Optimizing PaaS provider profit under service level agreement constraints

Djawida Dib

▶ **To cite this version:**

Djawida Dib. Optimizing PaaS provider profit under service level agreement constraints. Networking and Internet Architecture [cs.NI]. Université de Rennes, 2014. English. NNT : 2014REN1S044 . tel-01124007

HAL Id: tel-01124007
https://theses.hal.science/tel-01124007

Submitted on 6 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2014

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Djawida Dib

préparée à l'unité de recherche nº 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
ISTIC

Optimizing PaaS Provider Profit under Service Level Agreement Constraints

Thèse soutenue à Rennes
le 7 juillet 2014

devant le jury composé de :

Jean-Marc Menaud / Rapporteur
Professeur, École des Mines de Nantes

Pierre Sens / Rapporteur
Professeur, Université Pierre et Marie Curie

Michel Banâtre / Examinateur
Directeur de Recherche, Inria Rennes - Bretagne Atlantique

Stephen Scott / Examinateur
Professeur, Tennessee Tech University

Christine Morin / Directrice de thèse
Directrice de Recherche, Inria Rennes - Bretagne Atlantique

Nikos Parlavantzas / Co-directeur de thèse
Maître de Conférences, INSA de Rennes

When you see clouds gathering, prepare to catch rainwater ...
Golan proverb

Remerciements

Cette thèse est une aventure dans laquelle plusieurs personnes ont fait partie.

Tout d'abord, j'ai une profonde pensée pour Françoise André, avec qui j'ai commencé ma thèse et qui nous a malheureusement quitté avant la fin de ma première année de thèse. Françoise était une directrice rigoureuse et perfectionniste. Grâce à elle j'ai appris à développer des arguments solides pour défendre mes idées.

Je tiens à exprimer ma gratitude à Christine Morin pour avoir pris le relais de Françoise et d'avoir toujours veillé à ce que ma thèse se passe dans de bonnes conditions. Merci Christine d'avoir su orienter mes travaux. Merci également de m'avoir trouvé du temps pour des discussions malgré un planning très chargé. Chacune de nos discussions m'a permis de voir mes travaux de différents angles et de penser à plein de perspectives.

Je remercie Nikos Parlavantzas d'avoir co-diriger ma thèse et d'avoir été disponible pour discuter sur toutes les idées et les approches utilisées, même sur les plus petits détails de configuration. Toutes nos discussions étaient très enrichissantes. Merci Nikos pour toutes tes reflexions pertinentes qui ont fait avancer mes travaux de thèse.

J'adresse mes remerciements aux membres de jury : Pierre Sens et Jean-Marc Menaud d'avoir été rapporteurs, Michel Banâtre d'avoir présider le jury et Stephen Scott d'avoir participer à l'évaluation de ma thèse. Je vous remercie tous pour l'intérêt que vous avez porté à mes travaux et pour vos retours constructifs.

Je remercie tous les membres de l'équipe Myriads pour tous les moments agréables que j'ai pu partager avec eux. Merci Pierre d'avoir toujours été disponible pour donner un coup de main ou un conseil et ça depuis mon stage de Master. Je remercie Marko, Louis et Maxence pour leur support concernant Kerrighed. Je remercie Guillaume et Erwan pour leur support concernant Safdis. Un grand merci à Eugen pour toutes les discussions intéressantes et amusantes ainsi que pour son support concernant Snooze. Je remercie également Matthieu d'avoir pris le relais de Eugen dans le support technique de Snooze. Je remercie Anca pour ses scripts de configuration d'Hadoop. Je remercie Guillaume, Anne-Cécile et Alexandra pour notre contribution fructueuse *"Green PaaS"*. Merci à Roberto et Anne-Cécile pour la pertinence de leurs retours concernant mes présentations orales. Je remercie la communauté de Grid'5000 et plus particulièrement David, Yvon et Pascal pour leur support technique. Un grand merci à Maryse pour son aide dans les formalités administratives ainsi que dans l'organisation des missions qui n'étaient pas des plus simples. Je remercie chaleureusement Anca, Stefania, Elya, Peter, Rémy, Amine, Guillaume, Mohammed, Ghislain, Alexandra, Bogdan, Anne-Cécile, André, Pierre, Roberto, Eugen, Matthieu, ... et tous ceux qui rendaient le quotidien social agréable.

Un énorme merci à ma famille pour m'avoir toujours soutenue et encouragée. À mes parents pour leur affection, leurs sacrifices et leur soutien inconditionnel. À mon mari pour avoir été à mes côtés pendant les moments les plus difficiles de la thèse et d'avoir toujours su trouver les mots pour me rassurer. À mon petit bout'chou Abdesselem pour m'avoir tenue compagnie pendant la rédaction et la soutenance de la thèse. À ma soeur Nassima et mon petit frère Mohammed Riad pour leur amour et leurs encouragements infinis. À mes frères Mokhtar et Kheir-edinne. À mon beau frère Samir. À mes belles soeurs Leila et Souhila. À mes petits nerveux et nièces Mehdi, Zakia, Abderrahmene, Meriem et Yacine. À mes grands mères pour toutes leurs prières. À mes tantes, oncles, cousins et cousines. À mes beaux parents et mes belles soeurs Ilhem et Dounyazed pour

leur sympathie et leurs encouragements.

J'ai une pensée particulière pour ma mère Zakia. J'espère qu'elle est contente de ce que j'ai accomplis de là où elle est.

Enfin, un grand merci à tous mes amis. À Sarra pour son amitié exceptionnelle et pour ses encouragements. À Esma pour toutes nos discussions et pour son grand soutien, notamment durant les derniers jours précédant la soutenance. À Wahida, Fadhela, Rafik, Hakim, ... et tous ceux qui ont partagé avec moi cette aventure.

Contents

List of Figures

List of Figures

List of Tables

List of Tables

Chapter 1

Introduction

Contents

This PhD thesis examines economic optimizations in cloud computing environments, focusing mainly on the optimization of PaaS providers' profits. This chapter introduces the context, objectives and contributions of this PhD thesis and presents the outline of this document.

1.1 Context

Cloud computing is an emerging paradigm revolutionizing the use and marketing of Information Technology (IT). The cloud computing concept enables customers from all over the world to access very large computing capacities, using a simple Internet connection, while paying only for the resources they really use. Indeed, the *pay-as-you-go* pricing model of cloud computing services attracts many customers and small companies aiming at reducing the cost of their IT usage and offers providers new opportunities for commercializing computing resources. Cloud computing services are provided according to three fundamental models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The IaaS model provides basic computing resources, such as computing machines as well as storage and networking devices. The PaaS model provides configured execution environments, ready to host cloud applications. The SaaS model provides hosted applications, ready for use.

In this thesis we are interested in the PaaS service model, which gained a lot of attention over the past years. A PaaS environment is typically built on top of virtualized resources owned by the PaaS provider or rented on-demand from public IaaS providers. The advantage of this model over the two others is that it hides the complexity of the used resources while offering customers the possibility to run and control their own applications. The interactions between PaaS customers and providers are governed by Service Level Agreements (SLAs), specifying the obligations of each party as well as associated payments and penalties. For instance, the PaaS offerings of the current big players, such as Amazon Elastic Beanstalk [10] and Microsoft Windows Azure [13], guarantee a high service availability. Their clients pay for the resources consumed by their applications. Then, if the promised availability target is missed, the providers give customers a service credit.

The socio-economic impact of PaaS services becomes critical since the number of PaaS users and providers is growing. PaaS providers want to generate the maximum profit from the services they provide. This requires them to face a number of challenges. They have to efficiently manage the underlying resources and to decide between using privately owned resources or renting public IaaS resources. Moreover, they must manage the placement and isolation of customer applications on the resources and satisfy their SLAs in order to avoid the payment of penalties. Current big commercial PaaS providers, such as Amazon and Microsoft, use their own resources. However, smaller PaaS providers, such as Heroku [15], CloudBees [34] and dotCloud [35], rent on-demand public IaaS resources. In this thesis we consider a *cloud-bursting* PaaS environment, which is built on a limited number of private resources and is able to burst into public IaaS cloud resources. Such an environment represents the general case and offers several options for the PaaS provider to select resources for hosting applications.

Recently, a number of research works focusing on economic optimizations in multi-cloud environments have been proposed. However, most of them either focus on specific application types [120][79][64] or target specific hosting environments [90][141][158]. Therefore, exploring and investigating the possible solutions for managing a cloud-bursting PaaS environment with the objective of optimizing the provider profit remains an open research topic. The work of this thesis is part of this theme.

1.2 Objectives

The main objective of this thesis is to provide *a profit-optimization solution for SLA-enabled, cloud-bursting, PaaS systems*. To achieve this objective, this thesis investigates four main sub-objectives:

- **Profit Optimization.** The profit optimization is the central objective of every PaaS provider. On one hand, this involves accepting and satisfying client requests whether in low or peak periods. On the other hand, the competition between providers compels each provider to propose prices following the market, which limits their revenues. Therefore, the provider profit optimization should be performed through the optimization of the incurred costs for hosting applications. However, if such a cost optimization leads the provider to miss its commitments regarding the QoS promised to applications, the customers will be disappointed and may migrate to an other competitor. Thus our objective here is *to provide a policy that searches the cheapest resources for the provider to host clients applications, taking into account the provider's reputation and its long-term profit*.

- **SLA Support.** PaaS customers expect PaaS providers to propose SLA contracts that guarantee a QoS level to their applications. An important part of the SLA agreement consists in arranging to compensate users if the agreed QoS level of their applications is missed. Such compensations may penalize providers' profits but are necessary to gain the confidence of clients. Thus, our goal here is to *provide QoS-based SLA to applications* and to *take into account the payment of penalties if the providers fail in providing the QoS level promised to their clients' applications*.

- **Support for Multiple Application Types.** An open PaaS solution, easily extensible to support of new application types, is appealing to both customers and providers. On one hand, it offers more flexibility to customers having various requirements. On the other hand, it enables PaaS providers to attract clients from several professional areas and to easily add the support of any new profit-making application type, which helps increasing their turnovers. Therefore, we aim at providing *an open and generic PaaS profit-efficient solution, which is independent from a specific application type*. The difficulty in building such a PaaS solution lies in the support of various application types, each application type having its own hardware and software dependencies.

- **Cloud-Bursting PaaS System.** A cloud-bursting PaaS system enabling the deployment of applications simultaneously on private and public cloud resources offers PaaS providers several deployment options to optimize either application performance or costs. The support of cloud-bursting capability requires significant engineering work to enable the PaaS system to use several IaaS systems and to maintain the SLA of the applications deployed simultaneously on several IaaS clouds. Therefore, our goal is *to design a cloud-bursting PaaS system enabling the deployment of applications on multiple clouds*.

1.3 Contributions

This thesis tackles the introduced objectives with the following contributions.

- **Profit Optimization Model.** We define a generic model of an open cloud-bursting PaaS system, based on which we propose a policy for optimizing the PaaS provider profit. The profit optimization policy proposes a placement of applications on the cheapest resources, taking into account the satisfaction of their SLAs. The proposed policy is generic and may be applied on any application type with only one condition, which consists in cooperating with application type-specific entities providing information about the performance and SLAs of the hosted applications.
- **Application of the Optimization Model.** To demonstrate the genericity of the model, we applied it to rigid and elastic computational applications. Specifically, we defined corresponding SLA terms. Based on that, we proposed heuristics for providing the information required by the generic profit optimization policy. The idea behind this investigation is to show a concrete and complete illustration of the generic profit optimization policy.
- **Meryn: an Open Cloud-Bursting PaaS.** We propose an open cloud-bursting PaaS system, called *Meryn*, providing all the features required for implementing the generic profit optimization policy. To support the extensibility regarding application types, we use existing frameworks for managing the supported application types. For instance, the Oracle Grid Engine (OGE) framework is used for managing batch applications and the Hadoop framework is used for managing MapReduce applications. Moreover, to facilitate the management of cloud bursting, we built Meryn on top of virtualized private resources similar to the virtual resources leased from public cloud providers.

1.4 Outline of the Thesis

The rest of this thesis is organized as follows. Chapter 2 presents background definitions in the context of our work. Specifically, it gives an overview of the different IT infrastructures and application models, and defines the service level agreement and cloud computing concepts. Chapter 3 presents the requirements to achieve the thesis objectives, covers the state of the art and positions the contributions of this thesis. The main covered axes are: economic optimization policies, cloud bursting mechanisms, and PaaS systems. Chapter 4 describes the first contribution of this thesis consisting in a generic model for optimizing the provider profit in an open cloud-bursting PaaS system environment. Chapter 5 investigates rigid and elastic computational applications in order to enable a concrete applicability of our generic profit optimization model. Chapter 6 presents the design principles, architecture and implementation of *Meryn*, our open cloud-bursting PaaS system. Chapter 7 presents an evaluation of our approach through a set of simulations and experiments performed on the Grid'5000 testbed [40]. Chapter 8 concludes this thesis by summarizing our contributions and presenting some future work directions.

Chapter 2

Background

Contents

This chapter provides the background related to the contributions of this PhD thesis. First, we present existing IT infrastructures and a taxonomy of existing application models. Then, we present the Service Level Agreement (SLA) concept. Finally, we present the cloud computing technology, its service and deployment models as well as its business aspects.

2.1 IT Infrastructures

IT (Information Technology) infrastructures refer to systems composed of software and hardware to process and manage information automatically. IT infrastructures emerged over the last century, evolved very quickly, and revolutionized our society and our industry. The development of IT infrastructures has gone through several steps and gave rise to the emergence of multiple software models. Software refers to programs written using specific programming languages and hosted on specific IT infrastructures. There are mainly two classes of software: *system* and *user* software. System software is designed to operate and manage the IT infrastructure's hardware and provide a platform for running user software. User software, also called *Applications*, is designed to perform specific tasks to help users. In this section we present a number of existing IT infrastructures as well as their respective system software.

2.1.1 Cluster

A cluster is a set of homogeneous computers interconnected through a fast Local Area Network (LAN)[1]. There are mainly two types of cluster's system software: *programming frameworks* and *Single System Images (SSI).* A programming framework, also called a *cluster resource manager*, provides control over the cluster's compute resources to allocate them to users jobs[2]. Programming frameworks are often configured in a master node and a set of slave nodes. The master node has a global view of the cluster. It is responsible for scheduling the jobs on the slave nodes using specific scheduling algorithms such as the well known *first-fit* and *round-robin* algorithms. The slave nodes are responsible for the execution of the jobs assigned to them, as illustrated in Figure 2.1. For running applications in such systems, the users have to connect to the master node, wrap their applications using a job template specific to the programming framework and submit their jobs using the programming framework tools. Examples of well known programming frameworks include batch frameworks such as OAR [69], Torque [138], Oracle Grid Engine (OGE) [89] and data-intensive frameworks such as Hadoop [151] and HPCC [29] for respectively batch and data-intensive applications (more details about application types are given in Section 2.2).

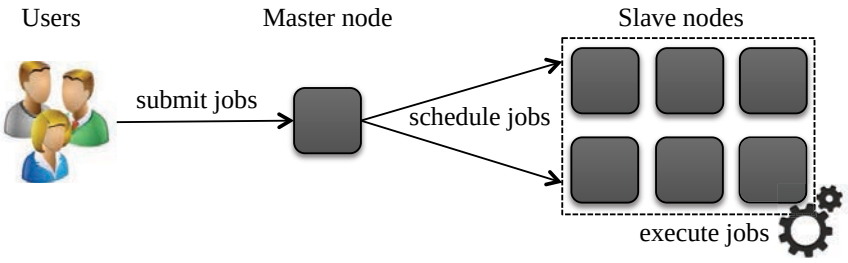


Figure 2.1: Overview of a cluster managed using a programming framework

A single system image (SSI), also called a *cluster operating system*, hides the distributed nature of the cluster and gives users the illusion of a single multiprocessor

[1]Local Area Network (LAN) is a communication system used to interconnect computers in a limited geographical area, such as a campus, a laboratory or a company.

[2]A job is a collection of one or more applications that run together.

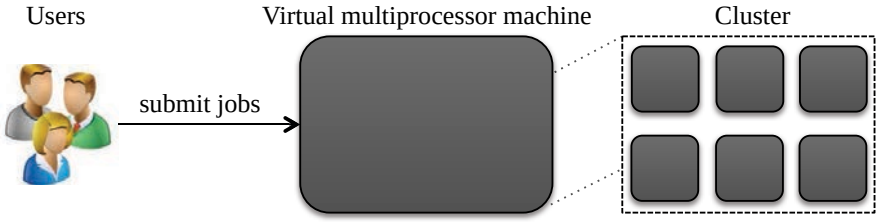


Figure 2.2: Overview of a cluster managed using a single system image

machine. This enables the use of the cluster in exactly the same way as the use of a single computer, as seen in Figure 2.2. Usually SSIs are implemented as an extension of standard operating systems. In such systems, users may connect to any node of the cluster to have a global view of the available resources and run their applications. Examples of SSI are Kerrighed [125], MOSIX [60] and OpenSSI [147].

2.1.2 Grid

A grid is a federation of geographically distributed heterogeneous computing resources interconnected through a Wide Area Network (WAN)[3]. The computing resources may be individual machines, clusters or any other PDA[4] devices, belonging to several Virtual Organizations (VOs). Virtual organizations may be a group of people or real organizations with common goals. They share their resources for the purpose of providing access to compute resources in the same way as electric grids. However, the access to resources is governed by specific sharing rules defined locally by each virtual organization. Users must join a VO to be able to use the grid resources. Examples of existing grids include experimental grids such as PlanetLab [62] and Grid'5000 [65], and compute grids such as the European Grid Infrastructure (EGI) [30].

There are mainly two types of grid system software: *grid middleware* and *grid operating system*. A grid middleware is a software layer between operating systems installed in the individual computers and the applications. It provides users with standardized interfaces for monitoring, reserving and using the grid resources for running their applications. Some well known grid middleware are Globus [31], gLite [112] and DIET [55]. A grid operating system hides the heterogeneity of the grid environment and provides users with the same functionalities of a traditional operating system. Examples of grid operating systems include XtreemOS [124], Legion [94], and GridOS [128].

2.1.3 Utility Computing

Utility computing is a computing business model which consists in providing computing resources on-demand, as a measured service, in exchange of a payment according to the use of resources. This model is inspired from the conventional utilities such as phone services, electricity and gas. Basically, the entity that owns, operates, manages and provides the resources is called a *provider* and the entity that accesses and uses the resources is called a *customer*. The provider makes the resources available to users

[3]Wide Area Network (WAN) is a communication network used to interconnect computers from very large geographic area that cannot be covered using a LAN.

[4]PDA (Personal Digital Assistant) is a handheld computing device such as smartphones.

through a network. The provided resources may be either computational hardware or software applications. Utility computing may be applied in any computing environment but the most known and used environment is *cloud computing*. We present the cloud computing technology in detail in Section 2.4.

2.2 Taxonomy of Application Models

An application is a computing software that consists of a set of tasks, designed to solve specific problems from a specific activity domain. Nowadays, there are several software applications for almost all life activity domains such as health, education, and science. Applications may be classified according to many parameters, such as the activity domain of the problem to solve, used programming languages, supported operating systems and infrastructures and so on. In this section we classify applications based on their parallelism model, resource requirements, resource usage, lifetime and interactivity.

2.2.1 Parallelism Model

There are mainly three application parallelism models: *sequential*, *parallel* and *distributed*. A sequential application is a single process containing a set of instructions to run one after another in a successive fashion. The sequential applications may run on any IT infrastructure type, even on a simple uni-processor and mono-task computer.

A parallel application consists of multiple processes or threads able to run simultaneously where each process or thread has a set of instructions to run sequentially. The parallel applications are designed to solve very large problems requiring capabilities of more than one CPU. Thereby, usually a separate CPU is assigned for each process or thread that composes the parallel application. Parallel applications may run on a multiprocessor machine or on a cluster managed using an SSI system software. They may also run or uni-processor machines but in this case they will not run in parallel, they will just have the illusion of parallelism.

A distributed application is a collection of multiple processes to run on multiple computers. The distributed applications are designed as independent processes; each one solves a subproblem. Applications are designed to be distributed either to take advantage of available computation resources or because the distribution is necessary for solving the concerned problem. For example utility computing providers may require a distributed software to manage users requests where one part of the software runs on the users side for enabling them specifying their requests and preferences and the other part runs on the providers side for configuring and providing the required resources for the users.

Parallel and distributed applications can also be classified as *tightly-coupled* or *loosely-coupled* applications according to their communication patterns. In a tightly-coupled application the processes have frequent communications and inversely in a loosely coupled application the communication between the processes is insignificant.

2.2.2 Resource Requirements

Based on their resource requirements, applications can be classified as *static* and *dynamic*. The static applications require a fixed amount of resources over their whole execution.

They can be subclassed as either *rigid* or *moldable* applications. The number of resources required for running a rigid application is decided at the development time. Therefore, rigid applications may only run on a specific configuration of resources. The number of resources required for running a moldable application is decided at the submission time. Thus, moldable applications can run on a wide variety of resource configurations. The flexibility on allocating resources to moldable applications may considerably improve the utilization of resources and the performance of the applications [100]. Some moldable applications may only run on a predefined set of specific resource configurations such as the NAS Parallel Benchmarks [59].

The resource requirements of dynamic applications may change during their executions. Basically, there are two types of dynamic applications: *malleable* and *evolving* applications. Malleable applications are able to adapt to changes in their resource allocation during their execution. Thus, the malleable applications may be shrinked or extended depending on the availability of resources. However, these applications have a minimum and maximum values of resource requirements. The application does not progress if its allocated resources are less than the minimum value and does not benefit from the allocated resources that exceed the maximum value. Examples of malleable applications include Bag of tasks [75] and MapReduce applications [77]. Evolving applications are constrained by a resource requirement change during their execution. The resource requirement change may be generated either internally because of an increased or decreased complexity of the executed algorithms, or externally because of specific interactions with external entities. Scientific workflows [63] are an example of evolving applications.

2.2.3 Resource Usage

The applications may be classified based on the main hardware resource used intensively. Namely, there are four main application classes according to the resource usage parameter: *CPU-intensive*, *IO-intensive* and *data-intensive* applications. A CPU-intensive application uses a high percentage of CPU resources and may reach 100% of CPU(s) usage. Usually CPU-intensive applications are designed to be parallel or distributed. An IO-intensive application reads and/or writes data very frequently, possibly with big size, either from stored files or from IO devices. A data-intensive application processes a very large amount of data and may have different access patterns to it. MapReduce applications are a well known example of data-intensive applications.

2.2.4 Lifetime

Software applications may be hosted in the infrastructures in two ways: *permanently* or *temporally*. A permanent software application ends only if the underlying infrastructure is turned off or following a manual user intervention. The permanent applications are not necessarily always actively using resources. Web applications are a good example of permanent applications. In contrast, a temporary software application runs during a finite duration that may be known in advance or not.

