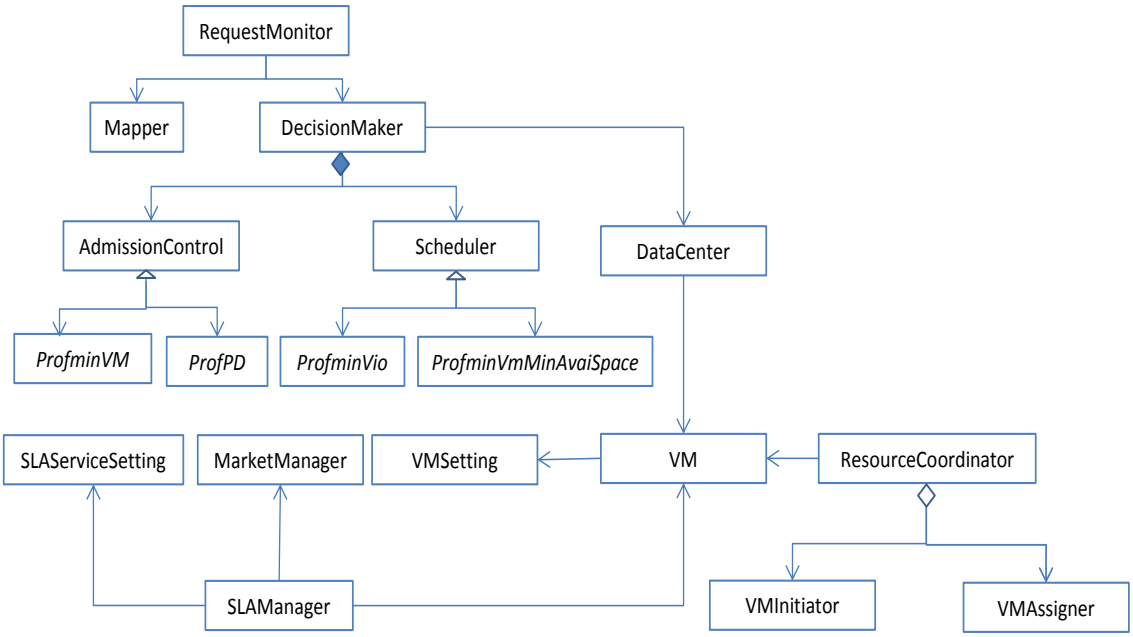


Figure 6.2 Class diagram

- **Decision Maker**: This class invokes the **admission control** and **scheduling** classes to make decision on whether to admit the customer request and how to assign resource to the customer.
    - **Admission Control:** This class is used to interpret and analyse customers' QoS requirements and receive the pre-scheduling result from scheduler, and then it uses admission control criteria to decide whether to accept or reject the request. The ProfminVM and ProfPD algorithms are proposed in Chapter **3**.
    - **Scheduler:** This class is responsible for pre-scheduling the request with scheduling strategies and returning where the request can be scheduled. The ProfminVio and ProfminVmMinAvaiSpace are algorithms proposed in Chapter **4**.
- **SLA Manager**: SLA Manager is the class that keeps track of SLAs fulfilment between customers and service providers. It also detects the penalty delay and updates the market manager.
- **Market Manager:** It is responsible for calculating and updating the cost and profit according to the actual resource usage. When there is a SLA violation, penalty cost is calculated and final profit is adjusted by the market manager.
- **Data Centre**: Characteristics and related functions of data centres are represented in this class.

- **VM**: This class represents actual VMs and includes their related data, such as VM initiation time.
- **VM Setting:** This class includes characteristics of VMs, which are average values based on history records.
- **Resource Coordinator**: This class assigns existing resource or initiating new resources for customer requests according to the decision. It includes **VM** Initiator, **VM** Assigner, **VM** Monitor, and **VM** Cleaner.
    - **VM Initiator**: It takes the responsibility of creating, deploying and configuring VMs using VM templates in an appropriate data centre.
    - **VM Assigner:** It is responsible for configuring software on the appropriate VM.

Sequence Diagram

Internal process among system entities: When the system receives a request from a customer, the *QoS request monitor* invokes the class *mapper*'s function called getVMTypebyServType(servType), which returns a suitable VM type. Following this, the *QoS request monitor* invokes the function *MakeDecision()* in class *DecisionMaker* to get decision whether this request can be accepted. Next, the *DecisionMaker* class invokes the function *AdmissionControlProcess()* in class *AdmissionControl*, which includes two stages: the first stage *AdmissionControlAnalysis()* calls the scheduler's SchedulingAnalysis() function, which checks current resource availability and capability using scheduling strategies and returns where the request can be scheduled. The second stage, *AdmissionControlDecisionMaking(),* checks if the request can be accepted regarding to the admission control criteria and returns the result to Decision Maker. Finally, the request monitor receives the decision.

Internal process on resource level: The *resource coordinator* detects the decision made by the decision maker. If the decision result is *accept* and scheduling result is *initiateNewVM*, then the request state goes into provisioning and resource coordinator calls the *initiateVM()* function in *VMInitiator* class to create and deploy a suitable VM image. If the scheduling result is *Wait* or *Insert*, then the resource coordinator calls the *assignRequest()* function in class *VMAssigner* to assign the request to an appropriate existing VM by configuring the software service. The status of the request becomes *inserting* or *waiting*. Following that, the *monitorVMIni()* function in class *VMMonitor* detects the actual VM initiation time and then updates the *VMinitiation* time by calling the *updateVM()* function in the class VM. When all

requests are finished on a VM, the *VMCleaner* invokes function *PowerOff()* to power off the VM.

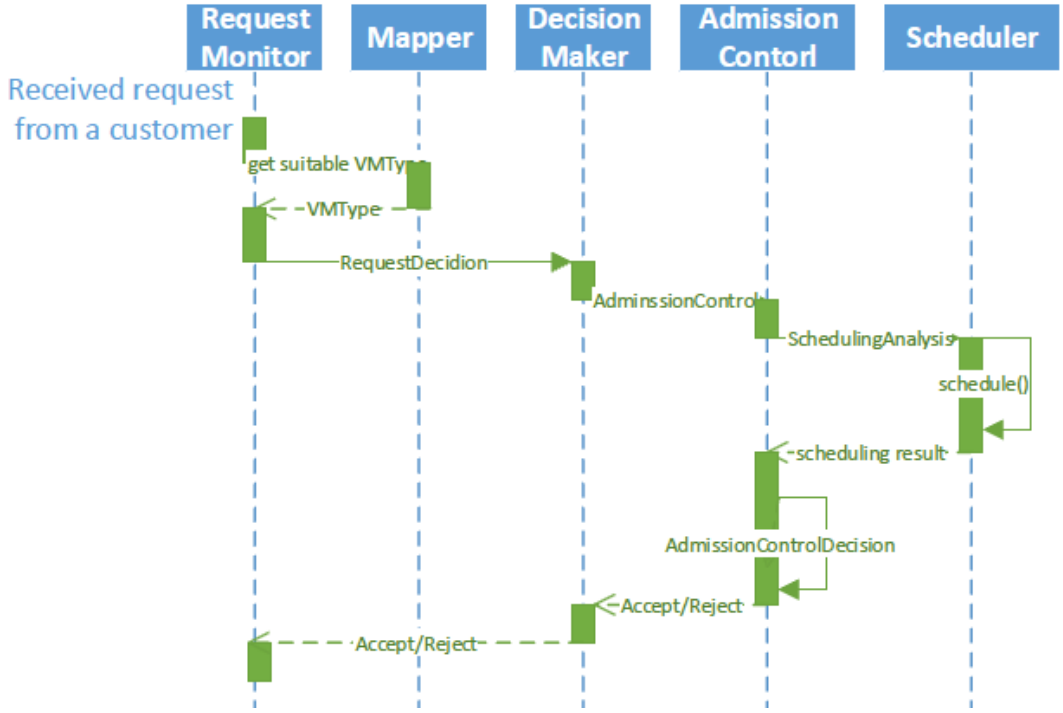


Figure 6.3 Sequence diagram among entities

States Diagram

Figure 6.5 illustrates diverse states that a customer QoS request can experience during its lifetime. When a request is submitted to the system, the new request goes to the *new* state and the upgrade request goes to the *upgrade* state. Both new and upgrade requests can go to the *rejected* state if a SaaS provider cannot gain the expected profit. If service start deadline is achievable with available resources, the request goes to the *inserting* state. If there is no resource available immediately but some existing requests will finish before the service start deadline, then the request goes to the *waiting* state. When the Scheduler detects that a new resource needs to be initiated for the request, either because there is no existing resource available before the service start deadline, but new resource can be initiated for the request, then the request goes to the *provisioning state*. For inserting, waiting and provisioning requests, after the request has been assigned to the VM, the states goes to the *running* state, which means a customer starts to use the service for enterprise software as a service or a task

starts to execute for bag of task service. Also, changes in state may happen every time a request contract expires and then the resource capability is recalculated.