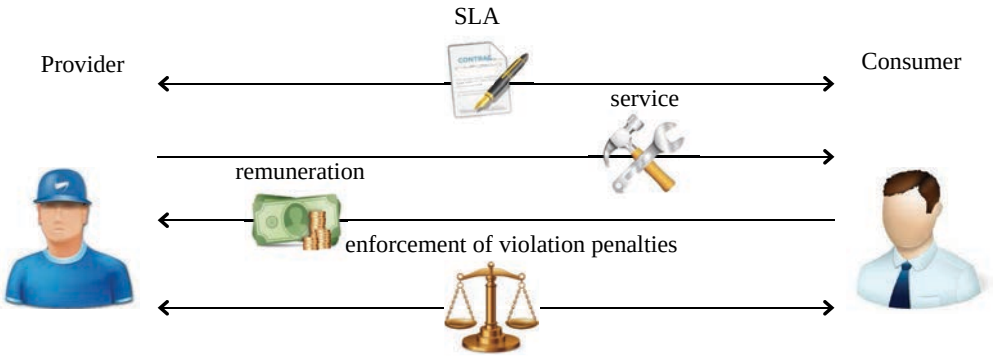


Figure 2.3: SLA overview

2.2.5 Interactivity

The applications may also be classified according to the degree of their interactivity with users during their execution. The interactivity between applications and users may be in the form of a request to select an option or to give specific data. An application is said to be *interactive* if it requires recurrent interactions with the user. In general, interactive applications become temporally idle when an interaction is required. Examples of interactive applications include visualization applications. An application is said to be *passive* or *batch* if it requires no interaction with users during its execution. The required input data of a passive application is given at submission time and the possible output data is given at the end of the application execution. Generally, passive applications are not permanent and can be suspended and resumed later, or check-pointed or killed and restarted later. Examples of passive applications include a big number of scientific simulations.

2.3 Service Level Agreement (SLA)

2.3.1 Overview

A Service Level Agreement (SLA) is a formal contract concerning a given service between two parties: the *service provider* and the *service consumer*. The SLA explicitly defines the concerned service using measurable metrics such as the service availability and performance. It also specifies the expectations and obligations of each party. A cloud provider may also list in the SLAs a set of restrictions or limitations, and obligations that cloud consumers must accept. If the customer or the provider violates the agreement, the SLA indicates how violation penalties will be enforced (see Figure 2.3). Thus, the SLA is used to guarantee the satisfaction of the service quality expected by the consumer and the deserved remuneration for the provider. SLAs started to be used in IT domains with the advent of utility computing. Then, many system models have been proposed in the literature to support autonomic SLA management [106][67][152] which consists in managing and automating the entire SLA lifecycle using an SLA specification language.

2.3.2 SLA Specification

To ease the automation of SLA negotiation and management, the SLA should be presented in a formalized way such that the concerned service may be adapted according to the agreed SLA terms. Several SLA specification languages have been proposed in the literature. The most popular and widely used ones are WSLA [119] and WS-Agreement [56]. Both of them provide support for the entire SLA lifecycle and rely on WSDL (Web Service Description Language) to describe the service functionalities. WSLA (Web Service Level Agreement) is a framework developed by IBM to specify and monitor SLAs for web services. It provides a formal language based on XML to represent SLAs for both the provider and the consumer. Based on the interpretation of the WSLA document the provider and the consumer configure their respective systems. The WSLA framework can measure, monitor and report to the two parties the values of multiple implemented SLA metrics. Moreover, it enables the creation and the implementation of new SLA metrics. WS-Agreement (Web Service Agreement) is a language and protocol defined by Open Grid Forum (OGF) for the creation, specification and management of SLAs. The WS-Agreement language uses XML to formalize the agreement in a template. The template defines, among other things, the agreement parties and describes the offered service. The WS-Agreement protocol is based on request response for the interaction between the provider and the consumer. WS-Agreement does not define specification for SLA metrics as WSLA but many research works have been proposed in order to extend the WS-Agreement model such as [134], [131] and [126].

2.3.3 SLA metrics

The SLA metrics are measurable aspects, defining what services and guarantees the cloud provider will provide. The SLA metrics may be categorized as functional and nonfunctional. Functional properties cover aspects like the number of arguments and the semantics of operations. Nonfunctional properties define the service capabilities and robustness, covering terms regarding the QoS, security, and remedies for performance failures. The SLA metrics should be monitored and reported to both the provider and the consumer for the assessment of the service's ability to achieve the agreement.

2.3.4 Price Formation and Billing Models

The price formation model determines how to account the price of a service based on a set of parameters. According to [113] there are mainly three price formation models: *cost-based*, *demand-driven* and *competition-oriented*. The cost-based price formation model establishes the price using the provider cost for operating the service which may include expenses of the used resources and third party services. The demand-driven price formation model establishes the price based on the consumers demand of the service in the current market. Finally the competition-oriented price formation model establishes the price based on competitors prices, thus competitive providers propose equivalent services with attractive prices to get a large market share.

The billing model determines how consumers pay for using a service. Service providers apply mainly two billing models: *pay-as-you-go* and *subscription*. The pay-as-you-go billing model accounts the service price according to the actual consumer's usage of the service. The service usage is accounted using service-specific units. Well

known examples of services provided in a pay-you-go model include the electric energy and the telephony, where consumers usually pay a predefined price per respectively consumed kWh or seconds of voice transfer. The subscription billing model gives consumers an unlimited access to the service or a maximum service usage value while demanding a periodical payment of a predefined price. For example, many mobile telephony suppliers propose monthly subscriptions for an unlimited access to calls to mobiles in a same country. Usually, the subscription billing model is more advantageous for the consumers than the pay-as-you-go billing model if the service is used more than a specific usage threshold and vice-versa.

2.3.5 Penalties

Penalties may be enforced against both providers and consumers if one of them does not fulfill the agreement. If it is the customer that does not satisfy the agreement, she may for example be forced to pay additional fees or undergo a deterioration in service quality. If it is the provider that fails in delivering the agreed service quality, it may be forced to refund the affected customers. The refund is usually in the form of service credit rather than a real monetary refund because the latter may seriously compromise the provider profit.

2.3.6 SLA Lifecycle

The SLA lifecycle proposed in the literature [98][152] includes six phases: (1) *definition*, (2) *discovery and negotiation*, (3) *establishment*, (4) *execution and monitoring*, (5) *termination*, and (6) *assessment and enforcement* phases. In the first phase the provider develops the service and defines its corresponding SLA terms which involve technical and legal aspects. SLA terms include a minimum and/or maximum values of SLA metrics as well as associated billing and penalty policies. In the second phase, the consumer discovers the service provider and negotiates the SLA terms until a mutual agreement between the two parties. The third phase consists in the establishment of the agreement and the deployment of the service for the consumer. The fourth phase consists in the execution of the service and the monitoring of the values of each SLA metric over all the service's execution time. In the fifth phase, the SLA terminates following the completion of the service or a violation of SLA terms from any party. In the sixth phase the monitored values of SLA metrics are assessed and if any party violates the contract terms the corresponding penalties are enforced. The sixth phase may operate in parallel with the fourth phase and the possible SLA violation penalties are enforced immediately. Otherwise it operates at the end of the fifth phase and the possible SLA violation penalties are enforced at the end of the service execution.

2.4 Cloud Computing

2.4.1 Overview

Several cloud computing definitions and features have been proposed in the literature [145][95][135][57]. Most definitions converge on describing the cloud computing as a new computing paradigm for providing on-demand software and hardware resources

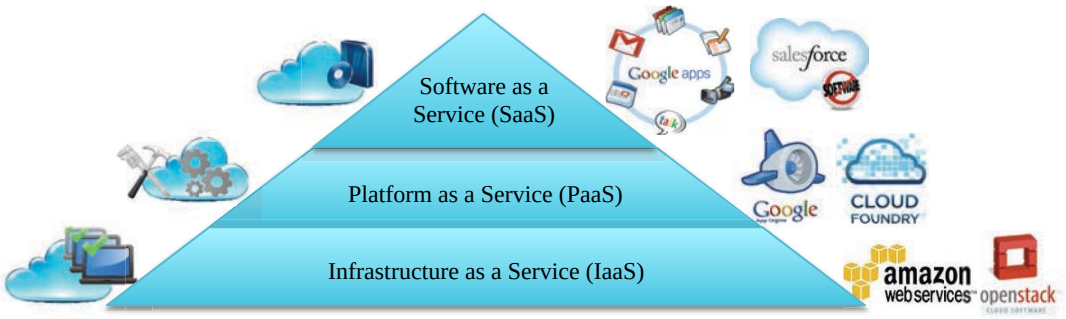


Figure 2.4: Cloud service models

as services over a network (e.g., Internet) in a pay-as-you-go pricing model. The main cloud computing features are: *elasticity, on-demand self-service, pay-as-you-go pricing model, availability,* and *multi-tenancy*.

Elasticity. Clouds appear to customers as an infinite pool of computing resources (e.g., compute and storage capacity) that may be purchased at any quantity and at any time. Thus, customers can easily scale up and down their resource provisioning to meet changes in their applications and workloads requirements.

On-demand self-service. Customers can instantly provision computing resources without requiring human interaction with the cloud provider. This feature is strongly related to the elasticity feature but it implicitly requires autonomic resource management software to enable an on-time reaction to new resource provision requests as well as changes in existing resource provisions.

Pay-as-you-go pricing model. Cloud services are measured using predefined units (e.g., resources consumption, QoS requirement) that depend on the type of the provided service. Thereby, the cloud services are monitored, controlled and reported to both the provider and customer in order to establish a billing according to the actual service consumption.

Availability. Cloud services are almost constantly available over the networked and accessible through standard network devices (e.g., laptops, servers, and PDAs).

Multi-tenancy. Cloud resources are pooled to be assigned to multiple customers in a multi-tenant model, where the exact location of customer's data and/or code is unknown. This implies a high privacy issue in cloud systems.

2.4.2 Service Models

The cloud computing services are commonly classified in three fundamental delivery models [135][95][156]: *Infrastructure as a Service (IaaS), Platform as a Service (Paas),* and *Software as a Service (SaaS).* Each service model has its own specific properties. Figure 2.4 visualizes the three cloud service models in a pyramid.

2.4.2.1 Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) is the base cloud service model in the pyramid. It delivers fundamental compute resources (computing, storage, and network). The IaaS model relies on the virtualization technology in order to hide the complexity of the underling

physical resources and expose a flexible virtual execution environment to customers. The virtualization allows the creation of a pool of virtual computing, storage and network resources on top of physical resources using specific virtualization tools. Virtual computing resources, commonly called Virtual Machines (VMs), represent full computers containing hardware components and running an operating system in the same way as real computers [137]. Virtual machines are obtained by partitioning physical machines using a hypervisor [91] such as Xen [61], KVM [110], VMware Workstation [38], and VirtualBox [149]. Virtual storage resources are obtained through the pooling of multiple data pieces stored in multiple physical storage devices and appear to customers as a single storage device (e.g., disk drive, folder, etc.) [150][132]. Examples of storage virtualization systems include Hitachi Virtual Storage Platform (VSP) [12], IBM SAN Volume Controller (SVC) [23], and DataCore SANsymphony [33]. Virtual network resources consist in separate logical networks that have their own topology and IP addressing scheme while sharing a common or multiple physical network infrastructures [70][74]. Examples of network virtualization software include VMware NSX [43], IPOP [88], CISCO Easy Virtual Network (EVN) [47], and VIOLIN [102].

IaaS customers are able to build customized and elastic execution platforms on-demand from the provided IaaS resources in order to fit their requirements. The elasticity in the IaaS model may be either vertical or horizontal, where the vertical elasticity refers to the modification of virtual machines capacities and the horizontal elasticity refers to the modification of the amount of virtual machines. The customers have control and administrative rights over operating systems installed in the rented virtual machines and are able to install, configure and operate their own software (e.g., applications, libraries, etc.). Moreover, they have the possibility to delete, save, stop, and restart their own environments as required. For instance, if a consumer does not use continuously his/her environment he/she may stop it during the no-usage period and restart it later in order to limit her expenses. On the other hand, IaaS providers may migrate virtual machines from one physical machine to another one for maintenance reasons or for consolidating virtual machines during low demand periods in order to save energy and/or cost of their infrastructure.

Today, there is a growing number of commercial IaaS providers, the main important industry players are Amazon (EC2) [54], Rackspace [2], Microsoft (Windows Azure Infrastructure) [1], and Google (Compute Engine) [49]. Furthermore, several open source and commercial solutions for building and managing IaaS cloud services have emerged. Open source solutions include Nimbus [108], OpenNebula [123], OpenStack [5], CloudStack [4], and Snooze [86] and commercial solutions include Flexiant [3], Microsoft private cloud [6], Red Hat CloudForms [7], IBM SmartCloud Orchestrator [8], and HP CloudSystem [9].

2.4.2.2 Platform as a Service (PaaS)

Platform as a Service (PaaS) is the middle cloud service model in the pyramid. It delivers a complete runtime environment for building, deploying and hosting software applications typically including operating systems, software libraries, storage and database services. The PaaS model relies on computational resources in the form of IaaS resources or operating system containers. On one hand, relying on virtual resources from IaaS clouds brings a high degree of flexibility and multi-tenancy to the PaaS, but at the

cost of being dependent on IaaS and virtualization layers. On the other hand, relying on operating system containers enables a better utilization of resources, but at the cost of being constrained to host only applications running on the same operating system. Therefore, the choice of computational resources on which the PaaS relies depends on the PaaS's goals and requirements.

PaaS customers deploy their applications on the PaaS environment without the complexity of managing the underlying software and hardware required by their applications. PaaS environments are usually designed to support the execution of specific application types (e.g., web services) developed using specific aprogram languages (e.g., Python, Java) and libraries (e.g., Apache libraries). Moreover, because of a lack of standardized interfaces, each PaaS provider exposes its own API to enable customers to use its services. Thus, customers have to carefully choose their providers according to the supported programming languages and the provided tools for building, running and managing their applications because they are often subject to vendor lock-in.

Most of prominent IaaS providers propose also PaaS services and run the hosted applications on their infrastructures. For instance, Amazon provides Elastic Beanstalk [10] for web-applications developed in familiar languages, such as Java, Python, Ruby, etc. and Elastic MapReduce [11] for data-intensive applications. Microsoft provides Windows Azure Cloud Services [13] for multi-tier web-applications. Google provides App Engine [14] for web-applications developed in Java, Python, Go, or PHP. Other commercial providers provide only PaaS services and host applications either on resources from other IaaS providers or on their own infrastructures. For instance, Heroku [15] hosts applications on resources from Amazon and Salesforce's platform (Force.com) [16] hosts applications on the Salesforce's infrastructure. Similar to IaaS clouds there are several open source and commercial solutions for building and managing PaaS cloud services. Open source solutions include ConPaaS [130], AppScale [17], Cloudify [18], and WSO2 Stratos [20]. Commercial solutions, also known as cloud managers, include RightScale [21], Scalr [22], and Dell Cloud Manager [24]. In addition, some companies provide PaaS solutions as well as hosted instances of their solutions, such as RedHat OpenShift [25] and VMware Cloud Foundry [19].

2.4.2.3 Software as a Service (SaaS)

Software as a Service (SaaS) is the highest cloud service model in the pyramid. It delivers specific software applications with specific capabilities accessible through a client interface such as a web browser (e.g., business functions, games, web-mails, etc.). The provided applications are hosted on the provider's infrastructure or on another cloud infrastructure or platform. SaaS customers directly access and use the software applications without having to acquire the software licenses or to configure the underlying infrastructure and hosting environment. However, customers have only a limited control on applications configuration where providers have full control and are in charge of installing, operating, updating and managing the applications. Examples of SaaS providers include Salesforce CRM [27], iCloud [28] and Google Apps [26] such as Gmail, Gtalk, and Google Agenda.

2.4.3 Deployment Models

Cloud computing services are also classified according to their deployment models. The are mainly three deployment models: *public cloud*, *private cloud*, and *multi-cloud*. In the following we present each deployment model and describe some existing instances of the multi-cloud deployment model.

2.4.3.1 Public Cloud

A public cloud is a commercial cloud owned by a business organization, known as *public cloud provider*, and available to the general public, individuals or organizations. Public cloud providers implement a payment method (e.g., automatic bank card debit) to bill customers their resource and service usage. Thus, customers do not invest in hardware equipments and pay only for what they really use.

2.4.3.2 Private Cloud

A private cloud is a cloud owned, managed and used by a single organization (e.g., company, academic institution). It is deployed on the organization's datacenter using an internal, open source or commercial solution and controlled by the organization's staff. The exposed cloud services and functionalities match the specific requirements of the organization. Usually, private cloud customers do not pay for using the services but some usage rules may be applied such as quotas and charters.

2.4.3.3 Multi-Cloud

A multi-cloud deployment model refers to the use of two or more cloud services (either private or public), where each cloud service is provided by a different entity. There are many instances of the multi-cloud deployment model. In the following we describe the most common of them.

- **Community Cloud.** A community cloud is an aggregation of multiple private or public clouds provided and shared by several organizations with common concerns. Community clouds share some similarities with grids in terms of resource sharing between multiple organizations. The community cloud is managed by the community or a third party and is used exclusively by the community members, following specific terms of use. For example a community cloud may be designed for academic organizations working on a joint project.

- **Hybrid Cloud.** A hybrid cloud is a cloud composed of at least one private cloud and one public cloud, which are bound together to form a unique cloud. This deployment model offers more flexibility and takes advantage at the same time of the private cloud in terms of control and security of sensitive services, and of the public cloud in terms of on-demand expansion during high load periods. A hybrid cloud may be built for either private or commercial purposes.

- **Cloud Bursting.** The cloud bursting deployment model can be classified as an instance of the hybrid cloud deployment model. Specifically, it is a deployment model based on private resources, in the form of cluster, grid or private cloud,

that may burst into public cloud resources on-demand, mainly to address the lack of resources when the demand for computing increases. The cloud bursting approach is an appealing intermediate solution between static provisioning of local computing resources and entirely outsourcing applications on public clouds.

- **Cloud Federation.** A cloud federation is the union of several cloud services, private or public, to form a single cloud. The cloud federation provides interoperability among many cloud providers and is responsible for mapping clients requests on the best cloud provider according to the requirements and the type of the request.

- **Cloud Broker.** A cloud broker is a third party entity that acts as an intermediary between cloud computing customers and cloud computing providers. The role of a cloud broker is either to assist customers to select the cloud computing provider based on their requirements, or to facilitate the distribution of customers requests between different cloud service providers.

2.4.4 Business Aspects

Cloud providers spend a lot of money to provide and maintain their services and in return they get payments from consumers using their services. In this document we call provider expenses *"costs"* and consumer payments *"revenues"*. To provide a global view of cloud providers business models we present in the following a non-exhaustive list of both cloud provider costs and revenues. Then, we briefly present characteristics and limitations of existing cloud SLAs of the main commercial cloud providers.

2.4.4.1 Costs

The cost of cloud computing services comprises multiple factors, already identified in many research works [57][78][107][140][143][127]. We classify the cloud cost factors in five categories: *acquisition*, *upgrading*, *electricity consumption*, *support and maintenance* and *third-party services*.

Acquisition. This category refers to the acquisition of software licenses and hardware resources. Software licenses cover basic system software, internal cloud system management software and application software used by final consumers. Hardware resources include computing hardware (e.g., servers, storage disks), needed network devices (e.g., routers, switches), plus the required cooling infrastructure.

Upgrading. The economic lifetime of the acquired software licenses and hardware resources is limited. Moreover, software updates and hardware improvements are regularly released. Thus, it is necessary for cloud providers to upgrade regularly their owned software licenses and computing hardware resources.

Electricity consumption. This category considers the consumed electricity in the entire datacenter including electric devices (e.g., computing hardware and cooling infrastructure) and premises lighting. The cost of the electricity consumption relies on the price of the electricity which depends on the datacenter location. The electricity consumption of computing resources varies according to their utilization. Usually, idle resources consume less electricity than heavily utilized ones.

Support and maintenance. This category comprises salaries of employees working on administering and managing the cloud system, maintaining software and hardware and providing support to customers.

Third-party services. This category includes expenses for services obtained from third party providers, such as internet network connectivity, premises renting and any additional utilities.

2.4.4.2 Revenues

Cloud providers revenues are determined based on three aspects: services prices, services utilization, and the providers ability to satisfy the promised service qualities. Service prices are determined using the used price formation model and the used billing model. Usually, with the subscription pricing model long contract periods imply cheaper service prices, which is profitable for both service providers and consumers [136]. And with the pay-as-you-go pricing model services have always same prices per time unit, but the payment time unit varies from one provider to another one. Initially all providers were charging the resource consumption of their consumers per hour but recently some providers started to charge their customer's resource consumption per minute, such as Google App Engine and Windows Azure Cloud Services.

Cloud customers pay, directly or indirectly, for the used computing resources, for providing them with services as well as for the provided services quality level. The computing resources include computing nodes, network bandwidth and storage. Customers may also pay for additional, often optional, services to improve and facilitate their applications execution and management, such as resource utilization monitoring and load balancing tools [116].

2.4.4.3 SLA

Cloud providers propose several services with different qualities to customers and also propose separate SLAs for each service. However, current cloud computing offerings provide a limited SLA penalty management support. Most of providers, including big players, focus on services availability and uptime. Table 7.1 provides examples of existing cloud SLA metrics practiced by respectively Amazon EC2, Google Compute Engine and Microsoft Azure. If the providers fail to ensure the promised service availability, they propose to credit the user with a percentage of the bill for the eligible credit period. However, their mechanisms are not automatic. The user should provide logs that document the errors and send a request through an Email or the customer contact support. Moreover, to benefit from SLA guarantees some providers impose specific conditions. For instance, to benefit from a minimum uptime of 99.9% in Windows Azure Cloud Services requires to have at least two compute node instances.

2.5 Summary

In this chapter we presented the background of this thesis. We started with a brief description of existing IT infrastructures and their respective system software. We also introduced the utility computing paradigm and gave a taxonomy of the different software

Table 2.1: Examples of existing cloud SLA metrics

	Metrics	Values	Penalties	Enforcement
Amazon EC2	Availability	99.95% 99%	10% service credit 30% service credit	User's request
Google Compute Engine	Uptime	99.95% 99% 95%	10% service credit 25% service credit 50% service credit	User's request
Microsoft Azure	Uptime	99.95% 99%	10% service credit 25% service credit	User's request

application models. We classified software applications according to their parallelism model, resource requirements, resource usage, accommodation and interactivity.

Then, we presented the Service Level Agreement (SLA) concept which consists in a formal contract between a service provider and a consumer. The SLA describes the concerned services as well as obligations and penalties of both the provider and the consumer. We described the SLA lifecycle, from its definition to its termination and presented the main existing SLA specification languages. For the sake of clarity, we highlighted the differences between functional and nonfunctional SLA metrics as well as the different SLA pricing functions and models. We also briefly described penalty enforcement forms currently practiced by service providers.

Finally, we presented the cloud computing technology and described its three service model layers: IaaS, PaaS and SaaS. For each service model we described the used technology and mechanisms, the exposed form of services to users and the main existing providers and solutions. We also described the four cloud deployment models: public, private, community and hybrid. Furthermore, we presented cloud computing business aspects in terms of providers costs, revenues and existing cloud SLAs of the main cloud providers.

In the next chapter, we describe the objectives of this thesis, discuss the related work, highlight the existing gaps and briefly introduce our contributions and show how we fill the gaps.

Chapter 3

SLA-Based Profit Optimization in Cloud Bursting PaaS

Contents

The objective of this PhD thesis is to provide a profit-optimization solution for SLA-enabled, cloud-bursting, PaaS systems. In this chapter, we specify the main required features to achieve this objective and review the most recent proposed approaches in three related fields: Platform as a Service (PaaS) systems, cloud-bursting environments, and economic optimization policies. We highlight the limitations of the main related work that prevent achieving the defined objectives. Finally, we summarize this chapter.

3.1 Thesis Objective

The main objective of this thesis is to provide a profit-optimization solution for SLA-enabled, cloud-bursting, PaaS systems. To achieve this objective, the solution must support the following two features: a PaaS hosting environment and a profit-making business model.

- **Profit-Making Business Model.** The required profit-efficient business model should take into account three main properties.
  1. *Generic.* It should be generic and independent from specific application types.
  2. *Support for SLA constraints.* It should take into account SLA constraints and the payment of penalties if the QoS promised to applications is not satisfied.
  3. *Applicability to PaaS environments.* It should be applicable on a cloud-bursting PaaS environment.
- **PaaS Hosting Environment.** The required PaaS hosting environment should provide a number of properties to enable the implementation of a profit-making business model for PaaS providers. In the following we identify two properties.
  1. *Open.* An open PaaS hosting environment should be easily extensible to host a large variety of application types. This is necessary in order to attract users having different requirements, thus increasing providers' turnovers.
  2. *Cloud bursting.* The cloud bursting feature is required in a PaaS hosting environment in order to avoid rejecting clients requests in peak periods. Moreover, this provides several options to the PaaS provider for selecting the resources that host applications.

Based on these requirements we review in the next sections three main fields: PaaS systems, cloud bursting environments, and economic optimization policies.

3.2 Platform as a Service (PaaS)

Nowadays the number of emerging Platform as a Service (PaaS) cloud systems is increasingly growing. Such systems aim at providing users with complete hosting environments to deploy and manage their applications while shielding them from managing the underlying resources. We classify the PaaS solutions in three categories: commercial, open source and research. In this section, we review some existing PaaS solutions from each category according to several characteristics: resources, services, applications, languages, architectures, billing and SLA. The *resources* characteristic represents the underlying infrastructure or computing nodes that may be used by the PaaS. The *service* characteristics describe the provided features for executing applications. The *applications* characteristic determines the application types supported by the considered PaaS. The *languages* characteristic represents the program languages supported in the PaaS. The *architecture* characteristic describes how the underlying resources are partitioned to be used for running an application. The *billing* characteristic shows the used pricing model which is often pay-as-you-go or subscription model. The *SLA* characteristic describes the promised service quality levels if any.

3.2.1 Commercial PaaS

In this section, we review the main current commercial PaaS solutions and summarize them in Table 3.1.

Google App Engine. Google App Engine (GAE) [14] is a PaaS designed and provided by Google for building and hosting distributed web applications. The supported applications may be interactive, modeled on processing operations in a task queue, or combining the two. GAE relies on the Google infrastructure [49], provides an automatic scalability for applications and supports four programming languages Java, Python, PHP and Go and three options for storing data: App Engine Datastore, Google Cloud SQL and Google Cloud Storage. GAE provides an App Engine software development kits (SDKs) for each supported language to emulate all of the App Engine services on users' local computers. To host applications, users create the corresponding code and configuration files and upload them to the App Engine. The execution of each App Engine application is charged according to its resources and services usage. It is possible to specify a maximum daily budget for each application to better control expenses. The GAE's SLA consists in providing to customers a monthly service's uptime percentage at least 99.95%. If Google does not meet the SLA, customers will be eligible to receive a service credit of 10%, 25%, or 50% depending on if the monthly service uptime is respectively less than 99.95%, 99% or 95%. However, this compensation is not automatically assigned to customers. They are responsible for keeping logs to request and justify their claim. Further, the service credit is applied to future service use within only 60 days after the request.

Windows Azure Cloud Services. Windows Azure Cloud Services (Azure) [1] is a PaaS designed and provided by Microsoft for building and hosting multi-tier web applications, also called cloud services. The cloud services are hosted on top of the Windows Azure IaaS resources where Windows Azure handles the resources provisioning and load balancing while enabling users to configure their applications to automatically scale up or down and match current demands. A cloud service is defined in two roles: a web role and a worker role. The web role provides a dedicated Internet Information Services (IIS) server used for handling web-based requests and hosting web applications front-end. The worker role hosts the long-running or perpetual tasks of the web applications that are independent of user interaction or input. The creation of a cloud service requires specific packages created using the Windows Azure SDKs or Visual Studio. There are six supported programming languages: .NET, Node.js, PHP, Python, Java, and Ruby and three types of storage services: SQL Database, Tables or Blobs. It is possible to manage hosted applications either through the Windows Azure dashboard or through a command line interface. The Azure's SLA guarantees a monthly external connectivity to the Internet facing roles at least 99.95% of the time, with the condition of deploying at least two or more web role instances. The monthly connectivity uptime percentage is calculated as the total number of maximum connectivity minutes less connectivity downtime divided by maximum connectivity minutes for a giving month. If the SLA guarantee is not met Microsoft undertakes to assign to customers 10% or 25% of service credit depending if the monthly connectivity uptime percentage is respectively less than 99.95% or 99%. To benefit from this compensation, customers have to contact the Customer Support and provide reasonable details regarding the claim. The claim should be submitted by the end of the billing month where the incident occurred.

Amazon Web Services. Amazon Web Services (AWS) provides two PaaS systems: Elastic Beanstalk [10] and Elastic MapReduce (Amazon EMR) [11]. Elastic Beanstalk is designed for deploying and hosting web applications. It enables the load balancing, automatic scaling and the management of applications in the AWS cloud. Moreover, it gives users full control over the AWS resources powering their applications. For instance, customers may specify the number and the type of virtual machine instances and decide to replicate the application on multiple geographic zones. AWS Beanstalk supports six programming languages: Java, PHP, Python, Node.js, Ruby and .NET and uses Amazon S3 for the data storage. Beanstalk's clients are charged based on the used resources to store and run their applications without any additional charge for the Elastic Beanstalk. Elastic MapReduce (Amazon EMR) [11] is designed for processing data-intensive tasks, specifically MapReduce applications. Amazon EMR utilizes a hosted Hadoop framework [151] on the Amazon infrastructure using Amazon EC2, Amazon S3 and Amazon SimpleDB services and supports seven programming languages: Java, Perl, Python, Ruby, C++, PHP and R. To process their data, customers create a resizable cluster and launch it without the need to worry about the cluster setup and Hadoop configuration. As opposed to Beanstalk, customers using EMR pay an extra charge in addition to normal Amazon EC2 and Amazon S3 pricing. Amazon provides no SLA for its PaaS services but applies the SLAs of the IaaS services composing them. For instance, the Amazon EC2's SLA consists in making the EC2 resources available to external connectivity at least 99.95% per month. Otherwise, affected clients receive 10% or 30% service credit of the billing of the unavailability period depending if the availability percentage is respectively less than 99.95% or 99%. To receive the service credit, customers must submit a claim to the AWS support center by the end of the second billing cycle after which the incident occurred.

Force.com. Force.com [16] is a PaaS, provided by Salesforce, for creating and deploying business web applications on the Salesforce's infrastructure. It uses a multi-tenant architecture where all users share the same infrastructure which allows a lower operational cost. The used application development model is metadata-driven where developers build applications functionalities with tools provided by the platform. This helps developers becoming more productive compared to the hard-coded development model. In addition, Force.com provides three APIs to create a more customized applications behaviors: SOAP, REST and Bulk. These APIs may be called from a wide variety of programming languages that support web services, such as Java, PHP, C#, or .NET. The use of Force.com requires an annual contract where users pay monthly a predefined invoice depending on the application characteristics and capabilities. To the best of our knowledge Force.com does not provide a compensation if something goes wrong

Heroku. Heroku [15] is a PaaS for web applications, initially founded by a number of engineers then acquired by the Salesforce company. Heroku is designed based on isolated and virtualized Unix containers, called *dynos*, that provide the runtime environment required for running applications. Heroku utilizes a process-based execution model for running the applications. It separately and dynamically assigns dynos to the hosted applications and uses a *dyno manager* that provides mechanisms analogous to ones provided by traditional OS process managers. To build and run an application, Heroku's customers should provide the corresponding source code and dependency files written in one of the five supported programming languages: Ruby, Node.js, Python, Java Clojure and Scala. Based on the provided files, Heroku generates

a runnable ready for execution. Heroku calculates billing based on dynos and database usage per application. Heroku provides four levels of the database uptime expectation per month: hobby, standard, premium and enterprise. The database downtime tolerance is up to 4 hours for the hobby tier, 1 hour for the standard tier and 15 minutes for the premium and enterprise tiers. However, any measure of compensation is specified if Heroku fails to achieve the promised database uptime.

Engine Yard. Engine Yard [32] is a PaaS designed for building, deploying and managing web-applications written in one of the three supported programming languages: Ruby on Rails, PHP and Node.js. It relies on the Amazon cloud infrastructure where it configures and manages multiple software components depending on applications particular needs. Engine Yard is responsible of VMs configuration and imaging, automatic scaling, administering databases and load balancing and performing backups. In addition, it enables customers to have control over VM instances. Engine Yard charges customers based on resource usage of their applications and promises, in its SLA, a system availability of 99.9% for a month. Otherwise, customers shall be entitled to a service credit equivalent to the amount of unscheduled service downtime in excess of 0.1%, divided by the amount of scheduled service uptime for that month, and multiplied by the total recurring fee for that month for the affected services payable by customer. However, the maximum cumulative credit in a month is 50% of the total recurring fee for that month for the affected services.

RightScale. RightScale [21] is a cloud management platform that provides abstraction for managing a set of cloud infrastructures on behalf customers. It enables interoperability among the supported public clouds: Amazon AWS [54], Datapipe [37], Google Compute Engine (GCE) [49], HP Cloud [9], Rackspace [2] and Windows Azure (WA) [1], and private clouds: CloudStack [4] and OpenStack [5]. It enables customers to easily run and manage their applications and development environments while providing them freedom to choose or mix vendors and technologies. RightScale provides automation techniques and enables customers to make their operations scalable while giving them visibility and control over the used IaaS resources. It delivers the necessary tools to create runtime environments for HPC (High Performance Computing) tasks, big data analytics, web, gaming and mobile applications. RightScale supports a range of languages and frameworks including Java, .NET, PHP, Python Django, and Ruby on Rails. For the pricing, RightScale proposes a range of customized subscription contracts where some services are charged according to the usage. No compensation rules are specified in the RightScale's website if the services are not correctly provided.

CloudBees. CloudBees [34] is a PaaS for building, hosting and managing web applications on the cloud. The CloudBees PaaS is hosted on the Amazon cloud infrastructure. It configures, scales (manually or automatically) and assigns the Amazon resources to customers' applications. CloudBees supports a set of well known tools for applications development such as Eclipse, and also proposes its own SKD. The main supported programming language in CloudBees is Java, but there are also other languages such as Node.js, PHP and JavaScript. CloudBees proposes several pricing models. Some services are charged based on subscriptions such as databases and applications development. Other services are made available on a pay-as-you-go basis. For example, the application execution is charged according to the dedicated VM instances or the consumed *app-cells*, where an app-cell is the basic unit for running an application on CloudBees and is more finely grained than the Amazon instance types. The users pay-

ment also depends on the a committed level of response and problem resolution time that varies from two days to four hours. No compensation measures are specified if the response time is longer than expected.

AppFog. AppFog [36] is a PaaS for web applications built on CloudFoundry, the open source PaaS of VMware described in section 3.2.2. AppFog provides a multi-cloud deployment and scalability options. It may run on Amazon, HP, Microsoft Azure and Rackspace clouds as well as on a private cloud. AppFog supports many web application runtimes: PHP, Node.js, Ruby, Python, Java, and .NET and provides a number of database services. To create and manage applications, AppFrog provides a graphical user interface and command line tools. AppFog uses memory as the basis for establishing different service levels and the corresponding subscription contracts. To the best of our knowledge AppFog provides no compensation if something goes wrong.

dotCloud. dotCloud [35] is a PaaS for deploying, managing and scaling web applications where an application is considered as a collection of one or more services working together. Each service includes specific support for a fixed program or data storage language. dotCloud runs on the Amazon EC2 resources where it uses a container system, called *Docker*, for allocating resources to the hosted applications. Docker [45] is an open source engine based on the Linux container runtime to automate applications deployment and execution. dotCloud is responsible for automatically configuring Docker on EC2 instances and applying scaling strategies specific to each application. dotCloud provides a Command Line Interface (CLI) to be installed on customers computers for deploying and managing applications. The supported programming languages on dotCloud are: Java, Perl, PHP, Nodes.js, Python and Ruby. The price of applications hosted on dotCloud is based on the total allocated memory per hour. dotCloud provides two SLA support levels: expedited and live. The promised applications response time is less than 30 minutes for the expedited level and less than one hour for the live level. However, no compensation is considered if the SLA is not met.

We observe that most of commercial PaaS solutions have mainly three limitations: (1) relying on specific IaaS providers, (2) focusing only on web applications and (3) having a very limited support of SLA. Moreover, all of the reviewed commercial solutions charge their users according to their resources consumption rather than service quality.

Table 3.1: Summary of commercia PaaS solutions

PaaS	Resources	Services	Applications	Languages	Architecture	Billing	SLA
GAE [14]	Google infrastructure	Building, hosting, data storage, auto scaling	Web	Java, PHP, Python, Go	Unknown	Pay-as-you-go	Monthly uptime percentage $\geq$ 99.95%
Azure [1]	Windows Azure infrastructure	Building, hosting, load balancing, auto scaling, data storage	Multi-tier web	.NET, Ruby, PHP, Python, Java, Node.js	Dedicated VMs per app	Pay-as-you-go, subscription	Monthly connectivity uptime $\geq$ 99.95%
AWS Elastic Beanstalk [10]	Amazon Infrastructure	Hosting, load balancing, auto scaling, data storage	Web	Java, Node.js Python, PHP Ruby, .NET	Dedicated VMs per app	Pay-as-you-go	IaaS resources availability $\geq$ 99.95%
Amazon EMR [11]	Amazon Infrastructure	Hosting, scaling, data storage	MapReduce	Java, Perl, R, Python, PHP, C++, Ruby	Hadoop cluster [151]		
Force.com [16]	Salesforce infrastructure	Building, hosting, data storage	Web	Metadata APIs: Bulk, REST, SOAP	Multi-tenant	Subscription	No compensation
Heroku [15]	Amazon infrastructure	Building, hosting, auto scaling, data storage	Web	Ruby, Python, Node.js, Java, Clojure, Scala	Dedicated dynos: Unix containers	Pay-as-you-go	Data base downtime $\leq$ 4hours - 15 minutes
Engine Yard [32]	Amazon infrastructure	Building, hosting, auto scaling, load balancing, data storage	Web	Ruby on Rails, PHP, Node.js	Dedicated VMs per app	Pay-as-you-go	Monthly system availability $\geq$ 99.9%
RightScale [21]	AWS, Datapipe, GCE, OpenStack, Rackspace, WA,	Building, hosting, auto scaling, data storage,	HPC, web, mobile, games, big da-	Java, Python, Ruby, .NET, PHP	Dedicated VMs per app	Pay as you go, subscription	Not mentioned

	HP, CloudStack,		ta analytics				
CloudBees [34]	Amazon infrastructure	Building, hosting, scaling, data storage	Web	Java, Node.js, JavaScript, PHP	Dedicated VMs or units (app-cells)	Pay as you go, subscription	Response time < 2 days - 4 hours
AppFog [36]	Amazon, Azure HP, Rackspace private cloud	Hosting, scaling, data storage	Web	PHP, Node.js Ruby, Python Java, .NET	Cloud Foundry	Subscription	No compensation
dotCloud [35]	Amazon infrastructure	Building, hosting, load balancing, scaling, data store	Web	Java, Perl PHP, Node.js Python, Ruby	Dedicated containers (Docker)	Pay as you go	Response < 30 minutes - 1 hour

3.2.2 Open Source PaaS

In this section, we review the main well-known open source PaaS solutions and summarize them in Table 3.2.

Cloud Foundry. Cloud Foundry [19] is an industry open source PaaS developed by VMware for deploying and running web applications written in Java, Groovy, Scala, Ruby, PHP, Python or Node.js, using the Spring, Grails, Lift, or Play frameworks. Cloud Foundry provides different services for storing data and enables an horizontal scalability of the services required by applications. Cloud foundry may be deployed on Amazon AWS, WMware vSphere, vCloud Director or OpenStack. The Cloud Foundry architecture uses Linux containers in the virtual machines obtained from an IaaS cloud in order to create isolated environments. These containers can be limited in terms of CPU usage, memory usage, disk usage, and network access. Each container is managed and controlled using a component called *Warden container* running in the same virtual machine. *Droplet Execution Agent (DEA)* components are also hosted on the virtual machines for managing Warden containers and assigning applications to them. The DEA manages the entire lifecycle of all its applications and periodically broadcasts the applications state to a Health Manager. The Health Manager is responsible for reconciling applications' current state (e.g. running, stopped, crashed) and their expected state. If the reconciliation fails the Health Manager directs the *Cloud Controller* to take actions. The Cloud Controller maintains a database with tables for applications, services, DEAs, etc. When a new application is submitted the Cloud Controller selects a DEA from the pool of available DEAs to stage the application. A Micro Cloud Foundry is provided as a virtual machine image that can be deployed on a laptop to help developers running a local instance of Cloud Foundry. Also, there is a public instance of Cloud Foundry operated by Pivotal[1] and hosted on the Amazon infrastructure to enable an online deployment of web-applications. The Hosted Cloud Foundry instance uses a price formation model based on the memory usage per application and the container hosting the application is configured accordingly. However, no compensation is provided if the services are not correctly delivered.

ConPaaS. ConPaaS [130] is an open source PaaS, developed in the framework of the EU FP7 Contrail project. It is designed to leverage cloud computing resources from a wide variety of public and private IaaS clouds in order to provide a flexible and scalable hosting environment for high-performance scientific applications as well as online web applications. The current ConPaaS implementation is compatible with Amazon EC2 and OpenNebula infrastructures. It provides data storage services, support for hosting web applications written in Java or PHP and high-performance computing environments including Hadoop MapReduce cluster and bag of tasks scheduler services. Each service is deployed on one or more VMs and each application is organized as a collection of services. The services can be scaled on demand to adapt the number of computing resources to the capacity required by the application. ConPaaS provides several tools and building blocks for building complex runtime environments and cloud applications. Moreover, it provides two ready-made applications: WordPress and MediaWiki. A hosted instance of ConPaaS is available[2] offering a free trial feature with a credit system. Users are given a number of credits where one credit corresponds to one

[1] run.pivotal.io

[2] https://online.conpaas.eu/

hour of execution of one virtual machine. However, no SLA is provided.

OpenShift. OpenShift [25] is the RedHat PaaS offered in three forms: Online, Enterprise and Origin. OpenShift Online is a public PaaS hosted by RedHat on top of the Amazon infrastructure. OpenShift Enterprise is a private PaaS designed to run within organizations' data center. OpenShift Origin is the open source software underlying OpenShift Online and OpenShift Enterprise. It is able to run on top of OpenStack, AWS, or in a data center on top of KVM or VMware. It may run also in a personal laptop on top of Virtual Box or on top of unvirtualized Linux hosts. OpenShift allows developers developing, hosting, and scaling their web applications written either in Java, Ruby, PHP, Python, Node.js or Perl. In OpenShift, an application is a combination of code, configurations and application components called *cartridges*. Each cartridge provides a specific services to build the application, which may be web framework, database, monitoring service, or connector to external backends. Cartridges are deployed and isolated in one or more containers known as *gears*. A single server may have multiple gears running at the same time and isolated from each other. Each gear is given an allocation of CPU, memory, disk, and network bandwidth using Linux control groups. Overall, in OpenShift the servers hold gears that contain one or more cartridges, and cartridges hosted on one gear belong to the same application. When a new cartridge needs to be added to an application, OpenShift chooses where to deploy it based on the type and needs of the cartridge. The OpenShift Online pricing model is based on the number and types of used gears per hour. However, no SLA is provided.

AppScale. AppScale [17] is an open-source PaaS implementing the Google App Engine APIs for hosting web applications either on the Google Compute Engine or any other supported infrastructure (Eucalyptus, RackSpace, CloudStack, OpenStack and Amazon EC2) or virtualization tool (KVM and VirtualBox). The AppScale software stack is packaged with a single VM image to be deployed on the cloud using one or more instances. AppScale tools are provided to automatically assign roles to each deployed VM instance and accordingly each instance implements one or more AppScale components and services. AppScale provides automatic configuring, deploying, and scaling as well as applications and data backup and migration from one IaaS cloud to another one. AppScale supports the same application runtimes supported in Google App Engine; namely Java, PHP, Python and Go.

Paasmaker. Paasmaker [42] is an open source PaaS designed to be run on a cluster of machines. It can also be run on Amazon EC2 resources or on a single node for development. The Paasmaker architecture consists of three node roles: pacemaker, router and heart. The pacemaker is the master node responsible for coordinating all the activities on the cluster and providing a front end for the users. The router directs incoming requests to the corresponding nodes that can serve them. The heart node is instructed by the pacemaker node to run and manage user applications written in one of the four supported languages: Python, PHP, Node.js and Ruby. An application may be replicated or scaled horizontally using multiple heart nodes.

Cloudify. Cloudify [18] is an enterprise-class open source PaaS for deploying, managing and scaling enterprise-grade web applications. Cloudify can run on a laptop, datacenter or any cloud supporting the JClouds API, particularly the following public and private cloud environments: AWS, Rackspace, Windows Azure, HP, OpenStack, and CloudStack. Cloudify supports many programming languages such as Java, PHP, .NET and Ruby on Rails and provides data storage and auto-scaling services. To de-

ploy applications on Cloudify, no change in their code is required but customers have to describe their needed services and interdependencies using *recipes* components. A recipe describes the execution plans for the application lifecycle and the required virtual machines as well as their images for a chosen cloud. Cloudify auto-installs its agents on each VM to process the recipes and install the corresponding application tiers. It enables customers to define custom application metrics such as availability and performance, to monitor the metrics and to define scaling rules based on those metrics. Cloudify recipes can also describe an SLA for each service as the minimum number of instances Cloudify has to start and maintain. If Cloudify detects that a service's minimum number of service instances are not running, it will attempt to heal by reinstalling and starting the service instances again. If it detects that a service instance's VM has stopped responding, Cloudify provisions a new VM for the service instance. Cloudify does not propose billing methods for using its services.