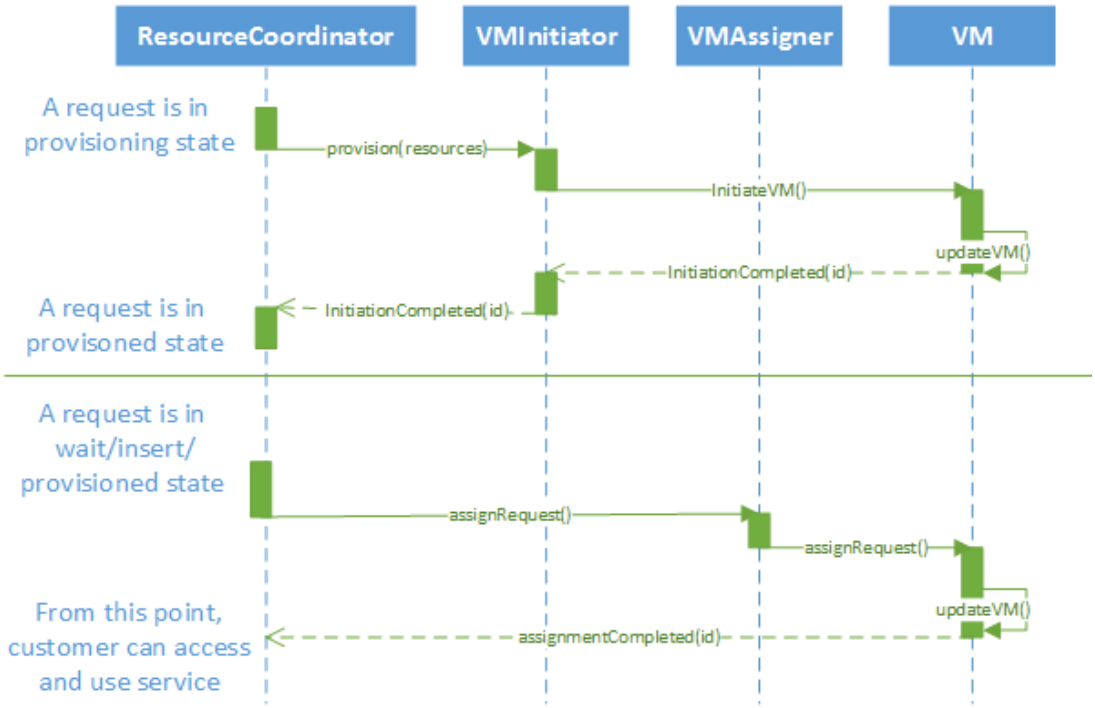


Figure 6.4 Sequence diagram among resource level entities

For both new and upgrade requests, the *finished* state is reached in three different situations: (i) contract expires; (ii) system failure; and (iii) the customer cancelled the request.

6.3 System Implementation Technologies

The SLARMS has been implemented by leveraging the following key technologies using C# on .Net platform: (1) SharePoint 2010, which is a secure, manageable, and web-based platform supporting application development. (2) PowerShell for creating, managing, and configuring VMs hosted on private and public cloud (such as Azure).

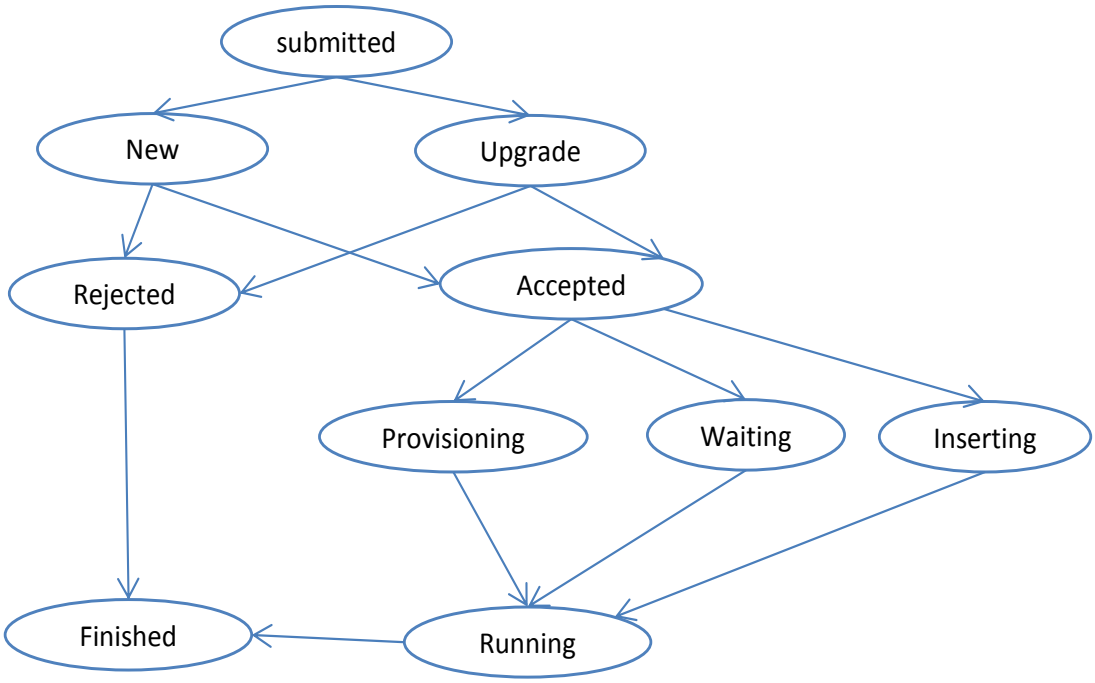


Figure 6.5 States diagram of requests in the SLARA system

6.3.1 Design Considerations

The design considerations of the SLARMS are the following:

- **Support for dynamic customer requests**: When there is a customer updating the request, the request monitor will be triggered to detect request changes and go through the decision making process.
- **Support for scalable infrastructure resources**: To allow easy utilization of using different types of Cloud infrastructures, SLARMS is designed to use C# in .Net platform to execute PowerShell command on remote VMs. PowerShell has been chosen because the most popular private VM infrastructure provision technology, VMWare, has a PowerShell based API (PowerCML). In addition, two of the most popular public infrastructure providers - Azure and Amazon, support PowerShell VM provision and configuration.
- **Fault tolerance**: SLARMS can handle failures at two stages: during decision making, and during resource provisioning. Failures during resource provisioning (initiation or allocation) can occur due to various reasons, such as network problems. In this case, the failed resource will be re-provisioned in the next resource allocation cycle.

- **Scalability**: Most of the SLARMS's components work independently and interact through a database, which facilitates the scalable implementation of SLARMS as each component can be distributed across different servers accessing a shared database.

6.3.2 Implementation Details

The implementation of each component is described as below:

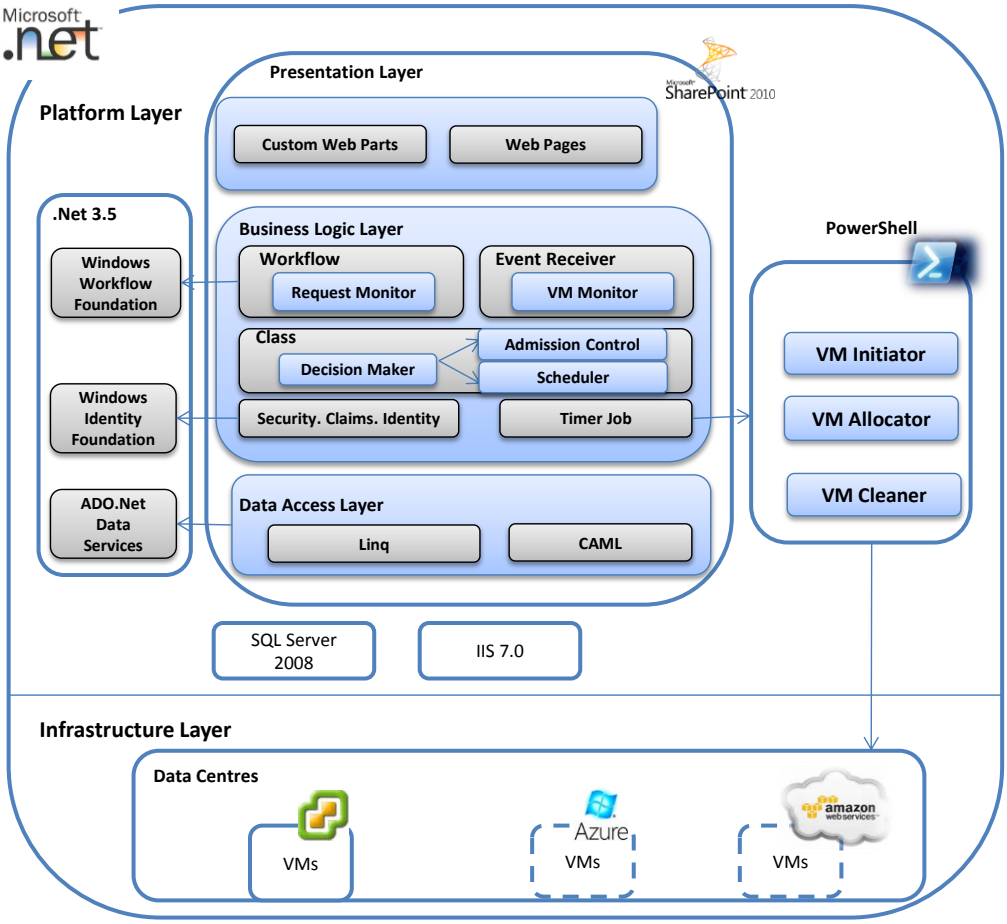


Figure 6.6 Implementation Technologies

The design followed the three layer design pattern containing data layer, business logic layer and presentation layer.

The main system entities are implemented using the following technologies:

- **Custom web parts and web pages**: In the presentation layer, custom web parts and web pages are used to provide an easy to use portal for customers to add or update their requests.
- **Workflow:** Workflow technology in SharePoint is used to implement **QoS Request Monitor**. The workflow can be triggered when there is a new request or any field of an existed request is updated. The background technology to support SharePoint workflow is the .**Net workflow foundation**.
- **Event Handler/Event Receiver:** SharePoint Event Handler/ Event Receiver technology is used to implement **VMMonitor**. Whenever there is any change happens on VM, such as actual VM initiation time is updated in the list, then the event handler will detect the change and invokes the SLA manager to calculate the penalty delay.
- **Class**: Standard C# classes are used to implement other components, such as main components decision maker, which includes admission control and scheduler.
- **Timer job**: SharePoint timer job is used to implement **VMCleaner**. The timer job runs every minute to detect if any VM does not have requests allocated and then the VM will be powered off in one hour.
- **PowerShell:** is used for most of resource coordinator related operations, such as VM initiation, because PowerCLI (based on PowerShell) is the easiest API to operate VMware Vsphere virtualized Cloud infrastructures (and for the extension of future work it is the common way to access Azure and Amazon EC2). In addition, for guest OS operation, the PowerShell is one of the most powerful technologies to configure the guest OS and install the software.
- **Linq and CAML (Collaborative Application Markup Language**): To implement the **Data Access Layer**, both Linq and CAML data access technologies are used because of some issues with Linq. For example, when disposing the data context, there is an error which is a known issue. Therefore, traditional CAML is used for insert operation and keep Linq for the rest data access operations.
- All data tables are presented using SharePoint **Column and List** technologies, which are more readable and friendly ways to structure the information, and all table structures and data are stored in **SQL Server 2008**.
- **Internet Information Services (IIS)**: IIS for Windows® Server is a flexible, secure, and manageable Web server for hosting anything on the Web. From media streaming to web applications, IIS's scalable and open architecture is ready to handle the most demanding tasks.