Tsuru. Tsuru [41] is an open source PaaS for hosting web applications written in Python, PHP, Node.js, Go, Ruby or Java. Tsuru uses either Ubuntu Juju [44] or Docker [45] to deploy the applications. Each application runs on a set of units having all dependencies the application needs to run. A unit may be either an isolated Unix container or a virtual machine, depending on the used provisioner, Docker or Juju. Juju deploys the applications on a data center or any other supported cloud environment such as EC2, CloudStack and Azure and Docker runs the applications on containers. Tsuru enables users to easily add and remove units in order to scale their applications. Tsuru does not provide any billing model but proposes the usage of quotas where a quota defines for each user the maximum number of applications and the maximum number of units that an application may have.

Many open source PaaS solutions rely on more than one IaaS infrastructure but they provide no policies for selecting the IaaS resources to use. Moreover, most of them only focus on web applications.

Table 3.2: Summary of open source PaaS solutions

PaaS	Resources	Services	Applications	Languages	Architecture	Billing	SLA
Cloud Foundry [19]	AWS, OpenStack, vCloud Director, VMware vSphere	Hosting, scaling, data storage	Web	Java, Ruby, PHP Groovy, Scala, Node.js, Python	Dedicated Linux containers	Pay as you go for the hosted instance	No compensation
ConPaaS [130]	Amazon EC2, OpenNebula	Building, hosting, scaling, storage	Web, batch, MapReduce	Java, PHP	Dedicated VMs per app	Pay as you go credit system	Services enforcement
OpenShift [25]	OpenStack, AWS, KVM, VMware, VBox, Linux cluster	Building, hosting scaling, storage	Web	Java, Ruby, Perl, PHP, Python, Node.js	Dedicated containers (Gears)	Pay as you go for OpenShift Online	Not provided
AppScale [17]	GCE, Eucalyptus, KVM, RackSpace, VBox, CloudStack, EC2, OpenStack	Building, hosting, data storage, fault tolerance, auto scaling	Web	Python, Java, Go, PHP	Dedicated VMs per app	Not provided	Not provided
Paasmaker [42]	Cluser, single node, AWS EC2	Hosting, scaling, data storage	Web	PHP, Node.js, Ruby, Python	Dedicated nodes	Not provided	Not provided
Cloudify [18]	AWS, Rackspace WA, OpenStack, HP, CloudStack	Hosting, data storage, auto-scaling	Web	Java, PHP, Ruby, .NET	Dedicated VMs per app	Not provided	Resource requirements
Tsuru [41]	Ubuntu Juju, Docker	Hosting, scaling, data storage	Web	Python, PHP, Node.js, Go, Ruby, Java	Dedicated units: VMs, containers	Not provided	Not provided

3.2.3 Research PaaS

In this section, we present some relevant research PaaS solutions and summarize them in Table 3.3. Note that, the majority of research PaaS works focuses only on specific aspects of the PaaS systems. Thus, not all the PaaS characteristics are provided. In Table 3.3, when either the applications types and programming languages are not a focus of a research PaaS work we just write *"Common"* in the corresponding cell.

MCP. MCP (Multi-Cloud-PaaS) [129] is an architecture for managing elastic applications across multiple cloud providers. MCP replicates the application on multiple clouds and continuously monitors their workloads and resources provisioning. A *load balancer* component routes requests to application instances and dispatches load among the different cloud providers in a round robin fashion. To avoid a single point of failure the MCP architecture is deployed in at least two different cloud providers. The MCP components are deployed on one cloud and replicated on a second a cloud. MCP allocates and deallocates IaaS resources according to the workloads and resources utilization.

PoI. PoI (PaaS on IaaS) [109] is a method for improving the utilization of supersaturated clouds while easily deploying SaaS. Supersaturated clouds are defined as allocating more logical resources than actual physical resources. In this work the authors compare Single Tenant PaaS (ST-PaaS) and Multi Tenant PaaS (MT-PaaS). The ST-PaaS dedicates a VM instance of IaaS to a single tenant user in order to allow applications to be easily installed. The MT-PaaS shares a VM instance of IaaS between multiple tenant users, thus reducing the cost of the cloud. The authors state the relationship between MT-PaaS and ST-PaaS as a trade off where the running cost of MT-PaaS is smaller than that of ST-PaaS but the development cost is higher.

[66] presents a PaaS architecture, developed in the frame of the EU IST IRMOS project, for online interactive multimedia applications. The authors identify requirements and opportunities for new PaaS capabilities to guarantee real-time QoS for interactive applications. They propose an architecture that uses and provisions on-demand virtualized IaaS resources (storage, networking and computing) for application service components. The resources to provision are determined based on specified applications QoS and a predicted user interaction in the form of SLA. The application runtime information is collected and if SLA violations are detected, the provider is notified to trigger corresponding mitigating actions.

CloudServ. CloudServ [154] is a PaaS system dedicated to service-based applications regardless of their programming languages and hosting frameworks. CloudServ consists of a set of PaaS resources able to offer or consume a particular PaaS service. The authors define three types of PaaS resources: *container*, *database* and *router*. The container is the engine that hosts and runs services, the database enables the data storage and the router provides routing protocol between services. These resources are continuously provisioned along with the arrival of new requests. CloudServ deploys a personalized container each time a new service is deployed where a service may be any dependency required by an application.

COSCA. COSCA [104] is a component-based PaaS inspired from the OSGi programing model. It is built on top of Java Virtual Machine (JVM) and runs on each cloud node a layer of cloud extension above the JVM to provide support for user applications. The applications are composed from multiple bundles that can be started, stopped, updated and replicated as OSGi bundles. COSCA provides a virtual OSGi-like container for

each application in order to grant isolation between components of the different applications. The platform allows the loading of new application components at runtime and permits the coexistence of components in different versions. To deploy an application, COSCA users have to provide an application description containing the application execution settings and links to the corresponding application components to be deployed. COSCA keeps track of the service usage in order to check the workload flow. In case of a high workload, COSCA automatically duplicates the services and distributes client requests among them while ensuring load balancing. The client requests distribution is based on a weighted round-robin scheduling where a load-level is assigned to each node. COSCA enables applications to be cloud-independent and easily switch between the use of a local infrastructure and a cloud platform. It also implements a module to collect the applications resources usage to be utilized for the establishment of a pay as you go pricing model. COSCA may run on a cluster or on cloud nodes. No explicit specification about the supported IaaS cloud solution is given by the authors.

Resilin. Resilin [101] is a PaaS system for running MapReduce applications using the Hadoop framework. Resilin provisions a Hadoop cluster and submits MapReduce jobs on behalf the user. The Hadoop cluster interacts with a cloud storage repository to provide input and output data for the MapReduce applications. Resilin supports MapReduce applications written in Java or Python as well as data analysis systems such as Apache Hive and Pig. The main Resilin feature, compared to Amazon EMR, is the possibility to execute MapReduce computations over multiple private, community and public clouds, such as Nimbus, OpenNebula, OpenStack, Eucalyptus and Amazon EC2. Moreover, users have the possibility to select the VM types, the operating system and the Hadoop version they want to use for running their applications. Users are also able to add and remove VMs to their Hadoop cluster during the execution of their jobs. For interoperability reasons, Resilin implements the Amazon EMR API and provides most of its supported features. In order to provide a uniform interface to the most open-source and commercial IaaS clouds, Resilin uses the Apache Libcloud library. Resilin does not provide any support for the pricing and SLA functions.

Merkat. Merkat [76] is a private platform that shares cloud resources between competing applications using a market-based allocation mechanism. In Merkat, each application runs in an autonomous *virtual platform*, defined as a composition of one or multiple *virtual clusters*, a set of *monitors* and an *application controller*. A virtual cluster is defined as a group of VMs having the same disk image and hosting the same application components. The application controller manages the application life-cycle and provides Service Level Objective (SLO) support by applying elasticity rules to scale the application demand according to user performance objectives. Each virtual platform adapts individually, without knowledge regarding other participants. Thus, users can run different applications and express a variety of performance goals. Merkat relies on a market-based resource allocation where users have an amount of credit received from a banking service and distribute it to the virtual platforms running their applications. The assigned credit to a virtual platform reflects the maximum cost for executing the application. Further, the CPU and memory resources have a price set through market policies which fluctuates based the resource demand. Virtual platforms autonomously adapt their applications resource demand to price variations with the aim to meet applications SLO. The Merkat system ensures a fair and maximized resource utilization based on a budget-proportional share policy. Merkat is designed to manage a single

cloud and it current implementation relies on the OpenNebula cloud manager.

Qu4Ds. Qu4Ds [111] is a PaaS-level framework for the applications based on the Master/Worker pattern. The framework includes mechanisms for SLA negotiation, translation, and enforcement and supports both performance and reliability QoS properties. When a new QoS contract is proposed Qu4Ds translates this on resource requirements and configurations able to ensure the QoS and provisions the resources until the end of the QoS contract. Qu4Ds provides a QoS assurance loop that monitors the job's dynamic metrics and handles the possible failures and delays.

Aneka. Aneka [146] is a PaaS framework for deploying multiple application models on the cloud. It supports many program models and languages by creating customizable software containers. The Aneka platform relies on physical or virtualized computing resources connected through a network. Each resource hosts an instance of a container, the Aneka core unit, representing the applications's runtime environment. The container provides the basic management features to a single node and leverages the other operations needed by the hosted services. The services available on the container may be customized and deployed according to the application-specific needs. Also, it is possible to increase on-demand the number of the Aneka framework nodes according to the user needs. Aneka provides a basic resource controlling feature that restricts the number of resources that can be used per deployment or the number of nodes allowed in the Aneka platform.

The reviewed research PaaS works have different focuses but none of them provides the features required to achieve our objectives. Specifically, they do not provide openness, support for cloud bursting and support for provider's profit optimization under SLA constraints.

Table 3.3: Summary of commercial, open source and research PaaS solutions

PaaS	Resources	Services	Applications	Languages	Architecture	Billing	SLA
MCP [129]	Multi-Cloud (WA, EC2, etc.)	Hosting, scaling load-balancing	Elastic	Common	Not specified	Not specified	Not specified
PoI [109]	IaaS	Hosting	SaaS	Common	Single/multi tenant users	Not specified	Not specified
[66]	IaaS resources	Hosting	Interactive multimedia	Common	Dedicated resources	Not specified	QoS assurance
CloudServ [154]	Virtual nodes (OpenNebula)	Hosting, data storage	Service based	Common	Dedicated containers	Not specified	Not specified
COSCA [104]	Cluster, cloud nodes	Hosting, load balancing, auto-scaling	Common	Java	OSGi containers	pay as you go	Not specified
Merkat [76]	single cloud (OpenNebula)	Hosting, scaling, load balancing	Common	Common	Dedicated resources	budget-based resource sharing	Aims at meeting applications SLO
Resilin [101]	Multi-cloud (Libcloud)	Hosting, scaling, data storage	MapReduce	Java, Python, Hive, Pig	Hadoop	Not specified	Not specified
Qu4Ds [111]	Physical/virtualized resources	Hosting, fault tolerance	Master / Worker	Common	Dedicated resources	pay as you go	QoS assurance
Aneka [146]	Physical/virtualized resources	Hosting, scaling, data storage	Web, batch, task framing	.NET, C#, C++, VB, etc.	Dedicated nodes	Not provided	Not provided

3.3 Cloud Bursting

Cloud bursting has been defined for the first time by Amazon's Jeff Barr [39] for running applications on a powerful and high scalable hosting environment that combines enterprises' owned infrastructures and cloud-based infrastructures. Many researchers have been attracted by the cloud bursting approach for its many benefits, such as cost-efficiency, performance optimization and users satisfaction. In this section, we review some recent research works related to cloud bursting. We classify the related work in three categories: works based on user constraints, works based on economic criteria, and works targeting specific application types. We describe the objective of each work, its context as well as the used method for achieving its objective. Finally, we summarize the reviewed work in Table 3.3.3.

3.3.1 Based on User Constraints

We identified a significant number of interesting cloud-bursting research works focusing on covering resources overload or satisfying user preferences and constraints in terms of applications deadline and performance as well as VM placement [84][105][115][103].

[84] proposes a multi-cloud environment for processing user demands where the users specify the percentage of resources to use from each cloud environment. If user preferences may not be satisfied initially because of unavailability of resources, the authors propose using already deployed instances in less desirable clouds and then rebalancing instances progressively instead of pending user demands until the environment matches the specified cloud preferences. The rebalancing policies determine whether or not an excess instance (an instance in a particular cloud that exceed the user desired amount) should be terminated to create a new one in a higher-preferred cloud. The authors proposed three rebalancing policies: *Opportunistic-Idle*, *Force-Offline* and *Aggressive Policy*. The Opportunistic-Idle policy terminates only idle excess instances. The Force-Offline policy marks excess instances offline to prevent them from accepting new jobs while allowing them to finish running jobs, then terminates the instances when they become idle. The Aggressive Policy aggressively terminates excessive instances even if they are running user jobs while giving the possibility to specify a job progress threshold after which the instance may no longer be terminated.

[105] considers optimizing the completion time of data-intensive jobs with cloud bursting while preserving the same order of their submission. They propose three scheduling heuristics that adapt to changing workload and available resources to optimize ordered throughput. The heuristics determine the number of resources to provision on the external cloud and decide which jobs to burst. The first heuristic parses the job queue in the order of job arrival and schedules each job in the internal or external cloud according to where it is expected to complete earliest. The second heuristic calculates the time of the entire job execution in the external cloud (uploading, processing, downloading) before the jobs preceding it complete in the private cloud. Based on this calculation the heuristic determines if there are enough workload queued in the internal cloud and decides whether or not bursting the current job. The third heuristic calculates a suffrage value as the difference between the best and the second best completion time of each job. Then schedules the job with the maximum suffrage on the node where it could complete earliest.

[115] presents a cloud bursting model for enabling elastic applications running across different cloud infrastructures. The authors propose a framework for building cloud-bursting applications that monitors their performance and decides when to burst them into the public cloud and when to consolidate them again accordingly, where they link applications' performance to the private resources load. When an application finishes, the mapping of the other applications to resources is replanning.

[103] aims at optimizing the performance of big-data analytics in loosely-coupled distributed computing environments such as federated clouds. The data is divided into chunks to be analyzed by an algorithm that determines the nodes to use for each data chunk. Initially, the algorithm sorts the set of data chunks according to their size and the nodes from each cloud according to their data transfer delay plus their computation time for a fixed size of data. The algorithm assigns data chunks to nodes such that large data chunks are handled by nodes with higher data transfer delay and computation time. The algorithm organizes the sequence of chunks for each node in order to maximize the overlap between the data transfer and computation.

The main limitation of the aforementioned works consists in under-estimating the additional cost incurred using public cloud resources.

3.3.2 Based on Economic Criteria

We identified other works that base their cloud bursting decisions according to economic criteria [97][141][90][142].

[97] proposes a system, called *Seagull*, for determining the most economic applications to migrate into the cloud when the enterprise local resources are overloaded. The authors assume that applications are composed of one or more virtual machines hosted in a private data center and each application may be scaled either horizontally or vertically. They also assume that the data center overload is predictable. Based on that, they generate a list of applications likely to become overloaded on which they periodically perform pre-copying by transferring an incremental snapshot of their virtual machine's disk-state to the cloud in order to reduce the cloud bursting latency once it occurs. When an application is overloaded, they decide whether moving it to the cloud or moving other applications and free-up resources to the overloaded application. Their decision is driven by the migration cost with the objective of optimizing the amount of local capacity freed per dollar spent running applications in the cloud. The underestimated cost in this work is the cost of saving VM disk images.

[141] considers cloud providers supporting two QoS levels and VM pricing models: on-demand and spot. The two models are inspired from the VM pricing models provided by Amazon EC2[3]. On-demand VMs are provisioned for the users and payed by its hour usage. If the provider doesn't possess enough resources the request is rejected. Spot VM requests include the number of VMs and a maximum price, called a *bid*, the user is willing to pay per VM/hour. The VMs are instantiated if the current price is below the bid price and run until either the user decides to terminate them or the price goes above the bid. When the provider receives an on-demand VMs request and resources are unavailable, the authors propose comparing the profit of outsourcing the request to other clouds member of a federation or terminating spot VMs. In this work, the cost of the lost computation due to terminating spot instances is underestimated.

[3]http://aws.amazon.com/ec2/purchasing-options/

[90] focuses on the parameters impacting the profitability of using a cloud federation for one cloud provider. A global scheduler is proposed to enable cloud providers deciding between selling their free resources to other providers, buying resources from other providers or shutting down unused nodes to save power. The scheduler tries to maximize the provider's profit while guaranteeing that the hosted services meet their performance goals. The main considered parameters for deciding where services will be executed are the provider's incoming workload, the cost of buying additional resources, the generated revenues from selling unused resources and the cost of maintaining the provider's resources active.

[142] proposes an architecture for cloud brokering and multi-cloud VM placement. The objective is to optimize the placement of VMs across multiple cloud providers according to user-specified criteria, such as VM hardware configuration, the number of VMs of each instance type, the minimum and maximum percent of all VMs to be located in each cloud as well as the minimum performance and the maximum price. Based on these criteria and the available cloud providers, the cloud broker should generate an optimal deployment plan. The authors provide an integer programming model formulation to enable an optimized VM placement based on price-performance tradeoffs and propose using modeling languages and solvers, such as AMPL, to solve this optimization problem.

Some of these works present a limitation that consists in under-estimating the cost of their used mechanisms [97][141] and other works do not consider how to concretely apply their approaches in a real cloud environment [90][142].

3.3.3 Targeting Specific Application Types

There are other relevant cloud-bursting research works aware of the public cloud resources cost but they target specific application types [99][79][121].

[99] presents a framework, called *PANDA*, for scheduling BoT (Bag of Task) applications across resources in both private and public clouds. The objective of this work is to optimize the performance to cost ratio and enabling users choosing the best tradeoff between application completion time and cost. The authors assume having a generator that produces a set of feasible Pareto-optimal points proposed to the user for selecting the point with the maximum utility. Based on the selected Pareto-optimal point PANDA finds a near optimal schedule by defining an optimal workload for each resource and rearranging tasks in the way cost and/or makespan is reduced. The authors use a model where each of the private and public resources has a specific speed and cost and each task of an application has a specific and known running time.

[79] proposes a set of algorithms to cost-efficiently schedule batch workloads under deadline constraints on either a private infrastructure or on public clouds. In this work, the authors assume that private resources are costless and they attempt to maximize the local infrastructure utilization. They propose a scheduler to decide whether an incoming application can be scheduled on the organization's private infrastructure or on the public cloud depending on the application's deadline and potential cost. When a new application is submitted for execution, it is inserted in the singe system queue according a queue sort policy that follows FCFS (First-Come First-Served) and EDF (Earliest Deadline First) policies. The scheduler executes the application at the front of the queue on the private cloud as soon as sufficient resources become available and

constantly scans the queue in order to detect unfeasible applications (applications that will not be able to finish before their deadlines). If an unfeasible application is detected, the authors propose two options, either moving the unfeasible application to a public cloud or moving application(s) which precede(s) the unfeasible application and incurs a lowest cost on the public cloud.

[121] proposes a model, called *Elastic Site*, to extend a private cluster using public cloud resources in order to respond to changing demand. The model is integrated with a batch scheduler where the resources are provisioned based on changes in the cluster job queue. The authors propose three provisioning policies: *On-demand*, *Steady stream* and *Bursts*. The on-demand policy boots a new cloud VM each time a new job is queued and terminates all idle VMs when the queue is empty. For the second and third policies the authors estimate the *waste time* as the sum of average startup times and shutdown times for a particular VM image on the cloud. In the steady-stream policy a new VM is booted when the total queued wall time becomes greater than five times the estimated waste time of a particular cloud and terminates when the total queued wall time drops below three times the estimated waste time of the cloud. In the bursts policy the number of VMs to launch is calculated by dividing the total wall time of all queued jobs by two times the estimated waste time.

3.4 Economic Optimizations

Economic optimizations have been a focus of multiple research work in cloud computing due to its economic importance. In this section, we survey and classify some of the recently proposed related work in three categories: works targeting specific application types, works targeting specific infrastructures, and works focusing on the optimization of energy costs. Our survey focuses on presenting the objective of each work, its considered constraints, its targeted type of applications and/or infrastructures, and the used methods for achieving its objectives. Finally, we summarize the reviewed works in Table 3.4.3.

3.4.1 Targeting Specific Application Types

We identified many economic optimization works in the literature that focus on specific application types [93][114][58][64][120]. In the following we describe some relevant ones.

[93] considers the SLA-based resource allocation problem for multi-tier applications in cloud computing. The objective is to optimize the total profit from the SLA contracts that may be reduced by operational costs. The authors assume that servers are characterized by a maximum capacity and the SLA of each client request has a price based on the response time. The proposed solution first provides an upper bound on the total profit then tries to reach it by applying a simultaneous server consolidation and resources assignment based on force-directed search method.

[114] proposes an SLA-aware scheduling mechanism for running scientific applications on a public IaaS cloud. The authors assume that each application has an approximated execution time and once it starts running it may not be interrupted or moved to a different resource. They take into account the required time to provision and schedule an application and consider the payment of penalties if an application's execution is delayed. To minimize the cost of running the applications, each time an application

Table 3.4: Summary of cloud bursting related work

	Objective	Context	Method
[84]	Satisfying user preferences	Multi-cloud	Terminating excess VMs: (1) if idle, (2) after running jobs complete, (3) aggressively.
[105]	Fast and ordered jobs' completion	Scheduling data-intensive apps	Bursting a job if (1) it completes earlier externally. (2) many jobs are queued internally. (3) it saves max time when running externally.
[115]	Covering load spikes	Elastic apps	Bursting and consolidating the applications based on the load increase and decrease.
[103]	Optimizing performance	big-data analytics	Assigning data to nodes based on chunks size and nodes transfer and computation time.
[97]	Managing local resources overload	Enterprise	Determining the least costly applications to migrate into the cloud.
[141]	Maximizing provider's profit	Cloud federation and using spot instances	Deciding between outsourcing resources or terminating spot instances to serve new requests.
[90]	Maximizing cloud provider revenues	Cloud federation	Deciding between resources outsourcing, insourcing or nodes shut downing.
[142]	Optimizing VM placement	Multi-cloud	Formulating an integer programming model to be solved with a modeling solver (AMPL).
[99]	Optimizing ratio performance/cost	Scheduling BoT apps	Preprocessing workload and assigning tasks according to a Pareto-optimal point.
[79]	Cost-efficiently meeting deadlines	Scheduling batch workloads	Moving to cloud either the application that cannot meet its deadline locally or the cheapest application preceding it in the queue.
[121]	Responding to changing demand	Cluster: batch scheduler	Adding one VM (1) when a new job is queued. (2) when queued wall time is 5x greater than waste time. Or (3) calculating VMs to add by dividing queued wall time by 2x waste time.

request arrives they decide between instantiating new resources for running it or using existing ones.

[58] provides a cost-efficient cloud resource allocation approach for web applications. The method proposed for reducing the applications hosting costs consists in reducing the number of required VMs by sharing them between applications. The authors define applications' QoS in terms of server resource utilization metrics and provide a proactive scaling algorithm accordingly. Thus, they add and remove fractions of VMs for the applications and maintain resource utilization below a specific upper limit and a suitable number of VMs to host the applications.

[64] focuses on time and cost sensitive execution for data-intensive applications executed in a hybrid cloud. The authors consider two different modes of execution: (1) Cost constraint-driven execution, where they minimize the execution time while staying below a user-specified cost constraint, and (2) Time constraint-driven execution, where they minimize the cost while completing the execution within a user-specified deadline. They monitor data processing and transfer times to predict the expected time and cost for finishing the execution. The allocation of resources may change, when needed, to meet the specified time and cost constraint. They propose a model based on a feedback mechanism in which compute nodes regularly report their performance to a centralized resource allocation subsystem, and the resources are dynamically provisioned according to the user constraints

[120] formulates the problem of minimizing the cost of a running computational application on a hybrid cloud infrastructure as a mixed integer nonlinear programming problem and specifies it using a modeling language. The authors assume heterogeneous compute and storage resources from multiple cloud providers and a private cloud. Each cloud has a maximum number of resources and the resources are parameterized by costs and performance. To calculate the cost of a single task, they include the cost of the VM instance, the time required for transferring data and the time for computing the task. For each task, they compute the relation between deadline and cost. The defined objective function consists in getting the minimum total cost for running the application tasks under a deadline constraint.

3.4.2 Targeting Specific Infrastructures

We identified other relevant economic research works which are applicable on specific computing infrastructures different from the infrastructure considered in this thesis, namely a cloud-bursting PaaS [158][157][153][68][139][87]. In the following we present some of these works.

[158] formalizes the global cost optimization problem among multiple IaaS clouds. The authors assume a cooperative scenario where each IaaS cloud exposes it's workload and cost information. They propose an inter-cloud new job scheduling and leftover job migration. Their method is based on a double auction mechanism where each cloud provider gives buy-bids and sell-bids for the required VM instances for the jobs.

[157] studies the profit maximization in a cloud environment with geo-distributed data centers, where the data centers receive VM requests from customers in the form of jobs. The authors assume having a complete information of all job arrivals to guarantee a bounded maximum job scheduling latency. They propose a dynamic pricing of VM resources related to customers' maximum value for a job at a data center and choose the

minimum number of servers required to meet the VM demands.

[153] proposes a resource allocation algorithm for SaaS providers to minimize infrastructure cost and SLA violations. To achieve this goal, the authors propose mapping and scheduling mechanisms to deal with the customer-side dynamic demands and resource-level heterogeneity. The mechanisms translate client requests into VM capacity and try to minimize their cost by using the available space within existing VMs which allows a cost effective usage of resources.

[68] studies the optimal multi-server configuration for maximizing the profit in a cloud environment. The multi-server system is treated as an M/M/m queuing model to formulate and solve the profit optimization problem. The servers are configured based on the speed and power consumption models of a service request. Then, a probability density function (pdf) of the waiting time of the newly arrived service request is derived. And based on that the expected service charge is calculated. If the service quality satisfies the customer expectation then the service charge is linearly proportional to the task execution time. Otherwise, the service charge decreases linearly as the service response time increases. If the request waits too long then no service charge is applied and the service is free. Based on that, the expected business gain is obtained in unit of time and the optimal server's size and speed are numerically deduced.

[139] proposes a cost-efficient task scheduling in a cloud environment. The problem is presented as a directed acyclic graph (DAG), where nodes are tasks and edges are precedence constraints between tasks. Weights are assigned to nodes and edges based on respectively the predicted tasks execution time and the predicted time to transfer data between VMs. Finally, the scheduling plans for mapping tasks are calculated according to the most cost-efficient VMs

[87] provides an SLA-driven management solution for maximizing the profit of PaaS providers that host their services on IaaS clouds. To achieve the providers' profit maximization, the authors propose reducing resource costs by sharing them among different contracts while ensuring agreed QoS in the SLA. They propose a mechanism with multiple control loops for adjusting services configurations and resource usage in order to maintain SLAs in the most cost-effective way. In peak periods, their mechanism chooses the most suitable request to be aborted to handle the lack of resources.

3.4.3 Focusing on the Optimization of Energy Costs

There are other research works that focus on the optimization of the electricity costs for optimizing cloud data centers costs. For instance, [148] studies the power consumption of data centers for cost-effective operation taking into account multiple electricity pricing schemes. [117] considers geographically distributed data centers in a multi-electricity market environment and proposes an energy-efficient, profit and cost aware request dispatching and resource allocation algorithm to maximize a service provider's net profit. [92] considers resource allocation problem aiming to minimize the total energy cost of cloud computing system while meeting specific client-level SLAs. [122] maximizes providers net revenue by dynamically powering servers on and off in order to minimize the amount of consumed energy.

As it can be observed, the majority of the reviewed economic optimization works do not handle peak periods. They consider either using a public IaaS cloud and having access to an unlimited number of resources or rejecting requests that may not be served

if no resources are available.

3.5 Gaps

Table 3.5 compares the goal of this thesis (the last row) with the most relevant related systems previously analyzed. The comparison criteria highlight the required features for meeting the objectives of this thesis; namely, optimizing the provider profit in a cloud-bursting PaaS environment under SLA constraints. In the comparison table we use three symbols to express (1) if the criterion is met (✓), (2) if the satisfaction of the criterion is unknown, i.e. there is no publicly available information (-), and (3) if the criterion is not met (✗). In the following we explain the meaning of each criterion.

Hosting Environment. The hosting environment describes the computing environment considered in each work and its capabilities. We focus on three capabilities.

1. *Open.* A hosting environment is considered as open if it is designed to be extensible and able to support a large variety of application types, frameworks and program languages.

2. *Multi-Cloud Applications Deployment.* This capability is valid if the hosting environment enables the deployment of applications on simultaneously several IaaS cloud systems.

Business Model. The business model considers the economic policy provided in each work and its corresponding features that are part of the business model. We focus on the three following features.

1. *Optimization of Provider Profit.* This feature is valid if the marketing solution provides a provider profit optimization policy.

2. *Support for SLA Constraints.* This feature is valid if the marketing solution provides SLA/QoS support, aims to satisfy them and incurs penalties if they are not satisfied.

3. *Support for PaaS.* This feature is valid if the marketing solution is applicable on a PaaS environment.

The general observation is that no related work targets the objectives of this thesis nor covers the required features to achieve them. The works focusing on optimizing the provider profit are generally specific to application models and may not be applied to an open hosting environment. Further, many works providing an open hosting environment provide no economic oriented features. Moreover, few works provide or consider the possibility of deploying applications on a multi-cloud environment.

3.6 Summary

In this chapter we presented the main required features to achieve the objective of this thesis, which consist in a cloud-bursting PaaS hosting environment and a profit-efficient business model. We reviewed the current state of the art of cloud bursting environments and economic optimization policies. Moreover, we analyzed some current commercial,

Table 3.5: Summary of economic optimization policies

	Objective	Constraints	Target applications and infrastructure types	Method
[93]	Optimizing profit	Response time	Multi-tier applications in IaaS cloud	Force-directed resources assignment
[114]	Minimizing cost	Execution time	Scientific applications on IaaS cloud	Deciding between reusing or renting new VMs
[58]	Minimizing cost	Bounded server load	Web applications hosting (PaaS)	Dynamically allocating VM's fractions to applications
[64]	(1) Minimizing execution time (2) Minimizing cost	(1) Cost (2) Deadline	Data-intensive applications in hybrid cloud	Dynamic resource allocation based on feedback
[120]	Minimizing cost	Deadline	Batch applications in hybrid cloud	Mixed integer nonlinear formalization
[158]	Minimizing cost	Response time	Multiple IaaS clouds	Inter-cloud job scheduling based on sell/buy bids
[157]	Maximizing profit	Bounded latency	Geo-distributed data centers	Dynamic VM pricing and server provisioning
[153]	Minimizing cost and SLA violations	-	SaaS	Optimizing the use of VMs available resources
[68]	Maximizing profit	Response time	Multi-server system	Defining server configurations
[139]	Minimizing cost and makespan	-	IaaS cloud	Scheduling tasks on VMs with low monetary cost
[87]	Maximizing profit	Throughput	Services hosting (PaaS)	Dynamic resource sharing between services
[117]	Maximizing profit	-	Geo-distributed data centers	Dynamic request dispaching on resources
[92]	Minimizing energy costs	Service time	IaaS Cloud	Dynamic resource allocation
[122]	Maximizing profit	-	IaaS Cloud	Dynamically powering servers on/off to minimize consumed energy

Table 3.6: Positioning this thesis with the main related works.

Approach	Hosting Environment		Business Model		
	Open	Multi-Cloud Applications Deployment	Optimization of Provider Profit	Support for SLA Constraints	Support for PaaS
[93]	✗	✗	✓	✓	✓
[157]	✗	✗	✓	✓	✗
[58]	✗	✗	✓	✓	✓
[87]	✗	✗	✓	✓	✓
[114]	✗	✗	✓	✓	✓
[120]	✗	✓	✓	✓	✓
[68]	✗	✗	✓	✓	✓
[99]	✗	✓	✗	✓	✓
[105]	✗	✓	✗	✓	✓
[90]	✗	-	✓	✓	✓
[141]	✗	-	✓	✗	✗
[79]	✗	✗	✓	✓	✓
GAE [14]	✗	✗	-	✓	✓
Azure [1]	✗	✗	-	✓	✓
RightScale [21]	✓	✓	✗	✗	✓
Cloud Foudry [19]	✓	✗	✗	✗	✓
ConPaaS [130]	✓	✓	✗	✗	✓
OpenShift [25]	✓	✗	✗	✗	✓
Cloudify [18]	✓	✗	✗	✓	✓
[66]	✗	✗	✗	✓	✓
Resilin [101]	✗	✓	✗	✗	✓
Merkat [76]	✓	✗	✗	✓	✓
Aneka [146]	✓	✗	✗	✗	✓
Thesis goal	✓	✓	✓	✓	✓

open source and research PaaS systems regarding a number of characteristics; particularly, regarding their underlying infrastructure, provided services, supported application types and program languages, architecture, and billing and SLA models . We also stressed the limitations of the related work that prevent achieving the objective of this thesis. In the next chapters we introduce the contributions of this thesis.

Chapter 4

Profit Optimization Model

Contents

In the previous chapter we presented and analyzed a number of recent works treating aspects related to the objective of this PhD thesis, which consists in optimizing the provider profit in a cloud-bursting PaaS environment under SLA constraints. Our analysis has shown that existing works are applicable either to specific application types or on specific hosting environments different from a cloud-bursting PaaS. To address these limitations we propose in this chapter a generic profit optimization model, independent from specific application types and considering the payment of penalties if the SLAs of clients applications are not satisfied. First, we present a mathematical model of a cloud-bursting PaaS environment as well as the objective and the constraints of the PaaS provider. Then, we propose generic policies for optimizing the PaaS provider profit.

This chapter is organized as follows. Section 4.1 provides definitions and assumptions about the considered PaaS hosting environment. Section 4.2 describes the considered system using a mathematical model. Section 4.3 presents a number of policies used to optimize the PaaS provider profit. Finally, Section 4.4 summarizes the chapter.

4.1 Definitions and Assumptions

In this section we define the characteristics of the considered PaaS hosting environment as well as the background assumptions. First of all, we consider an open PaaS system able to host a wide variety of application types and that relies on private resources owned by the provider and is able to burst into public IaaS cloud resources, as seen in Figure 4.1. The main function of the PaaS is to assign resources to applications according to specific policies that depend on the provider objectives. Furthermore, we consider that the PaaS system uses a different resource configuration for each application type because each type has specific dependencies and requirements in terms of software stack and management tools. Thus, the private resources are partitioned into several groups, where each group is dedicated to a specific application type. When necessary, each group may burst into public cloud resources independently from the other groups. Thereby, each group is composed of a set of private resources and possibly some public resources.

The PaaS providers objective considered here is the optimization of their profit while taking into account two parameters. The first parameter consists in providing SLAs to the hosted applications based on QoS properties, such as response time and throughput. The second parameter consists in the payment of penalties if the level of QoS promised in the SLA is not met. To achieve this objective, our approach consists mainly in selecting the cheapest resources for the provider able to host and satisfy the SLA of each application. Indeed, we consider that both of private and public resources have a known cost for the PaaS provider.

Our approach is generic and independent from the types of supported applications but it relies on one background assumption. Namely, the considered assumption is that for each supported application type there is a corresponding manager capable to translate the applications' required QoS levels into resource requirements, to estimate their possible penalties due to QoS impacts and to predict their resource usage scheme. Such capabilities are based on application type-specific knowledge and performance model.

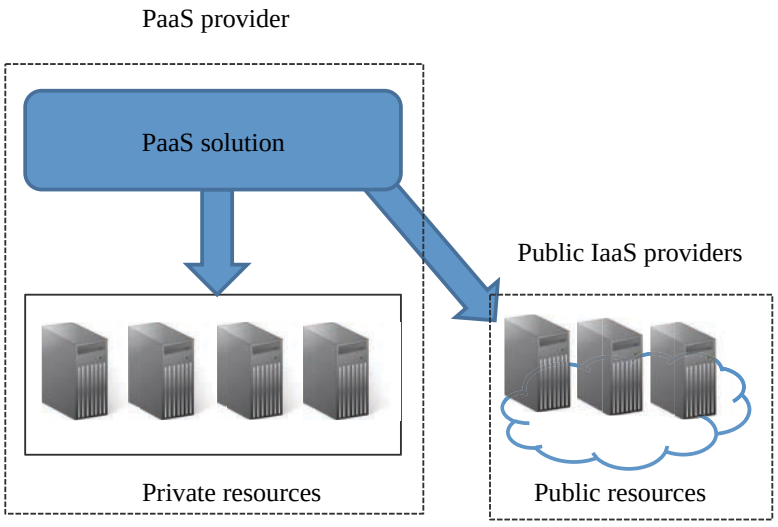


Figure 4.1: Overview of the considered hosting environment

4.2 PaaS System Model

In this section, we formalize the considered PaaS system as well as the provider objective and constraints using a mathematical model.

4.2.1 Notations

Basically, the considered PaaS system relies on a constant number K of private resources and on a variable number Z of public cloud resources, $R = \{R^1, R^2, ..., R^K, R^{K+1}, ..., R^{K+Z}\}$. The private resources may be either available or used by the hosted applications and the public resources may only be used, otherwise they are released to avoid the provider paying additional charges. Note that in this model we consider that the PaaS provider pays the public resources exactly according to their usage duration. For simplicity reasons, we consider that each of the private and public resources are built in a way to be homogeneous. Thus, if an application runs on a resource R^i from the private or public cloud it achieves the same QoS level. Each resource has a cost and a price. The resource price R_{price} is set by the PaaS provider and used for billing the provided services. The resource cost may be either private $R_{private}$ or public R_{public} depending on the location of the resource. The private cost is estimated by the PaaS provider according to the incurred operational costs and the public cost is dynamically obtained from public IaaS cloud providers. Obviously, we assume that the resource price is set to be higher than the private and public costs, in order to enable the provider to make profit.

The resources are partitioned into M groups, $G = \{G^1, G^2, ..., G^M\}$, where M is the number of the supported types of applications. Each group G^j holds a subset of resources G^j_R, where (1) the sum of the number of resources assigned to each one of the groups in the system is equal to the total number of resources in the system (see Equation 4.1a) and (2) each resource R^i in the system belongs to a unique group G^j (see Equation 4.1b). The PaaS system hosts N applications, $A = \{A^1, A^2, ..., A^N\}$, of all the supported types. Each group G^j hosts a subset of applications G^j_A of the same type where the sum of the number of applications belonging to each one of the groups in the system is equal to the total number of applications hosted in the system (see Equation 4.1c). Further, the resources are considered as computing units that may not be shared between multiple applications. Thus, each resource is held by at most one application and each application holds at least one resource.

More generally, each application A^x holds a number of resources A^x_R that may be either private $A^x_{private}$ or public A^x_{public} and requires a specific QoS level A^x_{QoS}. The cost of hosting an application for the provider A^x_{cost} is calculated as a function of the required QoS level, the number of private and public resources used by the application, and the cost of the private and public resources for the provider. The price paid by the user for hosting an application A^x_{price} is calculated as a function of the required QoS level, the number of resources used by the application, and the resources price set by the provider. If the QoS level required by an application is impacted according to an impact level A^x_{impact} (e.g., the impact level could be the extent that the application runtime exceeds its deadline), an application penalty $A^x_{penalty}$ is calculated as a function of this impact and the initial application revenue paid by the user. Otherwise, the application penalty is zero. The revenue generated by the provider from hosting an application $A^x_{revenue}$ is calculated as the difference between the application price and penalty. Finally, the

profit obtained by the provider from hosting an application A^x_{profit} is calculated as the difference between the application revenue and the application cost.

Table 4.1 summarizes all the notations used in this chapter, the aforementioned ones as well as the ones that will be used in the next sections.

PaaS system properties:

$$\sum_{j=1}^{M} \left| G^j_R \right| = K + Z \tag{4.1a}$$

$$\forall R^i \in R,\ \exists! G^j \in G,\ R^i \in G^j_R \tag{4.1b}$$

$$\sum_{j=1}^{M} \left| G^j_A \right| = N \tag{4.1c}$$

4.2.2 Objective

As already mentioned, our objective is to maximize the overall PaaS provider profit while providing specific QoS levels to applications or paying penalties otherwise. We consider that the QoS level required by an application is translated into resources requirements by a manager specific to the corresponding application types. If this QoS is impacted, an application penalty that affects the application profit is computed accordingly. Thus, the objective is formulated here in the maximization of the sum of all the hosted applications profits (see equation. 4.2).

Objective:

$$\max(\sum_{x=1}^{N} A^x_{profit}) \tag{4.2}$$

4.2.3 Constraints

The payment of penalties when the required QoS level by an application is not met is an attractive feature for the users. However, if the provider does not fulfill its commitments regarding the initially promised QoS level to applications, the customers will be disappointed and may migrate to an other competitor. Consequently, in addition to the penalties, the provider should also care about its reputation. Specifically, the provider has to limit the number of impacted applications as well as the level of impact for each impacted application, even if this may decrease its profit. Indeed, if a QoS impact exceeding a predefined limit is detected during the execution of an application the provider should fix it by adding resources for example, even if the additional resources will not be accounted in the price paid by the user. This may reduce the provider profit in the short term but it is advantageous for the longer term.

Namely, we consider two constraints. First, the impact level of impacted applications should not exceed a threshold $A_{threshold}$ predefined by the provider (see Equation 4.3a), where the threshold $A_{threshold}$ is the highest allowed level of QoS impact of each application in the PaaS system. Second, the ratio of the impacted applications in the PaaS

Table 4.1: Notations

Notation	Definition	Value
R_{price}	Resource price	Defined by the PaaS provider
$R_{private}$	Private resource cost	Estimated by the PaaS provider
R_{public}	Public resource cost	Obtained from public providers
G_R	Set of resources in the group	
G_A	Set of applications in the group	
G_{impact}	Percentage of impacted apps in G	
G_{wait_bid}	Group's waiting bid	See section 4.3.3.2
$G_{donating\_bid}$	Group's donating bid	See section 4.3.3.1
$G_{donating\_impact}$	Percentage of impacted apps if $G_{donating\_bid}$ is selected	See section 4.3.3.1
$A_{private}$	Application's private resources	
A_{public}	Application's public resources	
A_R	Application's used resources	$A_{private} + A_{public}$
A_{QoS}	Application's required QoS	Negotiated with SLA
A_{cost}	Application's cost	$cost(A_{QoS}, (A_{private}, R_{private}), (A_{public}, R_{public}))$
A_{price}	Application's price	$price(A_{QoS}, A_R, R_{price})$
$A_{revenue}$	Application's revenue	$A_{price} - A_{penalty}$
A_{impact}	Application's impact	
$A_{penalty}$	Application's penalty	$penalty(A_{impact}, A_{price})$
A_{profit}	Application's profit	$A_{revenue} - A_{cost}$
$A_{threshold}$	Application's impact threshold	Defined by the PaaS provider
$req_{private\_cost}$	Request's private cost	See section 4.3
req_{public_cost}	Request's public cost	See section 4.3
$PaaS_{threshold}$	Ratio of impacted applications threshold	Defined by the PaaS provider

system should be limited to a threshold $PaaS_{threshold}$ predefined by the provider (see Equation 4.3b), where the threshold $PaaS_{threshold}$ is the maximum authorized ratio of applications that may be impacted in the PaaS system.

Constraints:

$$\forall A^x \in A, A^x_{impact} \leqslant A_{threshold} \tag{4.3a}$$

$$\frac{\sum_{j=1}^{M} \left|G^j_A\right| \times G^j_{impact}}{N} \leqslant PaaS_{threshold} \tag{4.3b}$$

4.3 Policies

In this section we define three policies for sharing the private resources between the different application type-specific groups and hosting each newly arriving application. In the three policies we consider that initially the private resources are shared between

the groups either equally or according to application type-specific workload predictions. Then, each time a request *req* for hosting a new application arrives the policies try to find the suitable resources to host it. We consider that the user of the new application has already negotiated an SLA and QoS properties with the corresponding application type-specific manager which translates the application QoS level requirement into resources requirement. The request has the same attributes as the other applications hosted in the PaaS system, presented in Section 4.2, plus two additional ones: a private cost $req_{private\_cost}$ and a public cost req_{public_cost}. The private cost is calculated as if the new application will use private resources only and the public cost is calculated as if the new application will use totally or partially public resources. The two costs are compared and the cheapest one determines the resources to use for hosting the application.

4.3.1 Basic Policy

The basic policy is naive. It statically partitions the private resources between the groups. Thus, a constant number of private resources is assigned to each group. If a group has no more available private resources and a new application arrives, the new application will be hosted on public resources regardless of the possible available private resources in the other groups.