6.4 Case Study: CA (Computer Associates) Directory

This section describes how the SLARMS prototype is implemented using a private enterprise Cloud. This private Cloud is within an enterprise without affecting the productivity of their users, hence, it increases the amount of computing resources available within an enterprise to accelerate application performance.

6.4.1 System Details

Customer related

Customer: Request CA Directory services. This component is constituted by a simple Web Service client that generates all resource requests to SLARMS with the following QoS:

- **Request Type (reqType):** It defines the customer request type, which is 'new' or 'upgrade service'. A 'new' request will get one hour free service usage, while an 'upgrade service' is for an existing customer, who wants to upgrade from a lower service edition to an upper service edition (According to the customer usage, there maybe a customer loyalty level).
- **Product Type (*proType*)**: The software products offered to customers. It can be Standard, Silver, and Gold service. The Standard product includes CA Director. The Silver service package contains all functions of Standard plus JExplorer component. The Gold service includes all features of Silver plus dxgrid component.
- **Account # (accNum):** It constrains the maximum number of concurrent users from the same organization can use the software service.
- **Contract Length (conLen):** How long the software service is legally available for a customer to use (minimum is one hour).
- **Records storage (recNum):** The maximum storage capability for each DSA period and it will impact the data transfer time during the service upgrade (The value of this parameter is predefined in SLA).
- **Response Time (respTime):** It represents the elapsed time between the end of a demand on a software service and the beginning of a service. Violation occurs when actual elapsed time is longer than the pre-defined response time in the SLA.

SaaS provider related

Application Layer: provides CA Directory services. The CA directory provides a high-performance directory foundation for online applications. It allows customer organizations to

meet the needs of new and future dynamic business applications and improve operational efficiency by consolidating islands of data into a single information backbone.

Platform Layer: SharePoint 2010 platform and PowerShell are two main technologies used in this layer. SharePoint is used to implement most platform layer components except the resource allocator, which is implemented in PowerShell. The SharePoint platform and PowerShell scripts are integrated with C# language on .Net platform. Details are described in section 6.4.

Infrastructure Layer: In CA Lab, the internal operable Cloud infrastructure is built using VMware VSphere, which is the industry leading virtualization platform. This layer can be extended into public clouds.

The platform layer of a SaaS provider uses VM images to create instances according to the mapping (*Table 6.1*) and decision. (In *Table 6.1*, m is 5, n is 10). Therefore, it is important to identify the following properties for resource allocation mechanisms to ensure that the SLA is adequately drafted:

• **VM types (*l*)**: How many types of VM can be used and what they are. For example, there are three types of VM, which are large, medium, and small. The capacity of one large VM equals to that of two medium VMs or four small VMs.

• VM **Service Initiation Time (*iniTimeSev*)**: How long it takes to initiate a VM, which is deployed with the service appliance.

• **VM Price (*PriVM*)**: How much it costs to a SaaS provider for using a VM to serve the customer request per time unit. It includes the physical equipment, power, network, and administration price.

Table 6.1 Mapper Details

VM Type	Service	Account #	Storage	VM Price	CPU	Memory	Storage
Small	Standard	[0, m]	[0, n]	$0.085	1 core	1G	50G
Medium	Silver	[m+1,2m]	[n+1,2n]	$0.34	2 cores	2G	50G
Large	Gold	[2m+1,5m]	[2n+1,5n]	$0.68	4 cores	4G	50G

Decision Making

The decision making process includes two main components: scheduling and admission control, in this case study, we implemented the algorithms introduced in Chapters 3 and 4.

6.5 Performance Evaluation

6.5.1 Experiment Setup

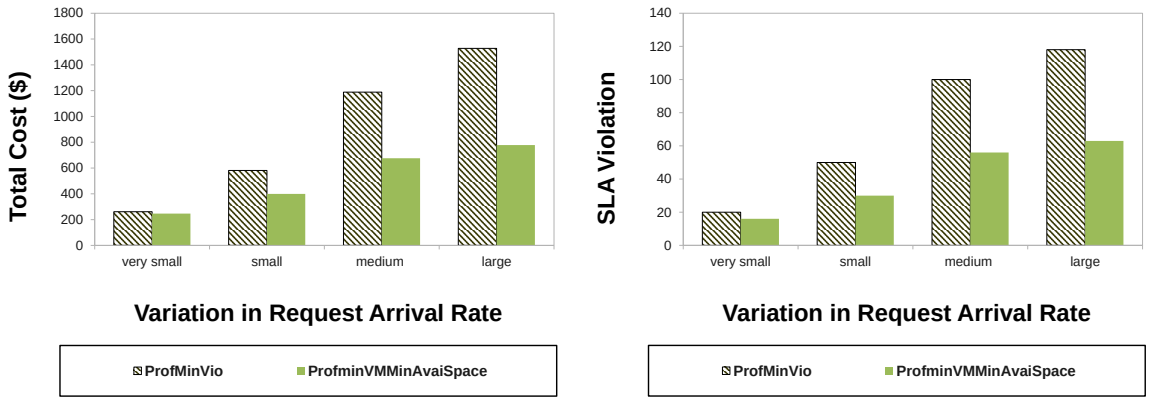
The evaluation of mechanisms of SLARMS, described in the previous section, has been carried out entirely in CA Lab VMware Vsphere Cloud infrastructure environment.

The experimental setup consists of three types of dynamic resources: small instance (1 GB of memory, 1 CPU core, 50G of local instance storage, Windows OS); medium instance (2 GB of memory, 2 CPU core, 50G of local instance storage, Windows OS); and large instance (4 GB of memory, 4 CPU core, 50G of local instance storage, Windows OS). An enterprise application CA directory is used for experiments. SLA is defined in terms of response times. The experiment evaluation is designed based on the CA CloudMinder test strategy and plan. CloudMinder is an online application that uses CA Directory as the directory foundation. In this set of experiments the total profit, number of accepted users and number of SLA violations are evaluated as follows during the variation of request arrival rate from 20 to 200 requests per second. Up to 200 concurrent user requests are considered because 1) The test strategy provided by CA is designed using 200 user requests, which has been analysed through their customer usage data and 2) The capability of the private data centre allocated to this research work is limited, which does not allow a very large number of user requests.

6.5.2 Scheduling algorithms evaluate

The evaluation is designed to test our proposed algorithms in Chapter 3 and 4. As expected, the algorithms perform the similar trend as the simulation results in the prototype implementation environment. In this set of experiment the total cost, SLA violations are evaluated in this section during the variation of request upgrade proportion varies from very low to very high.

It can be seen from *Figure 6.7*, in average the algorithm *ProfminVMminAvaiSpace* reduces about 50% cost compared with *ProfminVio*. As *Figure 6.7b* shows, during the arrival rate variation, the number of SLA violations caused by *ProfminVMminAvaiSpace* is less than the *ProfminVio* because the *ProfminVio* has more risk to cause VM initiation delay due to network-related issues. Therefore, during the variation of arrival rate, the *ProfminVMminAvaiSpace* performs better and minimize the SLA violations in the context of resource sharing, where it is impossible to avoid SLA violations.



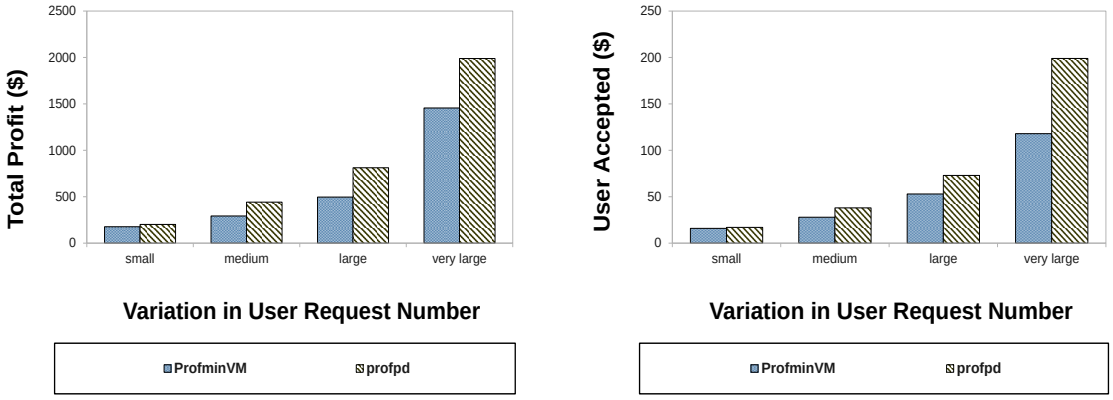
(a) Total cost($) (b) SLA violation

Figure 6.7 Variation in Request Arrival Rate

6.5.3 Admission control algorithms evaluate

The evaluation is designed to test our proposed algorithms in Chapters 3 and 4. The evaluation results show that the algorithms performs similar trend in the prototype environment. In this set of experiment total profit and number of accepted users are evaluated during the variation of user request number from 10(small) to 100(very large).

Figure 6.8 shows that the ProfPD achieves (17%) more profit over ProfminVM by accepting (15%) more user requests, when number of users changes from "small" to "very large". When the number of users is increased from "medium" to "large", the profit difference between ProfPD and ProfminVM became larger. This is because when the number of requests increases, the number of users being accepted increases by utilizing initiated VMs. Therefore, a SaaS provider should use ProfPD to maximize profit.



(a). Total profit($) (b). User accepted($)

Figure 6.8 Variation in User Request Number

6.6 Related Work

There are several previous approaches for resource management with respect to SLA. Control-theory approach has been proposed to dynamically adjust resource allocation to maintain the service differentiation [172]. CPU cycles of single servers are main concerns of other approaches, which share resources among multiple customer requests or applications [168][170]. For example, the *Shift* adjusts how much and when CPU resources should be allocated to a VM [173]. In contrast, SLARMS focuses on sharing at the granularity of whole VMs and the management of a whole farm of servers. *IcorpMaker* provides isolation via virtual private networks rather than VM[169]. *Océano* attempts to modify the computing environment (e.g. by installing an operating system) to satisfy the allocation. Finally, the Galaxy project [171] focuses on providing tools to build Windows-NT clusters. It does not consider SLA monitoring. SLARMS provides a unique and more comprehensive combination of technologies to address a number of issues ignored by these approaches and focused on SLA-based customer requirement driven resource provisioning.

6.7 Summary

To meet requirements of SLA-based resource management of Cloud services (in Chapter 1), this chapter focused on the design, development, and implementation of a software systems based on novel SLA-based resource management algorithms exclusively designed for SaaS. Through this prototype implementation, called SLA-based Resource Management System (SLARMS), we also demonstrated the usefulness of key algorithms and techniques proposed in this thesis. The architecture and implementation of SLARMS is comprehensively described and evaluated. Two sets of experiments performed to test algorithms proposed in Chapter 3 and 4. In the experiments, the total cost, SLA violations were evaluated and the experimental results show trend similar to simulation results.

7 Conclusions and Future Directions

This chapter summarizes our objectives and work carried out on this thesis. Our main findings and lessons learned are discussed along with their significance. This chapter also concludes with a discussion on the future research direction that emerged during this research.

7.1 Summary

Cloud computing is a solution for addressing challenges such as licensing, distribution, configuration, and operation of enterprise applications associated with the traditional IT infrastructure, software sales and deployment models. Migrating from a traditional model to the Cloud model reduces the maintenance complexity and cost for enterprise customers, and provides on-going revenue for Software as a Service (SaaS) providers. Clients and SaaS providers need to establish a Service Level Agreement (SLA) to define the Quality of Service (QoS). The main objectives of SaaS providers are to optimize resource provisioning for maximizingthe utilization of underline Cloud system in order to maximize profit and enlarge market share.

To achieve these objectives, the thesis started with a comprehensive survey on SLA and their creation, management and usage in utility computing environments. It discussed existing use cases in Grid and Cloud computing environments to identify the level of SLA realization in state-of-art systems and emerging challenges for future research. The survey identified that most works manage resources with the aim of minimizing cost without sufficiently considering the customers' needs. Thus, to achieve the SaaS providers' objectives, SaaS providers can manage resources in a way to 1) accept more profitable requests with guaranteed QoS and 2) improve the QoS for customers, since in general it costs much more to attract new customers than it does to keep an existing one.

There are several challenges to achieve the objectives in SLA-based resource provisioning for management of Cloud-based software as service applications for SaaS providers. First, the SaaS provider uses shared infrastructure and different types of requests loads that lead to dynamic variation in availability and capacity of resources. Second, there is possibility for existing customers to change their requirements, such as upgrade services, which may lead to resource reallocation. Third, the SaaS provider requires flexible SLA establishment processes to cater for individual customers and considering market competition among other providers. Therefore, three sub objectives of thesis are identified to overcome these challenges:

- To design SLA-based admission control and scheduling algorithms that differentiate customer requests based on the dynamic resource performance and capabilities to minimize cost and SLA violations by accepting more profitable requests.
- To investigate adaptive SLA-based resource provisioning algorithms according to the changes in customer requirements by considering more customer factors that provide personalized attention to customers and understanding their specific needs.
- To investigate the architectural model for automated SLA negotiation to establish SLAs between SaaS and customers, whose requirements are not covered by existing SaaS predefined static SLAs.

The first objective is achieved in Chapter 3, which proposed innovative cost-effective admission control and scheduling algorithms. Our proposed solutions are able to accept more profitable requests and minimize SLA violations through the efficient placement of requests on VMs leased from multiple IaaS providers. Our solution takes into account various customer's QoS requirements (such as deadline, budget, penalty rate) and infrastructure heterogeneity (such as different types of VM, service initiation time and price). Simulation results showed that our proposed algorithms provide substantial improvement (up to 40% cost saving) over reference ones across all ranges of variation in QoS parameters.

The thesis accomplished the second objective in Chapter 4 by designing customer requirements driven resource provisioning algorithms to maximize profit by minimizing resource and penalty cost. These algorithms also improve CSL by SLA violations minimization and service quality improvement. These algorithms consider customer profiles and providers' quality parameters (e.g. response time) to handle customer requirements changes and infrastructure level heterogeneity for enterprise systems. Customer-side parameters (such as the proportion of upgrade requests), and infrastructure-level parameters (such as the service initiation time) are considered to compare proposed algorithms. Simulation results showed that the proposed algorithms reduce the total

cost up to 54% and the number of SLA violations up to 45%, compared with the previously proposed best algorithm.

In order to enlarge customer base, SaaS providers have to attract customers with special requirements. Chapter 5 proposed a novel automated negotiation framework to establish SLAs with these special QoS requirements. The framework also considers the SaaS Broker as the one-stop-shop for customers to efficiently secure the required services. The framework also included negotiation policies, protocols, and strategies to perform adaptive and intelligent bilateral bargaining of SLAs between the SaaS provider and the SaaS broker. It designed decision-making heuristics considering time, market constraints, and trade-off between different issues. These negotiation heuristics are evaluated by extensive experimental study of our prototype framework using data from a real Cloud.

Chapter 6 introduced a prototype of SLA-based resource management system, which is implemented to prove the usefulness of the proposed algorithms using real Cloud resources.

7.2 Lessons Learned and Significance

Chapter 2 contained a comprehensive survey of how SLAs are created, managed and used with use cases in both academy and industry with major emphasis on the SLA-based resource management systems. This survey not only assists researchers to understand primary design factors and issues that are still outstanding and crucial, but also provides insights for extending and reusing components of existing Resource Management Systems (RMSs). Therefore, the survey can help in the design and implementation of more practical and enhanced SLA-based resource management systems in Cloud.

The admission control and scheduling algorithms proposed in Chapter 3 can be used by SaaS providers like Animoto. All proposed algorithms in this thesis can be used by SaaS providers who rent 3rd party resources or/and use in house hosted resources. Resources we considered are VMs, which are hosted in physical data centres. SaaS providers may not have knowledge of the configuration and capabilities of these physical resources. Moreover, SaaS providers are sharing physical resources with other SaaS providers, whose software usage and requests arrival rate may impact the performance of hosted software service. Proposed algorithms assist in identifying which request is more profitable and should be accepted and reduce the probability of SLA violations given the dynamic nature of Cloud resources.

Once a request is accepted by the SaaS provider, there is a possibility for customers to change their existing requirements (such as add more accounts or upgrade service package). Thus, SaaS is expected to be scaled up and out dynamically according to the customers' QoS requirements. When the customer changes his/her requirements, the SaaS provider has to dynamically relocate resources on-demand. Moreover, while allocating/reallocating resources, the SaaS provider has to minimize impact on existing customers while satisfying change requests. Customer requirements driven resource provisioning algorithms proposed in Chapter 4 can help in adapting to changes in the requirements. It takes into account more customer factors that provide personalized attention to the customer (such as customer profiles) and also is able to understand specific needs of customers by taking into consideration the customer's historical data.

These algorithms can be used by any enterprise software as service providers, who lease packaged enterprise software to customers with a fixed price. In addition, the scenario can also be applied to High Performance Computing and scientific applications by mapping VM capabilities and QoS requirements. The upgrade service package scenario may not be required by them, which simplifies the scenario compared to enterprise applications. Therefore algorithms and techniques proposed in this thesis can be applied to a wide range of applications from many domains.

As SaaS providers want to enlarge market share, they need to provide more flexibility in terms of services to cater to variations associated with an individual customer. This is generally done by a negotiation process between customers and service providers. However, while undertaking this negotiation process, the service provider needs to take into consideration not only what they can provide to customers but also the competition with other SaaS providers. Thus, Chapter 5 proposed that new negotiation frameworks are needed for the SaaS provider that considers dynamism in Cloud environment with time and market factors to make the best possible decisions for negotiation. The proposed negotiation framework can be used for the SaaS provider and the SaaS broker model.

To prove the usefulness of our proposed strategies, in Chapter 6, a prototype of the customer requirements driven SLA-based resource management system is implemented taking care of the changes in customer requirements and resource side heterogeneity using SharePoint platform and .NET technologies. The resource used in this prototype is a private Cloud, hosted by

Computer Associates, who is a Cloud software solution provider. The case study used CA Directory as a service because of the availability of the software. However, SaaS providers can offer any software as a service using our algorithms accordingly. This prototype can be plugged in with different resource management strategies to achieve different objectives. SaaS providers can scale out to use multiple resource providers including 3rd party resource providers with different resource APIs.

7.3 Future Directions

We have carried out detailed investigations in SLA-based resource provisioning for management of Cloud-based SaaS applications using dynamic resources in Cloud to maximize profit and market share for SaaS providers. However, there are still open issues that can serve as a starting point for future research.