Specifically, when a new application request arrives its corresponding manager behaves as shown in Algorithm 1. First, the manager gets the current resources prices from a set of public IaaS cloud providers on which the PaaS may burst. The cheapest public cloud price is selected as a resource's public cost. Then, the manager compares this public cost with the private cost predefined by the PaaS provider. The cheapest cost determines whether the new application will be hosted on the private or public resources. Nevertheless, if there are not enough private resources available in the corresponding group we rent public resources.

Algorithm 1: Basic Policy

Data: *req*
Result: Find resources for *req*

Get current resources' prices from a set of pubic cloud providers ;
$R_{public} = min(public\ cloud\ prices)$;
if $(R_{private} \leqslant R_{public}) \wedge (\exists r \subseteq G_R^{current}, (\forall R^i \in r, R^i\ is\ available) \wedge (|r| \geqslant req_R))$ **then**
    Use private resources ;
else
    Rent public resources from the provider with the cheapest price;
end

4.3.2 Advanced Policy

The advanced policy offers more flexibility than the basic one. It provides mainly two more features: *exchange* and *hybrid*. The exchange feature consists in enabling the application-type specific groups to exchange private resources between each other. Specifically, when the private resource cost is cheaper than the public resource cost and the corresponding group does not have enough private resources for hosting a new

application, we check if the other groups have available private resources before renting public cloud resources. The hybrid feature consists in enabling applications to run simultaneously on both private and public resources. Specifically, when the private resources cost is cheaper than the public one and there are not enough available private resources for hosting a new application, rather than renting all the required resources from a public cloud we rent only the number of missing resources.

In Algorithm 2 we show the pseudocode of the advanced policy run by an application type-specific manager when its corresponding group receives a request for hosting a new application. First, the manager gets the cheapest public cloud resource cost and compares it with the private resource cost in the same way as in the basic policy. If the public cost is cheaper the manager rents the resources from the corresponding public cloud provider. Otherwise, the manager checks the number of private resources available in the current application type-specific group. If the available private resources are sufficient the manager uses them to host the new application. Otherwise, the manager calculates the number of missing resources $R_{missing}$ and checks if the other groups have at least as much available private resources. If so, the required resources are transferred from their initial groups to the requesting group to be used for hosting the new application. If not, the number of resources available on the other groups $R_{available}$ is calculated to deduce the number of missing sources. Then, the resources available on the other groups will be transferred to the requesting group, and the missing resources will be rent from the public cloud provider.

Algorithm 2: Advanced Policy

Data: *req*
Result: Find resources for *req*
Get current resources' prices from a set of public cloud providers ;
$R_{public} \leftarrow min(public\ cloud\ prices)$;
$R_{missing} \leftarrow req_R$;
$r \leftarrow$ set of available resources in $G_R^{current}$;
if $(R_{private} \leqslant R_{public})$ **then**
    if $(|r| < req_R)$ **then**
        $R_{missing} \leftarrow req_R - |r|$;
        if *(other groups have* $R_{missing}$*)* **then**
            Get $R_{missing}$ from corresponding groups ;
        else
            $R_{available} \leftarrow$ available resources on other groups ;
            $R_{missing} \leftarrow R_{missing} - R_{available}$;
            Get $R_{available}$ from corresponding groups ;
            Rent $R_{missing}$ from corresponding public cloud ;
        end
    else
        Use available resources in $G^{current}$;
    end
else
    Rent $R_{missing}$ from corresponding public cloud ;
end

4.3.3 Optimization Policy

The optimization policy goes one step further than the advanced policy in optimizing the use of private resources if their cost is cheaper than the public one. Specifically, when the available private resources in all the groups are not sufficient for hosting a new application, the policy compares the cost of three options for getting the missing resources: (1) renting them from a public cloud provider, (2) getting them from running applications, or (3) waiting for private resource to become available. The first option is the simplest one and is similar to the one used in the advanced policy. The second option, called *donating option*, may generate an impact on the promised QoS properties of the affected applications. The third option, called *waiting option*, may generate an impact on the promised QoS properties of the new application. In two last options, the payment of the incurred penalties should be considered. In the next subsections, we explain each of the donating and waiting options and provide the algorithm of the optimization policy.

4.3.3.1 Donating Option

The donating option consists in removing or borrowing a part of resources used by applications already running and giving them to the new application. The resources could be obtained either from one application or from a subset of applications belonging to a same group, where the number of resources to donate is less or equal to the total number of private resources in the group. The QoS promised in the SLAs of the affected applications may deteriorate and as a result the provider loses some revenue due to the payment of penalties. The sum of the possible penalties of the applications that may be affected in a group in order to donate the required resources is called a group *donating bid* and referenced as $G_{donating\_bid}$. Each application type-specific manager is responsible for determining which applications could be affected for calculating the amount of their possible penalties. This calculation is based on the corresponding application type-specific performance model. Thus, each group provides (1) a donating bid, (2) the average ratio of impacted applications in the group if the donating bid is taken, and (3) the average ratio of applications impacted in the past.

The idea is to select one donating bid from the proposed ones and use it to make the choice between using the private and public resources. As shown in Algorithm 3, to select a donating bid first we perform an ascendant sort on the groups based on their corresponding donating bids. Then, we calculate the average ratio of the impacted applications in the overall PaaS system as if the smallest donating bid was selected. If this average ratio is smaller or equal to the predefined PaaS threshold then we select the smallest donating bid. Otherwise, we perform the same steps with the next smallest donating bids until we find a donating bid that satisfies the impact threshold constraint or exhaust all groups. If no proposed donating bids enable the satisfaction of the threshold constraint, we set the donating bid value to infinity in order to avoid selecting the donating option.

4.3.3.2 Waiting Option

The waiting option consists in either maintaining the new application in a queue until all the required resources become available or starting the application with only the part

Algorithm 3: get_donating_bid() function

Data: $R_{missing}$
Result: Find G^j, $donating\_bid$

$\forall G^j \in G, get(G^j_{donating\_bid}, G^j_{donating\_impact}, G^j_{impact})$;
$sort(G^j, G^j_{donating\_bid})$;
for $j \leftarrow 1$ ***to*** M **do**
  $bid \leftarrow G^j_{donating\_bid}$;
  $donating\_bid_{impact} \leftarrow \frac{\left|G^j_A\right| \times G^j_{donating\_impact} + \sum_{y=1,y\neq j}^{M} \left|G^y_A\right| \times G^y_{impact}}{N+1}$;
  if $(donating\_bid_{impact} \leqslant PaaS_{threshold})$ **then**
    $break$;
  end
end
if $(donating\_bid_{impact} > PaaS_{threshold})$ **then**
  $donating\_bid \leftarrow \infty$;
end
return G^j, $donating\_bid$;

of resources already available and adding the missing resources once they become available. The resources become available following either the end of a temporal application or a decrease in the resource consumption of a permanent application. This waiting time may impact the QoS properties promised to the new application which leads to the payment of a penalty, *waiting bid*, which is assumed to be calculated by the corresponding application type-specific manager. The required time for the other applications to release a number resources is also assumed to be computed by their corresponding application type-specific managers, knowing their application performance model. We assume that each application-type specific group provides its required time to have a number of available private resources G_{wait} and its corresponding *waiting bid* G_{wait_bid} calculated as the penalty of new application caused by G_{wait}. As shown in Algorithm 4, once all the groups propose their waiting bid, we select the smallest one and calculate the average ratio of impacted application in the overall PaaS system taking into account the impact of the new application. If this average ratio is greater than the predefined threshold, we set the waiting bid to infinity to avoid selecting this option and impacting one more application.

4.3.3.3 Algorithm

Algorithm 5 shows the pseudocode of the optimization policy with both options donating and waiting bids. First, the manager receiving the request gets the resource costs from a set of public cloud providers, selects the cheapest one as the public resource cost and compares it with the private resource cost. If the public cost is cheaper the manager hosts the new application on resources from the corresponding public cloud provider. Otherwise, the manager checks the availability of private resources on its corresponding application type-specific group. If the resources are sufficient the manager uses them for hosting the new application. Otherwise, the manager computes the number of missing

Algorithm 4: get_waiting_bid() function

Data: $R_{missing}$, req
Result: Find G^j, $waiting\_bid$

$\forall G^j \in G, get(G^j_{wait}, G^j_{wait\_bid}, G^j_{impact})$;
$sort(G^j, G^j_{wait\_bid})$;
$waiting\_bid \leftarrow G^1_{wait\_bid}$;
$wait_{impact} \leftarrow \frac{1 + (\sum_{j=1}^{M} \left|G^j_A\right| \times G^j_{impact})}{N+1}$;
if $(wait_{impact} > PaaS_{threshold})$ **then**
  $waiting\_bid \leftarrow \infty$;
end
return G^1, G^1_{wait}, $waiting\_bid$;

resources and checks if the other groups have as much private resources available. If so, the available resources will be transferred from their corresponding groups to the requesting group and will be used for hosting the new application. If not, in addition to transferring the available resources from their corresponding groups, the manger updates the number of missing resources to get the donating and the waiting bids. The two bids are compared and the smallest one is used to calculate the cost of the request using private resources. The cost of the request using public resources to cover the missing resources is also calculated. Finally, the two costs are compared and the cheapest one determines if the missing resources will be rent from a public cloud provider or obtained through the waiting or the donating bid options.

4.4 Summary

In this chapter we introduced the considered PaaS hosting environment and formalized it using a mathematical model. We also presented a set of policies for optimizing the profit of the PaaS provider when hosting applications. The main considered hosting environment features are the support of cloud bursting and the capacity to host several application types. Our mathematical model formalizes such a system as well as the applications' characteristics and requirements. The model formalizes also the PaaS provider objective and constraints. Specifically, the provider objective considered here is to earn the maximum profit from hosting applications with the constraint of satisfying their required QoS levels or paying penalties. Another considered constraint consists in limiting the QoS impact level of applications and the number of impacted applications if an impact is required to achieve the provider's objective. We provided a set of policies to map arriving applications on resources in a profit-efficient way. The policies compare the cost of hosting applications with the private resources and public resources at different levels to select the cheapest one. The main assumption of the proposed model and policies is that the PaaS system is able to estimate the QoS level of an application its type-specific knowledge and performance model.

In the following chapter we investigate computational applications and propose SLA and pricing function accordingly, in order to provide an example of an application-type

Algorithm 5: Optimization Policy

Data: *req*
Result: Find resources for *req*

Get current resources' prices from a set of public cloud providers ;
$R_{public} = min(public\ cloud\ prices)$;
$R_{missing} \leftarrow req_R$;
$r \leftarrow$ set of available resources in $G_R^{current}$;
if $(R_{private} \leqslant R_{public})$ **then**
  if $(|r| < req_R)$ **then**
    $R_{missing} \leftarrow req_R - |r|$;
    if *(other groups have $R_{missing}$)* **then**
      Get $R_{missing}$ from corresponding groups ;
    else
      $R_{available} \leftarrow$ available resources on other groups ;
      $R_{missing} \leftarrow R_{missing} - R_{available}$;
      Get $R_{available}$ from corresponding groups ;
      $(G^j, donating\_bid) \leftarrow get\_donating\_bid(R_{missing})$;
      $(G^j, wait, waiting\_bid) \leftarrow get\_waiting\_bid(R_{missing}, req)$;
      if $(donating\_bid < waiting\_bid)$ **then**
        $req_{private\_cost} \leftarrow$
        $cost(req_{QoS}, (req_R, R_{private}), (0, R_{public})) + donating\_bid$;
        $private\_option \leftarrow$ *donating bid* ;
      else
        $req_{private\_cost} \leftarrow cost(req_{QoS}, (req_R, R_{private}), (0, R_{public})) + waiting\_bid$;
        $private\_option \leftarrow$ *waiting bid* ;
      end
      $req_{private} \leftarrow req_R - R_{missing}$;
      $req_{public\_cost} \leftarrow cost(req_{QoS}, (req_{private}, R_{private}), (R_{missing}, R_{public}))$;
      if $(req_{private} < req_{public})$ **then**
        if *(private_option = waiting bid)* **then**
          if *(req support running with partial resources)* **then**
            Start running the new application with available resources ;
          end
          Wait during *wait* time ;
        end
        Get $R_{missing}$ from G^j ;
      else
        Rent $R_{missing}$ from corresponding public cloud ;
      end
    end
  else
    Use available resources in $G^{current}$;
  end
else
  Rent $R_{missing}$ from corresponding public cloud ;
end

specific manager.

Chapter 5

Computational Applications

Contents

This chapter completes the formalism of the PaaS profit optimization model proposed in the previous chapter with a computational applications' group model. Specifically, it proposes an SLA model for computational applications and presents corresponding donating and waiting bids heuristics for providing private resources from running applications.

The chapter is organized as follows. Section 5.1 defines computational applications and presents the considered background assumption in this work. Section 5.2 presents the assumed computational application performance model. Section 5.3 presents the used service level agreement contract, classes, functions and lifecycle. Section 5.4 describes the used donating and waiting bids heuristics. Finally, Section 6.4 summarizes this chapter.

5.1 Definitions and Assumptions

We define a computational application as a temporal passive application running for a finite duration without any human intervention (see definitions in Section 2.2). The applications may use either a fixed or a variable set of resources. Accordingly, we differentiate between two types of applications: rigid and elastic. The rigid applications require a fixed amount of resources and the elastic applications can be adapted to use different amounts of resources.

In this chapter, we propose a computational application manager responsible for providing and managing the entire lifecycle of applications and their corresponding SLAs. The role of this manager in our profit optimization model is to provide the donating and waiting bids of its corresponding application group. Our manager relies on one background assumption that consists in being capable to integrate a QoS predictor plugin. Such a plugin estimates the quality level of an application based on its characteristics and used resources. In the context of computational applications, we consider that the predictor plugin provides the runtime of applications according to the performance model described in the next section.

5.2 Performance Model

In our current implementation we assume that the computational applications have a linear speedup performance model, where the application runtime is inversely proportional to the used number of resources. Moreover, if an elastic application uses less or more resources during its execution, its remaining runtime changes according to its new number of resources. We also assume that based on the application progress and the number of resources we can predict its remaining execution time.

In practice, this simple performance model can be replaced by more realistic performance models. Examples of such models provided in the literature are [133] for MapReduce applications and [73] for scientific applications.

5.3 Service Level Agreement (SLA)

In this section we define a service level agreement for computational applications. Basically, we make use of a QoS-based SLA metric, called *deadline*. The deadline, referenced as $A_{deadline}$, is a main QoS property of computational applications, which consists in the overall time for running an application and giving results to its corresponding user. Accordingly, the user pays an amount of money to the provider, called henceforth the application price, referenced as A_{price}. In the following, we define the SLA contract and classes, propose revenue functions, and finally present the SLA lifecycle management.

5.3.1 SLA Contract

We consider an SLA contract where the user provides the description of the application. The application description contains the application characteristics such as its resource consumption pattern, and requirements in terms of software and hardware dependencies. Based on this description the PaaS system predicts the application runtime and

calculates the corresponding deadline and price. The application's deadline is calculated as the sum of its predicted runtime, $A_{runtime}$, and the required time for processing the application submission, $A_{processing}$, as seen in Equation 5.1a. The processing time includes the time for getting and configuring resources in order to host the application. The application's price is calculated as the product of its runtime, used resources and the resources price set by the provider, as seen in Equation 5.1b. If an application uses different quantities of resources during its execution, the price function will be decomposed in multiple runtime periods where in each period the application uses the same amount of resources. For example, if an application uses A_{R1} resources during a period A_{p1} and A_{R2} resources during a period A_{p2}, its price would be calculated as seen in Equation 5.1c. If the agreed application's deadline is exceeded, a penalty is calculated according to a specific function and deduced from the initial price in order to provide the real provider revenue (see Equation 5.1d).

$$A_{deadline} = A_{runtime} + A_{processing} \tag{5.1a}$$

$$A_{price} = A_{runtime} \times A_R \times R_{price} \tag{5.1b}$$

$$A_{price} = (A_{p1} \times A_{R1} + A_{p2} \times A_{R2}) \times R_{price} \tag{5.1c}$$

$$A_{revenue} = A_{price} - A_{penalty} \tag{5.1d}$$

5.3.2 SLA Classes

In order to target a wide range of consumers, service providers may propose several levels of QoS and corresponding prices. We call each QoS level and its corresponding price an *SLA class*. In our scenario, an application with a high SLA class has a shorter deadline and a higher price compared to the deadline and the price of the same application with a lower SLA class, taking into account the minimal possible application's runtime. On one hand, the SLA classes give many choices to customers for hosting their applications according to their preferences, best price or best QoS level. On the other hand, the SLA classes enable the providers to leave a margin when hosting applications with low SLA classes in order to serve applications with high SLA classes in peak periods .

We set a number of SLA classes, from the highest to the lowest one. Specifically, we define a deadline margin and a price factor for each SLA class, used to be multiplied respectively by the initial deadline and price, calculated in Equations 5.1 according to the predicted application's runtime (see Equations 5.2). The deadline margin and the price factor are set to be inversely proportional.

$$A_{deadline\_class} = A_{deadline} \times margin\_class \tag{5.2a}$$

$$A_{price\_class} = A_{price} \times factor\_class \tag{5.2b}$$

5.3.3 Revenue Functions

Provider revenues depend on their capacity to provide the QoS promised in the SLAs of their hosted applications. In our scenario, if the provider fails to give applications' results before their deadlines, the corresponding applications' revenues decrease according specific revenue functions. We call the difference between the initially expected application's revenue and the real application's revenue a *penalty*, referenced as $A_{penalty}$.

In this section we define three revenue functions for calculating an application's penalty: *linear*, *bounded linear* and *step*. The three functions depend on the delay to deliver results to the user, where an application's delay is calculated as the difference between the real application's completion time and its agreed deadline and is referenced as A_{delay}. In the following we describe each revenue function.

5.3.3.1 Linear

The linear function consists in increasing the penalty linearly to the increase of the delay. Specifically, it calculates the penalty according to the delay and an α value determining how fast the penalty increases, as seen in Figure 5.1. Specifically, the penalty is calculated as a delay coefficient (delay divided by the deadline) multiplied by the application price divided by α (see Equation 5.3). A high α value is more advantageous for the provider while a low one is more advantageous for the user. An α value is defined by the provider for each SLA class; the higher the SLA class, the lower the alpha value.

$$A_{penalty} = \frac{A_{delay}}{A_{deadline}} \times \frac{A_{price}}{\alpha},\ \alpha > 0 \tag{5.3}$$

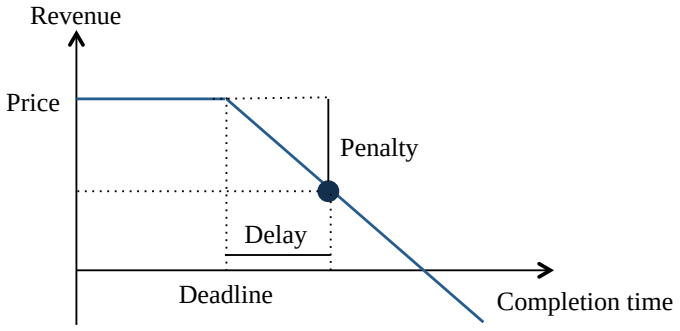


Figure 5.1: Linear revenue function

5.3.3.2 Bounded linear

The bounded linear function is quite similar to the linear function. However, the penalty is bounded to a maximum value defined by the provider in order to limit its loss of incomes (see Figure 5.2). Initially, the penalty is calculated exactly as in the linear function (Equation 5.3) then it is compared to a defined maximum penalty value. If it is greater, its value gets the maximum penalty value (see Equation 5.4). Otherwise, it keeps the same value.

$$\textbf{If } (A_{penalty} > penalty_{max}) \textbf{ then } A_{penalty} \leftarrow penalty_{max} \tag{5.4}$$

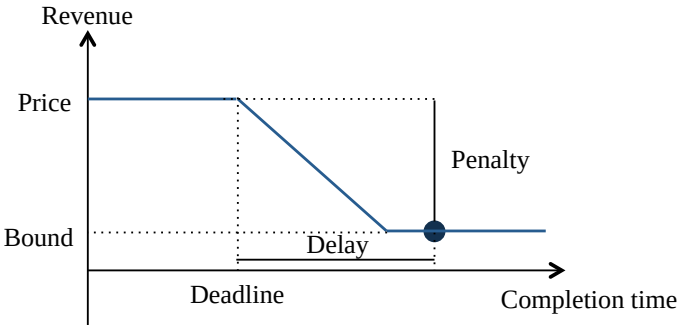


Figure 5.2: Bounded linear revenue function

5.3.3.3 Step

The step function consists in many constant values of revenues (see Figure 5.3). Each value corresponds to a completion time interval. Given that the penalty is the difference between the revenue and the initial price and the delay is the difference between the completion time and the deadline, the penalty gets a specific value for each delay interval (see Equation 5.5). A set of penalty values are defined by the provider for each SLA class; the higher the SLA class, the higher the penalty value.

$$A_{penalty} = \begin{cases} C_1 & \text{for } 0 \leqslant A_{delay} \leqslant x_1 \\ C_2 & \text{for } x_1 \leqslant A_{delay} \leqslant x_2 \\ \ldots & \\ C_n & \text{for } x_{n-1} \leqslant A_{delay} \end{cases} \tag{5.5}$$

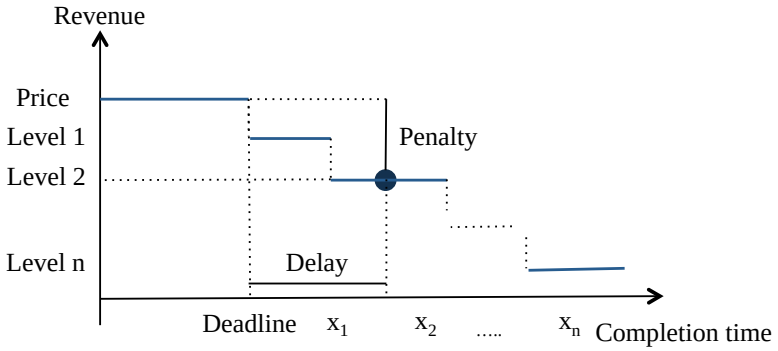


Figure 5.3: Constant revenue function

5.3.4 Lifecycle

As mentioned earlier, to enable the PaaS system to propose SLAs, first of all the user provides a description of the application to host. Based on this description and the used predictor plugin, the computational application manager provides the user with a set of pairs (deadline, price), each pair corresponding to an SLA class, and lets the user select one of them. If the user does not agree with any proposed pairs she may

propose one of the two SLA metrics. Then, whenever possible the provider proposes accordingly the second SLA metric for each SLA class if this is feasible. For instance, if the user has budget constrains, she may impose a price and the manager gives the corresponding deadline. Otherwise, if the user's application is urgent, she may impose a deadline and the manager gives the corresponding price. If the proposed SLA metric is not acceptable to the system, the user may propose another one and a new negotiation round is launched until the user and the provider agree. Based on the agreement, resources are provisioned and configured for hosting the application. At the end of the application execution, if the agreed deadline is exceeded, the penalty to be paid by the provider is calculated according to the delay and the used revenue function.

5.4 Bids Heuristics

As seen in the previous chapter, several options are possible when the PaaS system receives a request for hosting a new application, see Figure 5.4. The main options are the use of available private resources or public cloud resources. With the optimization policy there are two more options. Specifically, when no private resources are available and the cost of public resources is higher than the cost of private resources, each application type-specific group is asked to propose donating and waiting bids for providing a number of resources. The donating bid is calculated as the sum of the possible penalties of the applications that may be affected to donate the required resources. The waiting bid is calculated as the possible penalty of the new application due to waiting private resources to become available. In the following we present the heuristics for providing computational applications donating and waiting bids.

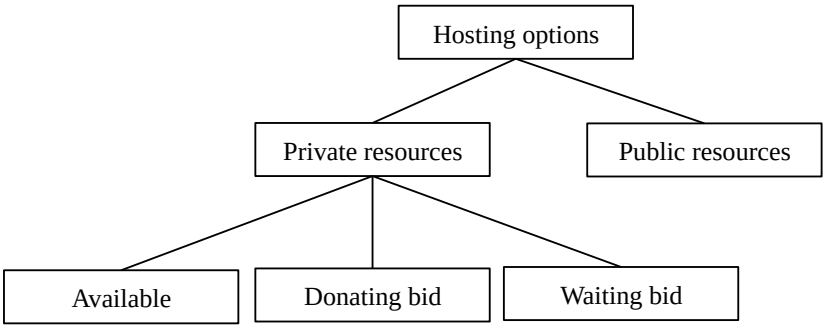


Figure 5.4: Hosting options

5.4.1 Waiting Bid

To propose a waiting bid, the computational applications' manager considers waiting private resources used by the running applications to become available. The required time for a running computational application to finish and release its resources is deduced from its predicted runtime and referenced henceforth as A_{wait}. Such a waiting time may impact the deadline of the new application with a specific delay that leads to a penalty, which is called henceforth application's waiting bid and referenced as A_{wait_bid}. The group's waiting bid is then calculated as the smallest possible application's waiting bid.

Specifically, we calculate the group's waiting bid as shown in Algorithm 6. First, the computational application manager performs an ascending sort of the applications belonging to the group according to their waiting bid. Afterwards, in a loop the manager selects the application with the smallest waiting bid and compares the number of its private resources to the number of missing resources. If it is greater or equal, the manager considers the application's waiting bid as the group waiting bid and exits the loop. Otherwise, the manager increments a variable R_{found} by the number of the application's private resources. Then, it selects the next application with the smallest waiting bid and performs the same check, and so on for the rest of applications. The manager exits the loop if (1) it finds an application holding enough resources, or (2) the number of found resources in the variable R_{found} becomes greater or equal to the missing resources. Thus, the group's waiting bid is respectively (1) the selected application's waiting bid, or (2) the waiting bid of the last application having incremented R_{found}.

Algorithm 6: Waiting bid of computational applications' group

Data: $R_{missing}$, req
Result: G_{wait}, G_{wait_bid}
$\forall A^i \in G_A$, calculate $(A^i_{wait}, A^i_{wait\_bid})$;
$sort(G_A, A^i_{wait\_bid})$;
// initializing variables ;
$G_{wait} \leftarrow \infty$; $G_{wait\_bid} \leftarrow \infty$;$R_{found} \leftarrow 0$; ;
for $(i \leftarrow 1$ ***to*** $|G_A|)$ **do**
    if $(A^i_{private} \geqslant R_{missing})$ **then**
        $G_{wait\_bid} \leftarrow A^i_{wait\_bid}$;
        $G_{wait} \leftarrow A^i_{wait}$;
        break ;
    else
        $R_{found} + \leftarrow A^i_{private}$;
        if $(R_{found} \geqslant R_{missing})$ **then**
            $G_{wait\_bid} \leftarrow A^i_{wait\_bid}$;
            $G_{wait} \leftarrow A^i_{wait}$;
            break ;
        end
    end
end
return G_{wait}, G_{wait_bid} ;

5.4.2 Donating Bid

To propose a donating bid, the computational applications' manager considers impacting its applications using one of three possible forms of impact. The first form, called *partial lending*, consists in lending a subset of application resources to the request, the application continues running with the remaining resources, and gets back the borrowed resources once the request finishes its execution. The second form, called *total lending*, consists in suspending the application, lending all its resources to the request, and re-

suming it once the request finishes its execution. The third form of impact, called *giving*, consists in giving a subset of application resources to the request and finish its execution only with the remaining VMs. The three forms of impact may potentially lead the applications to exceed their agreed deadlines by a given delay. Obviously, the estimation of the delay depends on the form of impact. Note that the rigid applications may be impacted only with a *total lending* while the elastic applications may undergo the three forms of impact. In the next subsections we explain in more detail how we estimate the delay of each of the rigid and elastic applications. Based on the delay and the used revenue function we calculate the possible penalty of each application (see Section 5.3.3).

We calculate the group's donating bid similarly to what we do for calculating the group's waiting bid. As shown in Algorithm 7, we calculate the applications penalty as well as their impact. The application impact is calculated here as a ratio between the delay and the deadline (see Equation 5.6). Then, we perform an ascending sort of the applications according to their penalties. The group's donating bid may be obtained from only one application that holds enough private resources for the request or from a number of applications. In the second case the group's donating bid is calculated as the sum of penalties of all the applications to impact. The number of applications to be impacted for providing the bid is counted in order to provide the percentage of impacted applications if this group's bid is selected.

$$A_{impact} = \frac{A_{delay}}{A_{deadline}} \times 100 \tag{5.6}$$

5.4.2.1 Rigid Applications Delay

To estimate the delay of a rigid application, we consider impacting the application with a *total lending* impact form. Thus, the application is suspended during a requested duration that includes the request runtime and the time for transferring the resources to the request and getting them back later. Thus, we first compute the application's spent time from its submission until the time the request is received, (see Figure. 5.5). Then, we deduce the remaining time as the predicted application's runtime minus the spent time. Afterwards, we estimate the time when the application ends its execution as the remaining time plus the requested duration and compare it to the deadline. If the estimated application's end time is before the deadline, then the delay is zero. Otherwise the delay is calculated as the difference between the end time and the deadline.

5.4.2.2 Elastic Applications Delay

To estimate the delay of an elastic application we first have to select the correct form of impact. If the application holds the requested number of resources or less, the application may only be impacted with a *total lending* and its delay is calculated in the same way as that of a rigid application. Otherwise, if the application holds more resources than requested, we estimate its delay that corresponds to the *partial lending* and the *giving* impact forms. Then, we select the form of impact according to the smallest corresponding penalty. Initially for both impact forms we calculate the spent time and the remaining time of the application, as we do for rigid applications. However, the remaining time of the application changes in a different way according to the impact form, see Figure. 5.5. With the *giving* form, we assume that the predictor plugin is able

Algorithm 7: Donating bid of computational applications' group

Data: $R_{missing}$, $req_{runtime}$, $A_{threshold}$
Result: $G_{donating\_bid}$, $G_{donating\_impact}$

$\forall A^i \in G_A$, calculate $A^i_{penalty}$ and A^i_{impact} ;
$sort(G_A, A^i_{penalty})$;
// initializing variables ;
$G_{donating\_bid} \leftarrow \infty; R_{found} \leftarrow 0; donating\_bid \leftarrow 0; donating\_impact \leftarrow 0$;
for $(i \leftarrow 1$ ***to*** $|G_A|)$ **do**
    if $(A^i_{private} \geqslant R_{missing})$ **then**
        if $(A^i_{penalty} \leqslant G_{donating\_bid}) \wedge (A^i_{impact} \leqslant A_{threshold})$ **then**
            $G_{donating\_bid} \leftarrow A^i_{penalty}$;
            $donating\_impact \leftarrow 1$;
            break ;
        end
    else
        if $(A^i_{impact} \leqslant A_{threshold})$ **then**
            $R_{found} + \leftarrow A^i_{private}$;
            $donating\_bid + \leftarrow A^i_{penalty}$;
            $impact + +$;
            if $(R_{found} \geqslant R_{missing})$ **then**
                $G_{donating\_bid} \leftarrow donating\_bid$;
                break ;
            end
        end
    end
end
$G_{donating\_impact} \leftarrow \frac{(\frac{G_{impact}}{100} \times |G_A|) + donating\_impact}{|G_A|}$;
return $G_{donating\_bid}$, $G_{donating\_impact}$;

to predict the application remaining time based on the progress of the application execution, the number of remaining resources, and the application performance model. With the *partial lending* form, the predictor estimates the application runtime according to two phases. In the first phase, during the requested duration, the application runs with the remaining resources and in the second phase, after getting back the borrowed resources, the application runs with all its resources.

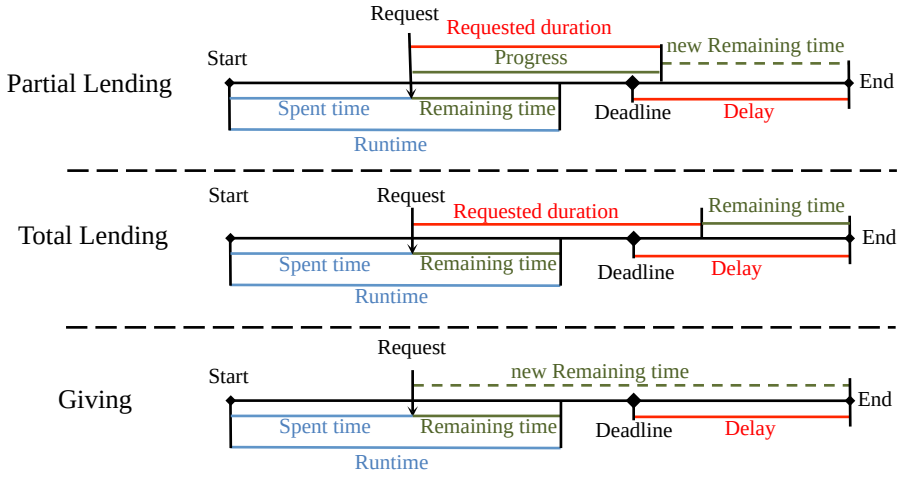


Figure 5.5: Application times according to the three impact forms.

5.5 Summary

In this chapter we investigated computational applications, proposed SLA metrics and functions, and presented the corresponding donating and waiting bid heuristics for maximizing the PaaS provider profit. Basically, when the PaaS system receives a request for hosting a new application and private resources are not readily available, each application type-specific group is asked to propose donating and waiting bids for providing a number of resources. We proposed a computational applications waiting bid heuristic that estimates the required time for applications to finish their execution and calculates the possible penalty of the new application accordingly. Then, it tries to provide the waiting time generating the smallest possible penalty of the new application. We also proposed a computational applications donating bid heuristic that calculates the bid based on the possible penalties of the running applications providing some of their resources. We defined three forms to be used by computational applications for providing resources during their execution. The first form, *partial lending*, consists in lending a part of running applications resources to the new application and continue running with the remaining resources until the new application gives back the borrowed resources. The second form, *total lending*, consists in suspending the running applications during the execution of the new application and resuming them once the new application finishes its execution. The third form, *giving*, consists in giving a part of the running applications resources to the new application and continue their execution only with the remaining resources. The *partial lending* and *giving* forms of impact may be used only by elastic applications. Each form leads to a different penalty for applications . Thus, the heuristic

selects the impact form causing the smallest penalty.

In the next chapter we present the design and implementation of an open cloud-bursting PaaS system able to incorporate the proposed profit optimization model as well as the computational applications heuristics presented in this chapter.

Chapter 6

Meryn: an Open Cloud-Bursting PaaS

Contents

In the two previous chapters we presented a set of heuristics providing a complete view of our profit optimization model for PaaS providers. This model requires to be incorporated in a real PaaS system providing support to specific features, in order to be able to evaluate its efficiency. In this chapter we present *Meryn*, our PaaS design and architecture, providing all the features required by our profit optimization model.

This chapter is organized as follows. Section 6.1 discusses the design principles of Meryn. Section 6.2 describes the Meryn system architecture. Section 6.3 presents the implementation details of the Meryn prototype. Finally, Section 6.4 summarizes this chapter.

6.1 Design Principles

Our main goal in designing a PaaS system architecture is to provide support for the main required features enabling the provider to optimize its profit. Specifically, as discussed in Section 3.1, to achieve our objectives we need a PaaS system supporting cloud bursting and extensibility regarding application types. The majority of existing PaaS systems do not provide such features. They either focus on specific application types or rely on specific infrastructures. Therefore, we made three key design decisions in designing our PaaS system, called *Meryn*[1].

The first decision concerns the resource granularity. We chose to use virtual machines (VMs) as the basic resource unit running applications. Thus, each application is separately hosted on at least one virtual machine. Our choice is motivated by the well-know advantages of virtualization in terms of flexible resource control and isolation. Moreover, as we aim at being extensible to host several application types, the use of virtual machines is more advantageous than the use of system containers because it enables hosting applications with different operating system requirements (find more details in Section 2.4.2.2). Furthermore, using this approach private resources and resources dynamically leased from public clouds can be managed in a similar way, which greatly simplifies the resources management as well as the deployment and accounting of the resources consumed by each application.

The second decision concerns the use of existing programing frameworks to manage the supported application types. A great variety of frameworks have proven to be widely useful and are currently part of PaaS offerings, including web application frameworks, MapReduce, batch frameworks, and task framing frameworks. These frameworks employ sophisticated scheduling and resource management policies, optimized to support the quality objectives of framework-based applications. The key idea here is to facilitate the extensibility of the PaaS system with the support of new application types with minimal effort.

The final decision concerns the division of resource management responsibilities between the frameworks and the PaaS system. We adopt a decentralized architecture in which the frameworks collaborate though exchanging resources with the aim of increasing the provider profit. Compared to a centralized architecture, the advantage of this approach is that it imposes minimal changes on the frameworks, thus facilitating the extensibility of the PaaS system. Indeed, the frameworks continue to take most of the resource management decisions regarding their hosted applications, taking advantage of their application type-specific knowledge to better satisfy their SLA objectives.

6.2 System Architecture

In this section we present the system architecture of Meryn, our open cloud-bursting PaaS system. First, we present a high level overview. Then, we provide more details about the system components, the application life-cycle and and used scaling mechanisms.

[1]Meryn is an Arabic word (مرن) that means Flexible and Elastic.

6.2.1 Overview

The overall Meryn architecture is illustrated in Figure 6.1, where private resources consist in a fixed number of VMs shared between multiple elastic Virtual Clusters (VCs). We define a VC as a set of VMs running on private resources plus possibly some VMs rent from public IaaS clouds. Each VC is associated with an application type-specific group and is managed by a specific framework. For example, it is possible to use the Oracle Grid Engine (OGE) framework to manage a VC that hosts batch applications, and the Hadoop framework to manage a VC that hosts MapReduce applications.

Note that for each framework we create a set of customized VM disk images with the suitable software stack, used to create VMs of the corresponding VC. Each VC has an initial amount of resources. Then, according to specific policies VCs may exchange the private resources among each other. The overall resource management is fully decentralized where each VC autonomously manages its own resources and decides when to burst into public cloud resources. The end users submit their applications through a common and uniform interface, whatever the type of their applications.

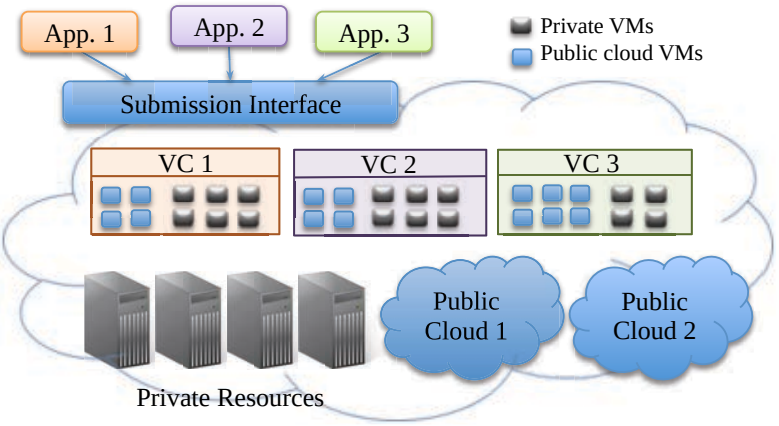


Figure 6.1: Meryn architecture overview

6.2.2 Components

The main components of the Meryn system, shown in Figure 6.2, are a *Client Manager*, a *Cluster Manager* for each VC, an *Application Controller* for each hosted application and a *Resource Manager*.

Most frameworks consist of one master daemon and a set of slave daemons. In Meryn, we run the master daemon separately on one or more private VMs, called *Master VM(s)*, and the slave daemons either on private VMs or public cloud VMs, called *Slave VMs*. The master VMs are shared between all the applications hosted in the same VC and run the Cluster Manager component as well as the Application Controllers of the applications running on the VC. The slave VMs are dedicated to applications and run their corresponding code. In the following we give more details about each Meryn component.

- **Client Manager.** The Client Manager is the entry point of the system providing users with a uniform submission interface. It is responsible for receiving submission requests and transferring them to the corresponding Cluster Manager. It

also enables users to get the results of their applications. Meryn may have several Client Managers in order to avoid a potential bottleneck, which could happen in peak periods.

- **Cluster Manager.** The Cluster Manager plays the role of the application type-specific manager and consists of two parts, a generic part and a framework-specific part. The generic part is the same for all Cluster Managers and consists in managing resources and deciding when to horizontally scale up and down. Specifically, the Cluster Manager has to decide when releasing resources to other VCs, when to acquire resources from other VCs and when to rent resources from public clouds. The framework-specific part depends on the hosted type of applications as well as the used framework. This part wraps the interface of the framework to enable a standardized interaction between the Client Manager and the Cluster Managers of all the supported application types. The wrapping implementation translates for example the submission request template of the Client Manager to a submission request template compatible with the framework. Moreover, depending on the hosted application type, this part implements a QoS predictor plugin and proposes corresponding SLA metrics accordingly. Consequently, the extension of Meryn with the support of a new application type requires three terms: (1) creating a new VC, (2) managing the VC and the applications using the corresponding framework, and (3) implementing the application type-specific part of the Cluster Manager.

- **Application Controller.** The Application Controller is responsible for monitoring the execution progress of its associated application as well as the satisfaction of its agreed SLA. This component is mainly based on mechanisms provided by the framework.

- **Resource Manager.** The Resource Manager provides support to functions for initially deploying the PaaS system and for transferring VMs from one VC to another one. The Resource Manager interacts with a VM manager software, such as OpenStack [5], OpenNebula [123] or Snooze [86].

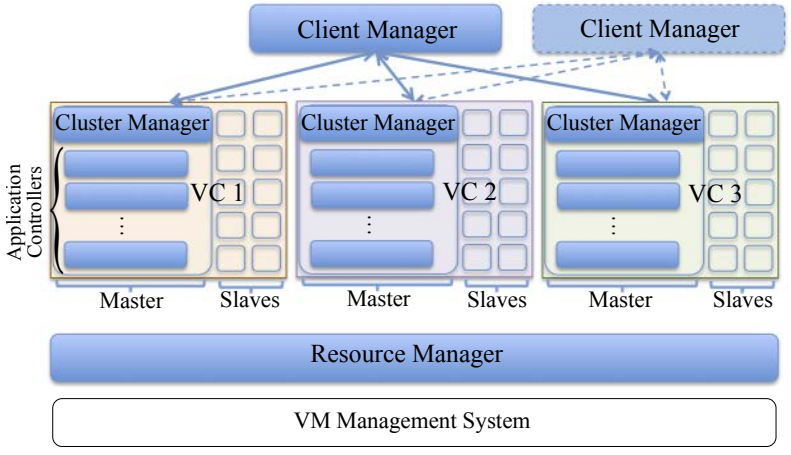


Figure 6.2: Meryn components

6.2.3 Application Life-Cycle

The application life-cycle is shown in Figure 6.3. The process starts when the user contacts the Client Manager and gives a description of her application using a standardized template provided by the submission interface. The application description should mainly characterize the application in terms of resource consumption requirements as well as software dependencies. Based on the application description the Client Manager determines its corresponding VC and acts as an intermediary between the user and the Cluster Manager. The Cluster Manager and the user negotiate the SLA terms, through the Client Manager, until a mutual agreement. Then, the user transfers the executable file and the needed input data of the application to the corresponding VC.

The Cluster Manager translates the application description to a template compatible with the corresponding framework, launches a new Application Controller instance, and submits the application to the framework. The Application Controller monitors the application progress and regularly checks the satisfaction of its agreed SLA. If an SLA violation is detected, the Application Controller computes the corresponding penalty and informs the Cluster Manager. Finally, the Client Manager provides a way for the user to retrieve the results of her application from the Cluster Manager.

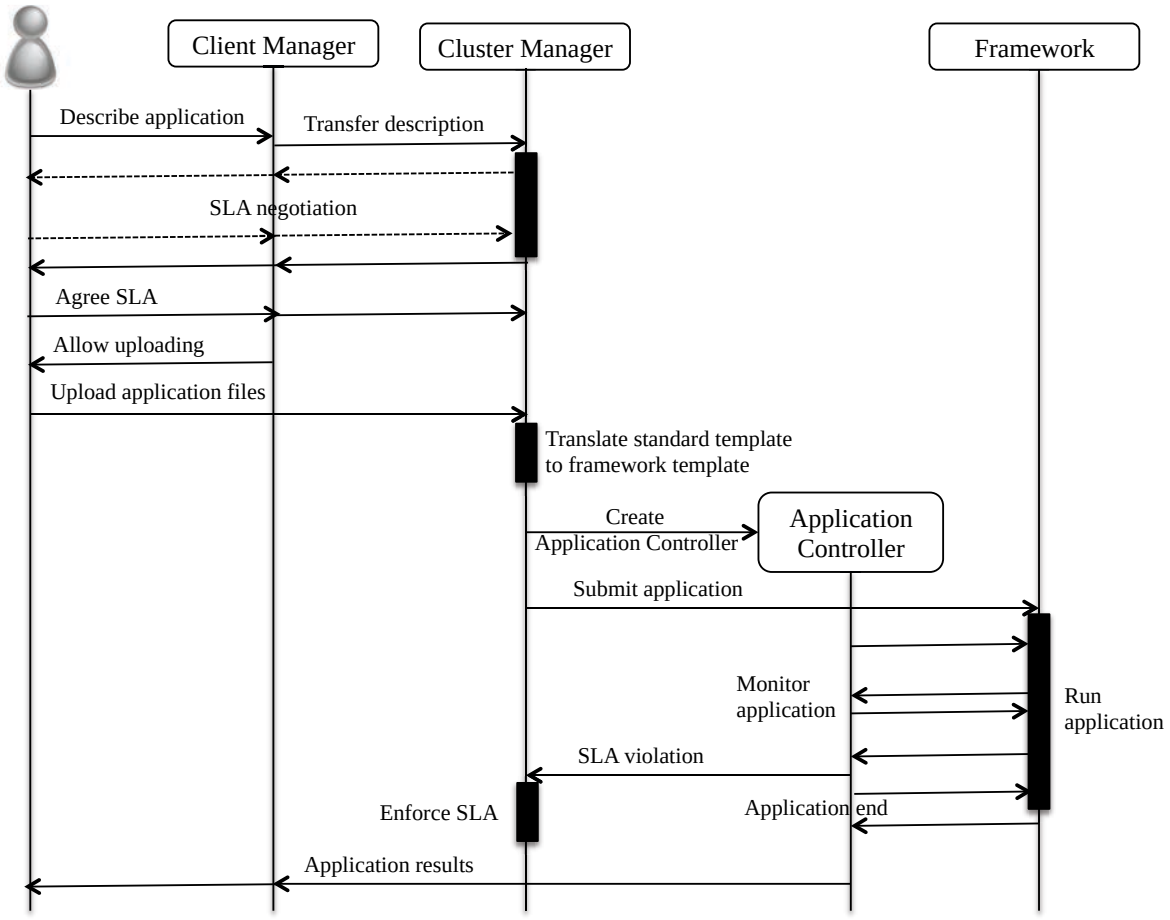


Figure 6.3: Application life-cycle

6.2.4 VC Scaling Mechanisms

Horizontally scaling up a VC is performed either using private VMs taken from the other VCs or using public clouds VMs. Below we describe the used technical mechanisms for each option.