7.3.1 Providing Services with Different Pricing Models

SaaS providers can design different dynamic pricing policies to maximize profit and increase market. For example, when customers buy laptops, there is self-service way for customers to customize their machines by paying different price according to the hardware configurations. SaaS providers can employ the similar functionality allows self-service feature for customers to customize software packages according to their needs in a more flexible and profitable way for SLA-based resource management strategies to achieve their objectives. Therefore, to design the resource management strategies, it is required to understand 1) what software components can be offered; 2) how resources consumption will vary during the variation of these components; and 3) how to design the price policy among the variation of these components.

7.3.2 Using Resources with Different Pricing Models

The SaaS provider can use resources with various price policies to satisfy customer requirements and reduce costs. For example, Amazon [13] has two types of pricing models; a) fixed pricing and b) spot pricing. Each of these models gives some advantages and disadvantages to consumers. For instance, the spot pricing can be exploited to maximise the consumer's profit but it reduces the chances of requests being executed successfully. In such environments, not only the current but also future status of resources needs to be considered to reduce the consumer's violation of SLA and spending. Hence, there is a need to understand the effect of using different pricing models on SLA-based resource management, and design novel resource management strategies to handle such varieties.

7.3.3 Resource Provisioning for Multi-tier Applications

In our scenarios, various kinds of applications are considered as a standalone package including the application and data. However, there are some enterprises using combination of single tier and multiple-tier structures for applications. For example, enterprises host both SharePoint and SQL-Server database in a single VM for development and testing environments, whereas host SharePoint and SQL-Server database in different VMs for staging and production environments. Therefore, the exploration of resource provisioning for multi-tier applications is a critical topic in the future.

7.3.4 Resource Provisioning for Network and Data-Aware Application

In Clouds, there are several applications that require petabytes of data from various repositories distributed across various nations. The resource provisioning process for these applications competing for compute and storage resources can be very challenging due to the highly dynamic nature of network. Moreover, computation should be ideally located near to storage, thus decreasing the delays in the execution. If the scheduling decisions are made just on the basis of either data size or computation time, the resultant schedule can lead to resource wastage in terms of network bandwidth, and performance degradation due to large execution delays. Therefore, approaches that consider both monetary execution costs and reconcile the competing storage, network and computation demand of users are required.

7.3.5 Customer Usage Model for Customer Driven Resource Management

We proposed user profile and using history-based method for predicting the transaction-based enterprise system usage to calculate the credit level. However, resource usage patterns and usage prediction is actually another area that has been studied intensely. The future research could explore more sophisticated credit level calculation based on the usage pattern and usage prediction technologies for SLA-based resource management strategies.

References

[1] Kleinrock, L. A. (2005). Vision for the Internet. ST Journal of Research, 2(1), (pp. 4-5).

[2] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. Future Generation Computer Systems, 25(6), (pp. 599-616).

[3] Buyya, R., and Vazhkudai, S. (2001). Compute Power Market: Towards a Market-Oriented Grid. Proceedings of the 1st International Symposium on Cluster Computing and the Grid, Brisbane, Australia.

[4] Buyya, R., Broberg, J., and Goscinski, A. (eds). (2011). Cloud Computing: Principles and Paradigms. ISBN-13: 978-0470887998, Wiley Press, USA, Feb. 2011.

[5] Patterson, D. A. (2008). The Data Center Is The Computer. Communications of the ACM, (pp. 105). NY, USA.

[6] Buyya, R., Garg, S. K., and Calheiros, R. N. (2011). SLA-Oriented Resource Provisioning for Cloud Computing: Challenges, Architecture, and Solutions. Proceedings of the 2011 IEEE International Conference on Cloud and Service Computing (CSC 2011, IEEE Press, USA), Hong Kong, China, Dec. 12-14, 2011.

[7] Broberg, J., Venugopal, S., and Buyya, R. (2008). Market-Oriented Grids and Utility Computing: The State-of-the-Art and Future Directions. Journal of Grid Computing, 6(3), (pp. 255-276).

[8] Hardin. G. (1968). The Tragedy of the Commons. Science, 162(3859), (pp. 1243-1248).

[9] Yeo, C. S., and Buyya, R. (2007). Integrated Risk Analysis for a Commercial Computing Service. Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), (pp. 51). CA, USA.

[10] Schneider, B., and White, S. S. (2004). Service Quality: Research Perspectives. Sage Publications, Thousand Oaks, CA, USA.

[11] Van Looy, B., Gemmel, P., and Van Dierdonck, R., editors. (2003). Services Management: An Integrated Approach. Financial Times Prentice Hall, Harlow, England, second edition, 2003.

[12] Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., and Xu, M. (2007). Web Services Agreement Specification. OGF Proposed Recommendation (GFD.107).

[13] AWS EC2 Service Level Agreement. Retrieved 28 March 2010, from AWS: http://aws.amazon.com/ec2-sla/.

[14] AWS S3 Service Level Agreement. (n.d.). Retrieved 28 March 2010, from AWS: http://aws.amazon.com/s3-sla/.

[15] Battre', D., Hovestadt, M., Kao, O., Keller, A., and Voss, K. (2007). Planning-based Scheduling for SLA-Awareness and Grid Integration. PlanSIG, (pp. 1).

[16] Blythe, J., Deelman, E., and Gil, Y. (2004). Automatically Composed Workflows for Grid Environments. IEEE Intelligent Systems, (pp. 16-23).

[17] Bonell, M. (1996). The UNIDROIT Principles of International Commercial Contracts and the Principles of European Contract Law: Similar Rules for the Same Purpose. Uniform Law Review, (pp. 229-246).

[18] Boniface, M., Phillips, S., Sanchez-Macian, A., and Surridge, M. (2009). Dynamic Service Provisioning Using GRIA SLAs. Service-Oriented Computing-ICSOC 2007 Workshops, (pp. 56-67). Vienna, Austria.

[19] Brandic, I., Venugopa S., Mattess, M., and Buyya, R. (2008). Towards a Meta-negotiation Architecture for SLA-aware Grid Services. International Workshop on Service-Oriented Engineering and Optimization, (pp. 17). Bangalore, India.

[20] Brandic, I., Music, D., and Dustdar, S. (2009). Service Mediation and Negotiation Bootstrapping as First Achievements towards Self-adaptable Grid and Cloud Services. In Grids and Service-Oriented Architectures for Service Level Agreements. P. Wieder, R. Yahyapour, and W. Ziegler (eds.), Springer, New York, USA.

[21] Buco, M. J., Chang, R. N., Luan, L. Z., Ward, C., Wolf, J. L., and Yu, P. S. (2004). Utility Computing SLA Management based upon Business Objectives. IBM Systems Journal, 43(1), (pp. 159-178).

[22] Buyya, R., and Alexida. D. (2001). A Case for Economy Grid Architecture for Service Oriented Grid Computing. Proceedings of the 10th International Heterogeneous Computing Workshop (HCW), San Francisco, CA.

[23] Buyya, R., Pandey, S., and Vecchiola, C. (2009). Cloudbus Toolkit for Market-Oriented Cloud Computing. Proceedings of the 1st International Conference on Cloud Computing (CloudCom 2009, Springer, Germany), Beijing, China.

[24] Buyya, R., Ranjan, R., and Calheiros R. N. (2009). Modelling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities. Proceedings of the 7th High Performance Computing and Simulation Conference (HPCS 2009, ISBN: 978-1-4244-4907-1, IEEE Press, New York, USA), Leipzig, Germany.

[25] Chu, X., Nadiminti, K., Jin, Ch., Venugopal, S., and Buyya, R. (2002). Aneka: Next-Generation Enterprise Grid Platform for E-Science and E-Business Applications. Proceedings of the 3rd IEEE International Conference on E-Science and Grid Computing, (pp. 10-13). Bangalore, India.

[26] Dan, A., Ludwig, H., and Kearney, R. (2004). CREMONA: An Architecture and Library for Creation and Monitoring of WS-Agreements. Proceedings of the 2nd International Conference on Service-Oriented Computing, (pp. 65-74). NY, USA.

[27] Dinesh, V. (2004). Supporting Service Level Agreements on IP Networks. Proceedings of IEEE/IFIP Network Operations and Management Symposium, 92(9), (pp. 1382-1388). NY, USA.

[28] Fitzgerald, S., Foster, I., and Kesselman, C. (1997). A Directory Service for Configuring High-performance Distributed Computations. Proceedings of the 6th IEEE Symposium on High-performance Distributed Computing. (pp. 365-375).

[29] Foster, A. K. (2003). The Grid 2: Blueprint for a New Computing Infrastructure. San Francisco, CA: Morgan Kaufmann.

[30] Frey, N. (2000). A Guide to Successful SLA Development and Management. Stamford, CT: Gartner Group Research, Strategic Analysis Report.

[31] Frolund, S., and Koistinen, J. O. (1998). A Language for Quality of Service Specification. HP Labs Technical Report, California, USA.

[32] Gong, Y. L., Dong, F. P., Li, W., and Xu, Z. W. (2003). VEGA Infrastructure for Resource Discovery in Grids. Journal of Computer Science and Technology, 18(4), (pp. 413-422).

[33] Hiles, A. (1999/2000). The Complete IT Guide to Service Level Agreements-Matching Service Quality of Business Needs. Oxford, UK: Elsevier Advanced Technology.

[34] Hudert, S., Wirtz, G., and Eymann, T. (2009). BabelNeg-A Protocol Description Language for Automated SLA Negotiations. Proceedings of the IEEE Conference on Commerce and Enterprise Computing, (pp. 162-169). ShangHai, China.

[35] Iamnitchi, A., and Foster, I. (2001). On Fully Decentralized Resource Discovery in Grid Environments. Proceedings of the 2nd International Workshop on Grid Computing, (pp. 51-62). Denver, Colorado.

[36] Jin, L. J., and Machiraju, V. A. (June 2002). Analysis on Service Level Agreement of Web Services. Technical Report HPL-2002-180, Software Technology Laboratories, HP Laboratories.

[37] Joita, L., Rana, O. F., Chacn, P., Chao, I., and Ardaiz, O. (2005). Application Deployment Using Catallactic Grid Middleware. Proceedings of the 3rd International Workshop on Middleware for Grid Computing. (pp. 6). Grenoble, France.

[38] Karaenke, P., and Kirn, St. (2010). Towards Model Checking and Simulation of a Multi-Tier Negotiation Protocol for Service Chains. Proceedings of the 9th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010.

[39] Keller, A., Kar, G., Ludwig, H., Dan, A., and Hellerstein, J. L. (2002). Managing Dynamic Services: A Contract based Approach to a Conceptual Architecture. Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium, (pp. 513-528). Florence, Italy, April 15-19, 2002.

[40] Keller, A., and Ludwig, H. (2003). The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Network and Systems Management Special Limitation on E-Business Management, 11(1), (pp. 57-81). USA.

[41] Kuo, D., Parkin, M., and Brooke, J. (2006). A Framework and Negotiation Protocol for Service Contract. Proceedings of the 2006 IEEE International Conference on Services Computing (SCC 2006), (pp. 253-256). Chicago, USA.

[42] Lee, Y. C., Wang, C., Zomaya, A. Y., and Zhou B. B. (2010). Profit-driven Service Request Scheduling in Clouds. Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID). Melbourne, Australia.

[43] Loyall, J. P., Schantz, R. E., Zinky, J. A., and Bakken, D. E. (1998). Specifying and Measuring Quality of Service in Distributed Object Systems. Proceedings of the 1st International Symposium on Object Oriented Real-Time Distributed Computing, (pp. 43-54). Kyoto, Japan.

[44] Ludwig, A., and Franczyk, B. (2006). SLA Lifecycle Management in Services Grid-requirements and Current Efforts Analysis. Proceedings of the 4th International Conference on Grid Services Engineering and Management (GSEM), (pp. 219-246). LeipZig, Germany.

[45] Marilly, E., Martinot, O., Papini, H., and Goderis, D. (2002). Service Level Agreements: A Main Challenge for Next Generation Networks. Proceedings of the 2nd European Conference on Universal Multiservice Networks, (pp. 297-304). Toulouse, France.

[46] Mobach, D. G. A., Overeinder, B. J., and Brazier, F. M. T. (2006). A WS-Agreement based Resource Negotiation Framework for Mobile Agents. Scalable Computing: Practice and Experience, 7(1), (pp. 23-26). March 2006.

[47] Philipp, W., Jan, S., Oliver, Z., Wolfgang, Z., and Ramin, Y. (2005). Using SLA for Resource Management and Scheduling. Grid Middleware and Services-Challenges and Solutions, 8(1), (pp.281-291).

[48] Rana, O. F., Warnier, M., Quillinan, T. B., Brazier, F., and Cojocarasu, D. (2008). Managing Violations in Service Level Agreements. Proceedings of the 5th International Workshop on Grid Economics and Business Models (GenCon), (pp. 349-358). Gran Canaris, Spain.

[49] Rashid, A. A., Hafid, A., Rana, A., and Walker, D. (2004). An Approach for Quality of Service Adaptation in Service-oriented Grids. Concurrency and Computation: Practice and Experience, 16(819), (pp.401-412).

[50] Rick, L. (2002). IT Services Management Description of Service Level Agreements. RL Consulting.

[51] Ron, S., and Aliko, P. (2001). Service Level Agreements. Internet NG. Internet NG project (1999-2001) http://ing.ctit.utwente.nl/WU2/

[52] Sahai, A., Graupner, S., Machiraju, V., and Van Moorsel, A. (2003). Specifying and Monitoring Guarantees in Commercial Grids through SLA. Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid, (pp. 292). Tokyo, Japan.

[53] Sakellariou, R., and Yarmolenko, V. (2005). On the Fexibility of WS-Agreement for Job Submission. Proceedings of the 3rd International Workshop on Middleware for Grid Computing (MGC05), (pp. 6). Grenoble, France.

[54] Service Level Agreement in the Data Centre. (April 2002). Retrieved 28 March 2010, from Sun Microsystems: http://www.sun.com/blueprints.

[55] Skene, J., Lamanna, D. D., and Emmerich, W. (2004). Precise Service Level Agreements. Proceedings of the 26th International Conference on Software Engineering (ICSE'04), (pp. 179-188). Bugzilla. May 23-28, 2004.

[56] Venugopal, S., Chu, X., and Buyya, R. A Negotiation Mechanism for Advance Resource Reservation Using the Alternate Offers Protocol. Proceedings of the 16th International Workshop on Quality of Service (IWQoS 2008, IEEE Communications Society Press, New York, USA), June 2-4, 2008, Twente, The Netherlands.

[57] Rosenberg, I., and Juan, A. (2009). The BEinGRID SLA framework, Report available at http://www. gridipedia. eu/slawhitechapter.html

[58] Tosic, V., Pagurek, B., Patel, K., Esfandiari, B., and Ma, W. (2005). Management Applications of the Web Service Offerings Language (wsol). Web Services, E-Business, and the Semantic Web, (pp.564-586). Galway, Ireland.

[59] Wieder, P., Seidel, J., Yahyapour, R., Waldrich, O., and Ziegler, W. (2008). Using SLA for Resource Management and Schedurling-A Survey. GRID Middleware and Services, 4, (pp. 335-347).

[60] Windows Azure Service Level Agreement. Retrieved 28 March 2010, from http://www.microsoft.com/windowsazure/sla/.

[61] Wurman, P. R., Wellman, M. P., and Walsh, W. E. (1998). The Michigan Internet Auctionbot: A Configurable Auction Server for Human and Software Agents. Proceedings

of the 2nd International Conference on Autonomous Agents, (pp.301-308). Irsee, Germany.

[62] Yeo, C. S., and Buyya, R. (2006). A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. Software: Practice and Experience (SPE), 36 (13), (pp.1381-1419). Jan. 2006.

[63] Yeo, C. S., DeAssuncao, M. D., Yu, J., Sulistio, A., Venugopal, S., Placek, M., and Buyya, R. (2006). Utility Computing on Global Grids. In H. Bidgoli (Ed), Handbook of Computer Networks. ISBN: 978-0-471-78461-6, John Wiley and Sons, New York, USA.

[64] Yeo, C. S., and Buyya, R. (2007). Pricing for Utility-driven Resource Management and Allocation in Clusters. International Journal of High Performance Computing Applications, 21(4):405-418. Nov. 2007.

[65] Yeo, C. S., and Buyya, R. (2005). Service Level Agreement based Allocation of Cluster Resources: Handling Penalty to Enhance Utility. Proceedings of the 7th IEEE International Conference on Cluster Computing (Cluster 2005), (pp. 1-10). Bostan, MA, USA.

[66] Youseff, L., Butrico, M., and Da Silva, D. (2008). Toward a Unified Ontology of Cloud Computing. Grid Computing Environments Workshop, (pp.1-10). Austin, Texas, USA.

[67] Jaideep, D. N., and Varma, M. V. (2010). Learning based Opportunistic Admission Control Algorithms for Map Reduce as a Service. Proceedings of the 3rd India Software Engineering Conference (ISEC 2010), Mysore, India.

[68] Irwin, D. E. and Grit, L. E. and Chase, J. S. (2004). Balancing Risk and Reward in a Market-based Task Service. Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC 2004), Honolulu, HI, USA.

[69] Yemini, Y. (1981). Selfish Optimization in Computer Networks Processing. In Proceeding of the 20th IEEE Conference on Decision and Control including the Symposium on Adaptive Processes, San Diego, USA.

[70] Popovici, I., and Wiles, J. (2005). Profitable Services in an Uncertain World. Proceedings of the 18th Conference on Supercomputing (SC 2005), Seattle, WA.

[71] Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. (2009). A Break in the Clouds: towards a Cloud Definition. ACM SIGCOMM Computer Communication Review, 39(1), (pp.50-55).

[72] Parkhill, D. (1966). The Challenge of the Computer Utility, Addison-Wesley, USA.

[73] Vouk, M. A. (2008). Cloud Computing-Issues, Research and Implementation. Proceedings of the 30th International Conference on Information Technology Interfaces (ITI 2008), Dubrovnik, Croatia.

[74] Bichler, M., and Setzer, T. (2007). Admission Control for Media on Demand Services. Service Oriented Computing and Application. Proceedings of IEEE International Conference on Service Oriented Computing and Applications (SOCA 2007), Newport Beach, California, USA.

[75] Chun, N. B., and Culler, D. E. (2002). User-centric Performance Analysis of Market-based Cluster Batch Schedulers. Proceedings of the 2nd IEEE/ACM International Symposium on Cluster and Grid Computing (CCGrid 2002), Berlin, Germany.

[76] Coleman, K., Norris, J., Candea, G., and Fox, A. (2004). OnCall: Defeating Spikes with a Free-market Application Cluster. Proceedings of the 1st International Conference on Autonomic Computing, New York, USA.

[77] Buyya, R., Ranjan, R., and Calheiros, R. N. (2010). InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2010), Busan, South Korea.

[78] Rochwerger, B., et al. (2009). The Reservoir Model and Architecture for Open Federated Cloud Computing. IBM Systems Journal, 4(53), (pp.1-11).

[79] Keahey, K., Matsunaga, A., and Fortes, J. (2009). Sky Computing. IEEE Internet Computing, 13(5), (pp. 43–51).

[80] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2011). CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. Software: Practice and Experience, 1(41), (pp. 23-50), ISSN: 0038-0644, Wiley Press, New York, USA.

[81] Nudd, G. R., Kerbyson, D. J., Papaefstathiou, E., Perry, S. C., Harper, J. S., and Wilcox, D. V. (2000). Pace-A Toolset for the Performance Prediction of Parallel and Distributed Systems. International Journal of High Performance Computing Applications, 14(3), (pp. 228–-51).

[82] Smith, W., Foster, I., and Taylor, V. (1998). Predicting Application Run Times Using Historical Information. Proceedings of IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 1998), Florida, USA.

[83] Liu, Z., Squillante, M. S., and Wolf, J. L. (2001). On Maximizing Service-Level-Agreement Profits. Proceedings of the 3rd ACM conference on Electronic Commerce (EC 01), Tampa, Florida, USA.

[84] Menasce, D. A., Almeida, V. A. F., Fonseca, R., and Mendes, M. A. (1999). A Methodology for Workload Characterization of E-Commerce Sites. Proceedings of the 1999 ACM Conference on Electronic Commerce (EC 1999), Denver, CO, USA.

[85] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A. (2007). SLA Decomposition: Translating Service Level Objectives to System Level Thresholds. Proceedings of the 4th IEEE International Conference on Autonomic Computing, Florida, USA.

[86] Reig, G., Alonso, J., and Guitart, J. (2010). Deadline Constrained Prediction of Job Resource Requirements to Manage High-level SLAs for SaaS Cloud Providers. Tech. Rep. UPC-DAC-RR, Dept. d'Arquitectura de Computadors, University Polit'ecnica de Catalunya, Barcelona, Spain.

[87] Xiong, K., and Perros, H. (2008). SLA-based Resource Allocation in Cluster Computing Systems. Proceedings of the 17th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008), Alaska, USA.

[88] Netto, M., and Buyya, R. (2009). Offer-based Scheduling of Deadline-constrained Bag-of-Tasks Applications for Utility Computing Systems. Proceedings of the 18th International Heterogeneity in Computing Workshop (HCW 2009), in conjunction with the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), Roma, Italy.

[89] Garg, S. K., Buyya, R., and Siegel, H. J. (2010). Time and Cost Trade-off Management for Scheduling Parallel Applications on Utility Grids. Future Generation Computer Systems, 26(8), (pp. 1344-1355).

[90] Islam, M., Balaji, P., Sadayappan, P., and Panda, D. K. QoPS: A QoS Based Scheme for Parallel Job Scheduling. Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003), Seattle, USA.

[91] Islam, M., Sadayappan, P., and Panda, D. K. (2004). Towards Provision of Quality of Service Guarantees in Job Scheduling. Proceedings of the 6th IEEE International Conference on Cluster Computing (Cluster 2004), San Diego, USA.

[92] Varia, J. (2010). Architecting Applications for the Amazon Cloud. Cloud Computing: Principles and Paradigms, Buyya, R., Broberg, J., Goscinski, A. (eds), ISBN-13: 978-0470887998, Wiley Press, New York, USA. Web - http://aws.amazon.com

[93] CIO, retrieved 10 Sep 2010: http://www.cio.com.au.

[94] GoGorid, retrieved on 10 Sep 2010: http://www.gogrid.com.

[95] RackSpace, retrieved on 10 Sep 2010: http://www.rackspacecloud.com.

[96] Microsoft Azure, retrieved on 10 Sep 2010: http://www.microsoft.com/windowsazure/.

[97] IBM, retrieved on 10 Sep 2010: http://www.ibm.com/ibm/cloud/ibm_cloud/.

[98] Ostermann, S., Iosup, A., Yigitbasi, M. N., Prodan, R., Fahringer, T., and Epema, D. (2009). An Early Performance Analysis of Cloud Computing Services for Scientific Computing. Proceedings of the 1st International Conference on Cloud Computing (CloudCom 2009), Beijing, China.

[99] Kumar, S., Dutta, K., Mookeriee, V. (2009). Maximizing Business Value by Optimal Assignment of Jobs to Resources in Grid Computing, European Journal of Operational Research, 194(3).

[100] McManus, M. L., Long, M. C., Copper, A., and Litavak, E. (2004). Queuing Theory Accurately Models the Need for Critical Care Resources. Anesthesiology, 100(5), (pp. 1271-1276), Lippincott Williams and Wilkins; ISBN (0003-3022), USA.

[101] Wolff, R.W. (1982). Poisson Arrivals See Time Averages. Operations Research, 30(2), (pp. 223-231).

[102] Saleforce.com, retrieved on 10 Sep 2010: http://www.salesforce.com/au/.

[103] Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. (2009). A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. Proceedings of the 1st International Conference on Cloud Computing (CloudComp), Munich, Germany.

[104] Popovici, I., and Wiles, J. Proitable Services in an Uncertain World. Proceedings of the18th Conference on Supercomputing (SC 2005), Seattle, WA.

[105] Reig, G., Alonso, J., and Guitart, J. (2010). Prediction of Job Resource Requirements for Deadline Schedulers to Manage High-level SLAs on the Cloud. Proceedings of the 9th IEEE International Symposium on Network Computing and Applications (NCA 2010), Massachusetts, USA.

[106] Vecchiola, C., Chu, X. C., Mattess, M., and Buyya, R. (2011). Aneka-Integration of Private and Public Clouds. Cloud Computing Principles and Paradigms, Willy, USA.

[107] salesforce.com, retrieved on 06 Dec 2010: http://www.salesforce.com.

[108] Computer Associates Pty Ltd, retrieved on 06 Dec 2010: http://www.ca.com.

[109] Compiere ERP on Cloud, retrieved on 06 Dec 2010: http://www.compiere.com/.

[110] Yang, E. F., Zhang, Y., Wu, L., Liu, Y. L., and Liu, S. J. (2011). A Hybrid Approach to Placement of Tenants for Service-Based Multi-tenant SaaS Application. Proceedings of the 6th IEEE Asia-Pacific Services Computing Conference, Korea.

[111] Gad, T. (2010). Why Traditional Enterprise Software Sales Fail. http://www.sandhill.com/opinion/editorial_print.php?id=307. Referenced on March 6 2010.

[112] Fu, Y., and Vahdat, A. (2010). SLA Based Distributed Resource Allocation for Streaming Hosting Systems. Retrived on 06 Dec 2010: http://issg.cs.duke.edu.

[113] Yarmolenko, V., and Sakellariou, R. (2006). An Evaluation of Heuristics for SLA Based Parallel Job Scheduling. Proceedings of the 3rd High Performance Grid Computing Workshop (in conjunction with IPDPS 2006). Rhodes, Greece.

[114] Wu, L., Garg, S. K., and Buyya, R. (2011). SLA-based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments. Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011), Los Angeles, USA.

[115] Mensce, D., and Almeida, V. (2002). Capacity Planning for Web Performance: Metrics, Models and Methods. Prentice-Hall, Upper Sadale River, NJ.

[116] Hamscher, V., Schwiegelshohn, U., Streit, A., and Yahyapour, R. (2000). Evaluation of Job-Scheduling Strategies for Grid Computing. Proceedings of the 9th IEEE International Conference on Grid Computing (GRID 2000), Tsukuba, Japan.

[117] Gomoluch, J., and Schroeder, M. (2003). Market-based Resource Allocation for Grid Computing: A model and simulation. Proceedings of the 1st International Workshop on Middleware for Grid Computing (MGC 2003), Rio de Janeiro, Brazil.

[118] Pacifici, G., Spretzer, M., Tantawi, A. (2003). Performance Management of Cluster BasedWeb Services. Proceedings of the 11th IEEE/IFIP Symposium on Integrated Management, 2003, Colorado Springs, USA.

[119] Waldspurger, C. (2002). Memory Resource Management in VMware ESX Server. Proceedings of the 5th Symposium on Operating Systems Design and Implementation, 2002, Boston, USA.

[120] Alvarez, G., Borowsky, E., Go, S., Romer, T., Becker-Szendy, R., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., and Wilkes, J. (2001). Minerva: An Automated Resource Provisioning Tool for Large-Scale Storage Systems. ACM Transactions on Computer Systems, 1(19), (pp. 483-518), November, 2001.

[121] Kimbre, T., Schieber, B., and Svirdenko, M. (2004). Minimizing Migrations in Fair Multiprocessor Scheuling of Persistent Tasks. Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms, 2004, New Orleans, USA.

[122] Khanna, G., Beaty, K., Kochut, A., and Kar, G. (2006). Dynamic Application Management to Address SLAs in a Virtulized Server Environment. Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium, 2006, Vancouver, Canada.

[123] Grit, L., Irwin, D., Yumerefendi, A., and Chase, J. (2006). Virtual Machine Hosting for Networked Clusters: Building the Foundations for Autonomic Orchestration. In Proceeding of the 2nd IEEE International Workshop on Virtualization Technology in Distributed Computing, 2006, Tampa, USA

[124] Van, H. N., Tran, F. D., and Menaud, J.-M. (2009). SLA-aware Virtual Resource Management for Cloud Infrastructures. In Proceeding of 9th IEEE International Conference on Computer and Information Technology, 2009, Xiamen, China.

[125] Hermenier, F., Lorca, X., and Menaud, J.-M. (2009). Entropy: A Consolidation Manager for Clusters. In Proceeding of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), 2009, Hamilton Crowne Plaza, Washington.

[126] Bobroff, N., Kochut, A., and Beaty, K. (2007). Dynamic Placement of Virtual Machines for Managing SLA Violations. In Proceeding of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM), 2007, Munich, Germany.

[127] Chaisiri, S., Lee, B., and Niyato, D. (2011). Optimization of Resource Provisioning Cost in Cloud Computing. IEEE Transactions on Services Computing, preprint, Feb. 2011, DOI: http://doi.ieeecomputersociety.org/10.1109

[128] He, X. S., Sun, X. H., and Von Laszewski, G. (2003). QoS Guided Min-min Heuristic for Grid Task Scheduling. Journal of Computer Science and Technology, 18(4), (pp. 442-451), July 2003.

[129] Bryant, A., and Colledge, B. (2002). Trust in Electronic Commerce Business Relationships. Journal of Electronic Commerce Research, 3(2), (pp. 32-39).

[130] Crago, S., Dunn, K., Eads, P., Hochstein, L., Dong-In, K., Mikyung, K., Modium, D., Sigh, K., Woo, S. J., Walters. J. P. (2011). Heterogeneous Cloud Computing. In Proceeding of the IEEE International Conference on Cluster Computing (CLUSTER). Austin, Taxas.

[131] Sumit, A., Driscoll, J., Gabaix X., and Laibson, D. (2009). The Age of Reason: Financial Decisions over the Life-Cycle with Implications for Regulation. Brooking Chapters on Economic Activity Fall 2009, (pp. 51-117).

[132] Garey, M. R., and Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, San Francisco, USA.

[133] Chen, Y., Das, A., Qin, W., Sivasubrammaniam, A., Wang, Q., and Gautam, N. (2005). Managing Server Energy and Operational Costs in Hosting Centers. ACM Sigmetrics Performance Evaluation Review 22(1), (pp. 303-314).

[134] Martello, S., and Toth, P. (1981). An Algorithm for the Generalized Assignment Problem, Operational Research 81, (pp. 589-603).

[135] Goolgle App Engine, retrieved on 06 June 2012: http://www.google.com/enterprise/apps/business.

[136] Micorosoft, Hyper-V, http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx. Accessed on 06 June 2010:

[137] VMWare, retrieved on 06 June 2012: http://www.vmware.com/.

[138] Wu, L. L., Garg, S. K., and Buyya, R. (2012). SLA-based Admission Control for a Software-as-a-Service Provider in Cloud Computing Environments. Journal of Computer and System Sciences, 78(5), (pp. 1280-1299), Sep. 2012.

[139] Cooley, R. (2003). The Use of Web Structures and Content to Identify Subjectively Interesting Web Usage Patterns. ACM Transactions on Internet Technology 3(2), (pp. 93-116).

[140] Srivastava, J., Cooley, R., Deshpande, M., Tan, P.-N. (2000). Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. ACM SIGKDD Explorations Newsletter, 1(2), Jan. 2000.

[141] Brian, D. D. (2004). Learning Web request patterns. In A. Poulovassilis and M. Levene (eds), Web Dynamics: Adapting to Change in Content, Size, Topology and Use, (pp. 435-460). Springer.

[142] Su, Z., Yang, Q., Lu, Y., Zhang, H. (2000). WhatNext: A Prediction System for Web Requests Using N-gram Sequence Models. Proceedings of the 1st International Conference on Web Information Systems Engineering (WISE'00), 1, (pp. 214), June 19-20, 2000.

[143] Kurian, H. (2008). A Markov Model for Web Request Prediction. Master's thesis, Kansas State University, Department of Computing and Information Sciences, Kansas, USA.

[144] Chao, K., Anane, R., Chen, J. H., Gatward, R. (2002). Negotiating Agents in a Market-oriented Grid. Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002), Berlin, Germany.

[145] Sim, K. M. (2006). A Survey of Bargaining Models for Grid Resource Allocation. ACM SIGECOM: E-Commerce Exchange, 5(5), (pp. 22–32).

[146] Faratin, P., Sierra, C., Jennings, N. R. (1998). Negotiation Decision Functions for Autonomous Agents, Robotics and Autonomous System, 24(3-4), (pp. 159-182).

[147] Chhetri, M., et. al. (2006). A Coordinated Architecture for the Agent-based Service Level Agreement Negotiation of Web Service Composition. Proceedings of Australian Software Engineering Conference. (ASWEC), Washington,

[148] Comuzzi, M., and Pernici, B. (2005). An Architecture for Flexible Web Service QoS Negotiation. Proceedings of the 9th IEEE International Enterprise Computing Conference, Enschede, The Netherlands.

[149] Zulkernine, F. et al. (2009). In a Policy-based Middleware for Web Services SLA Negotiation. IEEE International Conference on Web Serivce (ICWS), (pp. 1043-1050).

[150] Akhani, J., Chaudhary, S., and Somani, G. (2011). Negotiation for Resource Allocation in IaaS Cloud. Proceedings of the 4th Annual ACM Bangalore Conference, Banglore, India.

[151] Brzostowski, J., and Kowalczy, R. (2006). Adaptive Negotiation with On-line Prediction of Opponent Behaviour in Agent-based Negotiations. Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology, HongKong, China.

[152] Zukernine, F., and Martin, P. (2011). An Adaptive and Intelligent SLA Negotiation System for Web Services. IEEE Transactions of Service Computing, 4(1), (pp. 31-43).

[153] Shell. M., Comuzzi, M., and Pernici, B. (2007). An Architecture for Flexible Web Service QoS Negotiation. Proceedings of the 1st IEEE International Enterprise Distributed Object Computing (EDOC) Conference, Maryland, USA.

[154] Li, H., Su, S., and Lam, H. (2006). On Automated E-Business Negotiations: Goal, Policy, Strategy and Plans of Decision and Action. Journal of Organizational Computing and Electronic Commerce, 13(1), (pp. 1-29).

[155] Retrieved on 10 April 2012: http://vitlive.com.

[156] Retrieved on 06 April 2012: http://www.cloudharmony.com.

[157] Comuzzi, M., and Pernici, B. (2009). A Framework for the QoS-based Web Service Contracting. ACM Transaction on the Web, 3(3), (pp. 1-10).

[158] Retrieved on 10 April 2012: http://sites.google.com/site/gistcloudresearchgroup/automated-sla-negotiation.

[159] Garg, S. K., Vecchiola, C., and Buyya, R. (2012). Mandi: A Market Exchange for Trading Utility and Cloud Computing Services. The Journal of Supercomputing, Volume

64, No. 3, Pages: 1153-1174, ISSN: 0920-8542, Springer Science+Business Media, Berlin, Germany, June 2013.

[160] Czajkowski, K., Foster, I., and Kesselman, C. (1999). Resource Co-allocation in Computational Grids. Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing.

[161] Cao, J. W., Spooner, D. P., and Nudd, G. R. (2002). Agent-based Resource Management for Grid Computing. Proceedings of the 2nd International Symposium on Cluster Computing and the Grid, Germany

[162] Binmore, K., and Vulkan, N. (1997). Applying Game Theory to Automated Negotiation. Chapter prepared for DIMACS Workshop on Economics, Game Theory and the Internet.

[163] Arai, S., Sycara, K., and Payne, T. (2000). Experience-learning in based Reinforcement Learning to Acquire Multi-Agent Domain. Proceedings the Sixth Pacific Rim International Conference on Artificial Intelligence, Springer-Verlag.

[164] Teuteberg, F., and Kurbel, K. (2002). Anticipating Agents' Negotiation Strategies in an E-marketplace Using Belief Models. Proceedings of the 5th International Conference on Business Information System, Poland.

[165] Faratin, P., et. al. (2000). Using Similarity Criteria to Make Negotiation Trade-offs. Proceedings of the 4th International Conference on Multi-Agent Systems, Boston, USA.

[166] Dastjerdi, A. V., and Buyya, R. (2012). An Autonomous Reliability-Aware Negotiation Strategy for Cloud Computing Environments. Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid), Ottawa, Canada.

[167] Retrieved on 10 April 2012: http://www.cordys.com/cordys-for-cloud-brokers.

[168] Reumann, J., Mehra, A., Shin, K.G., and Kandlur, D (2000). Virtual Services: A New Abstraction for Server Consolidation, Proceedings of the 2000 USENIX Annual Technical Conference (USENIX ATC), San Diego, CA.

[169] Rooney, S (2000). The IcorpMaker: A Dynamic Framework for Application-Service Providers, Proceedings of the IEEE Workshop on IP-oriented Operations and Management, Cracow, Poland.

[170] Bruno, J., Gabber, E., Ozden B., and Silberschatz A. (1998). The Eclipse Operating System: Providing Quality of Service via Reservation Domains, Proceedings of the 1998 USENIX Annual Technical Conference (USENIX ATC), New Orleans, LA.

[171] Vogels, W., and Dumitriu, D.M. (2000). An Overview of the Galaxy Management Framework for Scalable Enterprise Cluster Computing, Proceedings of the IEEE International Conference on Cluster Computing (Cluster), Chemnitz, Germany.

[172] Padala, P. et. al. (2009). Automated Control of Multiple Virtualized Resources. Proceedings of the IEEE 4th EuroSys Conference, Nuremberg, Germany.

[173] Sukwong, O., Sangpetch, A., and Kim, H.S. (2012). SageShift: Managing SLAs for Highly Consolidated Cloud, Proceedings of the 31st IEEE INFOCOM, Orlando, FL.

[174] Appleby, K et. al., Océano – SLA Based Management of a Computing Utility (2001). Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management, Dublin, Ireland.

[175] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. (1998). Practical solutions for QoS-based resource allocation problems. Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS), Madrid, Spain.

[176] Telefónica I + D, Claudia Platform, 2013. URL http://claudia.morfeoproject.org/.
[177] The OPTIMIS Consortium, OPTIMIS: optimized infrastructure services, 2013. URL http://www.optimis-project.eu/.
[178] The 4CaaSt Consortium, Building the PaaS cloud of the future, 2013. URL http://4caast.morfeo-project.org/.
[179] The BonFIRE Consortium, Building service test beds on FIRE, 2013. URL http://www.bonfire-project.eu/
[180] The Cloud-TM Consortium, Cloud-TM: a novel programming paradigm for cloud computing, 2013. URL http://www.cloudtm.eu/.
[181] The PaaSage Consortium, PaaSage: model based cloud platform upperware, 2013. URL http://www.paasage.eu/.
[182] The SLA@SOI, 2014. URL http://sla-at-soi.eu/
[183] Cloud Security Alliance (CSA), Security guidance for critical areas of focuses in cloud computing v3.0, https://cloudsecurityalliance.org/. Accessed on 10 July 2014.