Obtaining VMs from other VCs. The transfer of VMs from a *source VC* to a *destination VC* operates as follows. First, the Cluster Manager of the source VC selects the VMs to remove, removes them from the framework and requests the Resource Manager to shut them down. Then, the Cluster Manager of the source VC informs the Cluster Manager of the destination VC about the availability of the VMs. The Cluster Manager of the destination VC requests the Resource Manager to start new VMs with the corresponding VM disk image, gets their corresponding IP addresses, configures them and adds them to its framework resources.

Obtaining VMs from Public Clouds. The cloud bursting mechanism of a VC operates as follows. First, the Cluster Manager of the corresponding VC requests a selected public cloud to create the VMs, gets their corresponding IP addresses, configures them and adds them to its framework resources. When the VC finishes using the public VMs, its Cluster Manager removes the VMs from the framework and asks the corresponding public cloud to stop them. Note that before enabling a VC to burst into public clouds, we first make sure that the corresponding VM disk images are saved in the set of public clouds that may be used.

6.3 Implementation

We have developed a Meryn prototype in about 4,000 lines of shell script. The prototype is built upon the Snooze VM manager software for virtualizing the private physical resources. The prototype provides support to batch and MapReduce applications using respectively Oracle Grid Engine OGE 6.2u7 and Hadoop 0.20.2 frameworks. In the following we describe the configuration of each framework, the implementation details of each Meryn component, the management of parallel submissions, and the implementation of the bursting mechanism.

6.3.1 Frameworks Configuration

The two used frameworks have different structures and mechanisms for scheduling jobs. The OGE framework operates as any classic batch scheduler (as seen in Figure 2.1), where the Hadoop framework has specific configurations and scheduling mechanisms for running data intensive applications. Therefore, we configured differently each framework such that we get more control over the assignment of resources to applications.

OGE. We set the OGE virtual cluster such that it consists of one master node running the *sgemaster* daemon and a number of slave nodes, each node running an *sgeexec* daemon. The slave nodes are set such that each node cannot host more than one application. However, one application may run on multiple slave nodes. Furthermore, the slave nodes are set to belong to a unique queue that handles all the submissions. In order to control the submitted jobs spend in the queue, our policy specifies the hostname of the slave node(s) to use for hosting the new job .

Hadoop. The default Hadoop scheduler performs a fair assignment of resources to jobs. For instance, when there is only one job running in the cluster, the job uses all the resources. If other jobs are submitted some resources are assigned to the new jobs, so that each job gets on average the same amount of CPU time. This scheduling mechanism, performed by the *JobTracker* daemon, makes it complex to predict the application runtimes. Therefore, we setup our Hadoop virtual cluster to have multiple sub-clusters, such that each sub-cluster runs a separate *JobTracker* daemon and hosts only one application. Each application sub-cluster consists on at least one node running the *JobTracker* and *TaskTracker* daemons. When required by the application, its sub-cluster may hold additional nodes running the *Tasktracker* daemon. Furthermore, we configured the Hadoop storage nodes to be shared between all the hosted applications. The storage nodes consists in one node running the *NameNode* daemon and a set of nodes running the *DataNode* daemon to store data and ensure replicas. In our Meryn prototype we run the Cluster Manager and Application Controllers components on the node running the *NameNode* daemon. Finally, the free nodes belonging to the VC do not have any particular function and do not run any daemon. When a new MapReduce application needs to be hosted, we configure and start daemons in a number of free nodes according to the application requirements.

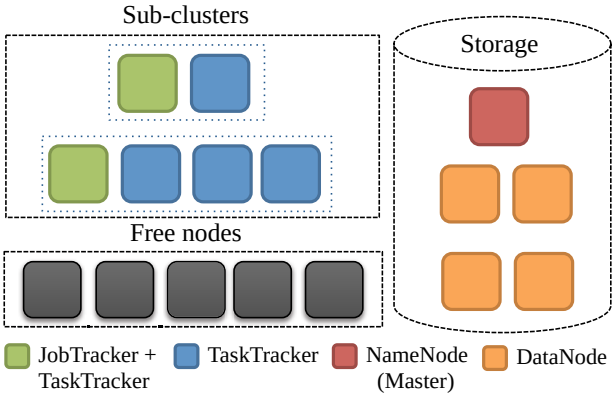


Figure 6.4: Configuration of the Hadoop VC

6.3.2 Components Implementation

In the following we present the important implementation aspects of each Meryn component.

Client Manager. We implemented a command line interface for submitting applications where the client has to fill a submission template, such as the example shown in the Extract 6.1. The main required fields are the application type (batch or MapReduce), the number of processes, the path to the source code file and possibly the required parameters. Note that the path should be accessible by the Client Manager for uploading the source code. The Client Manager transfers the information provided in the template to the corresponding Cluster Manager, using the Secure Copy Protocol (SCP). The required interaction between the client and the Cluster Manager to negotiate SLA is also performed using files exchanges between the Cluster Manager and the Client Manager with SCP. After an SLA agreement, the Client Manager transfers the template file as well

as the application source files to the Cluster Manager for performing the submission.

Extract 6.1: Example of job submission template

```
# User Input
Type=mapreduce
Nb_processes=1
Source_file=/tmp/hadoop/benchmarks.jar
Parameters="Benchmark2 /data/visits/ output/aggre101/ -m 10 -r 3"

# SLA
Deadline=1230
Price=.08668800
Class=Medium
```

Cluster Manager. First, to enable a framework to add a new VM, the Cluster Manager starts on the new VM the corresponding daemon and informs the master daemon about its availability. Inversely to remove a VM, the Cluster Manager asks the master daemon to remove it from the list of its slave nodes, then it optionally stops the daemon running on the VM. Then, to standardize the basic actions performed on the applications such as start/stop and suspend/resume, the Cluster Manager wraps the functionalities provided by each framework's management tool with common commands. Finally, the Cluster Manager implements the policies described in Chapters 4 and 5 for making decisions about resources provisioning and assignment, in about 400 lines of Java code and interacts with them through synchronized read/write operations in XML files.

Application Controller. The Application Controller regularly gets the state of its corresponding application using commands provided by the programing framework and checks its number of resources. We implemented an Application Controller specific to computational applications that also checks the elapsed runtime of applications and compares it to their agreed deadline. If the agreed deadline is violated, the Application Controller informs the Cluster Manager. If the real application completion time exceeds its expected deadline, the Application Controller computes the corresponding penalty and informs the Cluster Manager.

Resource Manager. The Resource Manager interacts with Snooze through its Libcloud API for creating and removing VMs either for initially deploying the VCs of the PaaS system or upon Cluster Manager request for VMs exchange.

In the current deployment, we host the Client Manager and Resource Manager components in one physical node, hosting also some Snooze daemons and not used for creating virtual machines. In a production deployment, we can host separately the Client Manager in a front-end web node.

6.3.3 Parallel Submission Requests

In order to handle requests submitted in parallel and provide consistent resource assignment, we implement two mechanisms. First, we use mutual exclusion mechanisms to access the parts of code checking or modifying the resources availability and status. Second, we use a *reservation* flag to tag resources (1) under SLA negotiation, (2) waited for by an application or (3) lent to an application for a specific duration that should be

reassigned to their initial application. The tagged resources may not be assigned to any application until they are untagged.

6.3.4 Cloud Bursting

To enable each VC to burst into public clouds, we set in a specific config file of the Cluster Manager the IP addresses and the user account credentials of the entry point of each supported public cloud provider. Currently, we deploy a second instance of the Snooze VM manager in remote resources accessed over a wide-area network from the deployed Meryn prototype.

6.4 Summary

In this chapter we presented the design, architecture and implementation of Meryn, our open cloud-bursting PaaS system. The architecture relies on private virtual machines and may burst into virtual machines from public cloud providers. To enable Meryn to support extensibility regarding application types, we make use of independent virtual clusters. Each virtual cluster is managed using an existing framework dedicated to a specific application type. In the next chapter we evaluate our profit optimization model using the Meryn prototype.

Chapter 7

Evaluation

Contents

In this chapter we evaluate the contributions of this PhD thesis. The purpose of our evaluations is to show the efficiency of the proposed profit optimization policies in increasing the provider profit as well as in satisfying SLAs and QoS properties required by the clients. This chapter is organized as follows. Section 7.1 describes our evaluation setup, specifying the configuration parameters of the Meryn prototype and optimization policies, and describing the used workloads, pricing model, SLA classes, revenue functions, and testbed. It also defines the used evaluation metrics. Section 7.2 presents a first evaluation based on simulations and discusses the corresponding results. Section 7.3 presents a second evaluation based on experiments and discusses the corresponding results. Finally, Section 7.4 summarizes this chapter.

7.1 Evaluation Setup

This section discusses the evaluation setup used in both the simulations and the experiments. We describe the setup of the Meryn prototype, the profit optimization policies and the submitted workloads. We also present the used configuration of resources costs and prices as well as SLA classes and revenue functions.

7.1.1 Meryn Prototype

As shown in Figure 7.1, we configured the Meryn prototype with two VCs, one VC for MapReduce applications managed with the Hadoop framework and one VC for batch applications managed with the OGE framework. We used a VM instance model similar to the Amazon EC2 medium instance[1], which consists of 2 CPUs and 3.75 GB of memory. The Snooze managed private cloud is configured to have 100 VMs. Thus, we assigned 50 private VMs to each VC. We assume that the number of VMs available in the Snooze managed public cloud, able to be rent as public VMs, is infinite.

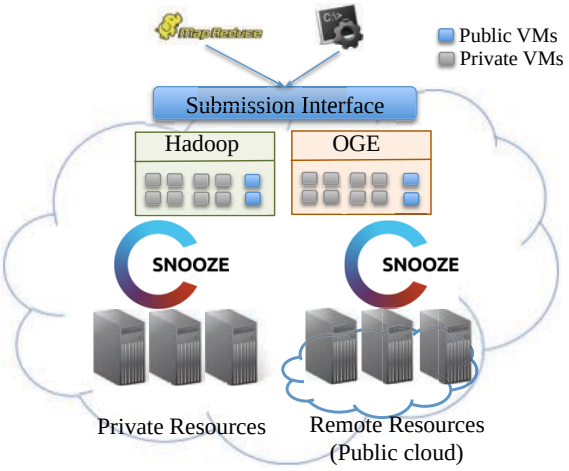


Figure 7.1: Meryn prototype configuration.

7.1.2 Policies

We evaluate and compare the efficiency of the three policies described in Chapter 4: basic, advanced and optimization. The basic policy statically partitions the private resources between the groups and hosts applications using public resources each time the group has no more available private resources. The advanced policy offers more flexibility than the basic policy. It enables the exchange of resources between the application type-specific groups and the deployment of applications simultaneously on private and public resources. The optimization policy goes one step further than the advanced policy. In order to optimize the provider profit, it considers either impacting the QoS level of running applications to host a new application or impacting the QoS level of the new application due to waiting for private resources to become available. Accordingly, we define two threshold setups for both the ratio of applications in the PaaS system that

[1]http://aws.amazon.com/ec2/instance-types/ [Online; accessed February 2013]

may be impacted and the allowed impact on the impacted applications. In the first setup, referenced henceforth as *opt1*, the threshold of applications ratio able to be impacted is set to 20% and the threshold of the allowed applications impact is set to 50%. In the second setup, referenced henceforth as *opt2*, the threshold of applications ratio able to be impacted is set to 50% and the threshold of the allowed applications impact is set to 100%. Overall, the opt2 policy is more aggressive than the opt1 policy in impacting applications.

7.1.3 Workloads

We submit to the Meryn prototype simultaneously two workloads, one for the batch VC and one for the MapReduce VC. The batch workload follows the Lublin workload model [118]. To adapt this workload model to our scenario, we consider the number of nodes in each Lublin request as the number of VMs in a job and limit this number to 128 simultaneous VMs for one job. In order to generate a shorter workload we changed the following parameters of the Lublin workload model: the number of jobs from 1000 to 100, the maximum job runtime from two days to one hour, and the maximum inter-arrival time from five days to half an hour.

The MapReduce workload is based on the distribution of job sizes and inter-arrival times seen at Facebook over a week in october 2009, reported in [155]. The workload consists of 100 jobs randomly submitted over 25 minutes with a mean inter-arrival time of 14s. The jobs are instances of the four queries of Hive benchmarks [48]: *text search (grep)*, *selection*, *aggregation*, and *join*. Basically, there are ten job instances submitted a number of times such that the total number of submitted jobs is 100. Each job instance runs one of the four Hive queries and uses a specific number of mappers and reducers. To determine the runtime of each job instance in the workload, we generated 2 GB of Grep data, 3 GB of Rankings data, and 2 GB of UserVisits data. Then, we separately measured the average runtime of each job instance using a randomly predefined number of TaskTracker VMs.

We generated three instances of each workload, batch and MapReduce, and randomly combined them to get three workloads (workload 1, workload 2, and workload3). Each workload consists of one batch workload instance and one MapReduce workload instance to be submitted together.

7.1.4 Pricing

In our pricing model we assume a per-second billing, following the current trend for increasingly shorter billing periods in cloud platforms [53]. To calculate the cost and revenue of applications we defined a cost for public cloud VMs, a cost for private VMs, and a VM price as follows. The cost of a public cloud VM is based on the per-hour Amazon EC2 pricing of a standard medium VM instance in the Ireland datacenter[2] divided by 3600 (the number of seconds per hour), giving 0.00003612$/s. The cost of a private cloud VM is based on the per hour power cost of one core, reported in [144]. The reported cost is converted to dollars (with an exchange rate of 1.35), multiplied by 2 (as medium VM instances have 2 cores), and divided by 3600 (the number of seconds per hour), giving 0.0000064497$/s. The VM price for end users is calculated as the cost

[2]http://aws.amazon.com/ec2/pricing/ [Online; accessed June 2013]

of a public cloud VM multiplied per 2, giving 0.00007224$/s. This is justified by the fact that our system offers an extra service compared to IaaS providers.

7.1.5 SLAs

First, we define three SLA classes: *high*, *medium*, and *low*. We randomly distribute the SLA classes among jobs in workloads as follows. We set 10% of jobs with a high SLA class, 70% of jobs with a medium SLA class and 20% of jobs with a low SLA class. Furthermore, we defined a different deadline margin, price factor, and α value in the linear and bounded linear revenue functions for each SLA class.

In the bounded linear revenue function, we compute a lower revenue bound for each application according to its SLA class and initial price. Specifically, we define a lower revenue percentage compared to the initial price of applications for each SLA class, based on which we deduce the minimum revenue bound for each application. The value of the maximum penalty of each application is then calculated as the application initial price minus the value of the application lower revenue bound.

In the step revenue function, we defined three penalty levels corresponding to three delay levels. The delay levels are computed for each application as a percentage of the agreed deadline and are common for all applications and all SLA classes. The first defined level is included between zero and 20% ($0 < level\ 1 \leqslant 20\%$). The second level is included between 20% and 50% ($20\% < level\ 2 \leqslant 50\%$). Finally, the third level is greater than 50% ($50\% < level\ 3$). For each SLA class and delay level we define a different penalty where penalties are calculated as a percentage of the application initial price.

All the defined values for each SLA class are shown in Table 7.1.

Table 7.1: Configuration parameters of SLA classes.

	High	Medium	Low
Deadline margin	1	2	3
Price factor	3	2	1
Linear SLA: α	2	5	20
Bounded linear SLA: min revenue percentage	10%	20%	30%
Step SLA: L1 delay percentage	15%	5%	0%
Step SLA: L2 delay percentage	30%	15%	5%
Step SLA: L3 delay percentage	50%	25%	10%

7.1.6 Grid'5000 testbed

Grid'5000 [40] is a testbed supporting experiment-driven research in large scale parallel and distributed computing. It provides 5000 cores geographically distributed on 10 sites in France (Bordeaux, Grenoble, Lille, Luxembourg, Lyon, Nancy, Reims, Rennes, Sophia-Antipolis, Toulouse) plus one site abroad in Porto Alegre, Brazil. All sites in France are connected through a 10 Gb/s backbone.

We carried out our simulations and experiments on the Grid'5000 testbed. According to our requirements and availability of resources, we have used resources from three clusters of the Rennes site (*paradent*, *paradent*, and *parapluie*) and two clusters of the Nancy site (*griffon* and *graphene*). The *paradent* cluster consists of 64 Carri System CS-5393B nodes supplied with 2 Intel Xeon L5420 processors (each with 2 cores at 2.5GHz), 24 GB of memory, and Gigabit Ethernet network interfaces. The *parapide* cluster consists of 25 SUN FIRE X2270 nodes supplied with 2 Intel Xeon X5570 processors (each with 2 cores at 2.93 GHz), 24 GB of memory, and Gigabit Ethernet network interfaces. The *parapluie* cluster consists of 40 HP Proliant DL165 G7 nodes supplied with 2 AMD Opteron(tm) 6164 HE processors (each with 6 cores at 1.7 GHz), 48 GB of memory, and Gigabit Ethernet network interfaces. The *griffon* cluster consists of 92 Carri System CS-5393B nodes supplied with 2 Intel Xeon L5420 processors (each with 2 cores at 2.5GHz), 16 GB of memory, and Gigabit Ethernet network interfaces. The *graphene* cluster consists of 144 Carri System CS-5393B nodes supplied with 1Intel Xeon X3440 processors (with 4 cores at 2.53 GHz), 16 GB of memory, and Gigabit Ethernet network interfaces.

7.1.7 Evaluation Metrics

In this section we describe a set of evaluation metrics used to show to which extent (1) the PaaS provider profit is optimized and (2) the deadlines of computational applications are violated.

- **Profit.** The profit of a workload is calculated as the sum of profits of all the submitted jobs from that workload. The profit of one job is calculated as the job's revenue minus the job's cost.

- **VMs usage proportions.** The VMs used for hosting each application are obtained either from the private or public IaaS resources (*public VMs*). The private resources may be obtained either (1) from available VMs on the same VC where the application is hosted (*local VMs*), (2) from available VMs on a VC different than the one hosting the application (*VC VMs*), (3) from running applications, from any VC (*donating VMs*), or (4) by waiting the execution end of running applications, from any VC (*waiting VMs*). The amount of VMs used from each source is calculated as a percentage compared to the total number of VMs used in the workloads.

- **Completion time.** The completion time of a workload is the elapsed time between the arrival of the first job in the workload and the completion of the final job in the workload (not necessarily the last submitted one). The goal of this metric is to show the overhead of the optimization policies.

- **Deadline violations.** The deadline violations are presented in two ways. The first way consists in presenting the ratio of the delayed applications in the workload by counting the number of applications missing their deadlines. The second way consists in presenting the average delay percentage of the delayed applications by computing the delay percentage of each delayed application.

7.2 Simulations

In this section we evaluate our policies through a series of simulations. The objective of these simulations is to show the behavior of the policies in mapping the applications on the resources for optimizing the provider profit. The interest of performing simulations in our context is the limited amount of needed resources, in contrast to real experiments. In the following we describe the simulations environment setup, then we present and analyze the results.

7.2.1 Environment Setup

To perform the simulations we used the implemented Meryn prototype with the following modifications.

1. No VC is really created. Instead, we create a different folder for each VC containing the corresponding Cluster Manager component, the applications information as well as their corresponding Application Controller components.

2. We removed the interaction of the Resource Manager with Snooze. Thus, instead of really creating and removing VMs we simply modify the values of the number of used and available VMs of the VCs in the corresponding configuration files.

3. The submitted jobs are treated as follows. First we find VMs for hosting them and switch the VMs states from available to busy. Then, instead of really running the applications we create a process that sleeps for the entire application runtime. Finally, when the sleep process ends we switch the state of VMs from busy to available. We simulate the possibility that applications' execution time is different from their predicted runtime by adjusting the sleep time

4. We run all Meryn components (Client Manager, Resource Manager, Cluster Managers and Application Controllers) in only one physical machine.

Using this prototype, we measured the submission time of an application with different resource availability scenarios and different VMs requirements (from 1 to 100 VMs). We define the submission time as the required time from the submission of an application in the PaaS system to the time the application really starts running. The application submission time varied from one second to 13 seconds. Thus, for homogeneity reasons and to leave a margin, we set the processing time of all the submitted applications to 15 seconds. Note that the difference between the submission time and the processing time of an application is that the processing time should be the same whatever is the source of the used VMs, because in this thesis we consider that the deadline proposed to a specific application requiring a specific number of VMs and a specific SLA class is always the same. We also measured the required time for an application to lend one or more VMs and to get them back (without counting the runtime of the application borrowing the VMs). This loan duration varied from 5 seconds to 16 seconds, depending on the number of the lent VMs (measured with 1 to 50 VMs). Therefore, we left a margin and set this parameter to 20 seconds.

We carried out our simulations on the Grid'5000 testbed using nodes from the *paradent* and the *parapide* clusters of the Rennes site.

7.2.2 Results

We evaluate the optimization policies, opt1 and opt2, with the three defined revenue functions: opt1 with linear revenue function (Lopt1), opt2 with linear revenue function (Lopt2), opt1 with bounded linear revenue function (Bopt1), opt2 with bounded linear revenue function (Bopt2), opt1 with step revenue function (Sopt1), and opt2 with step revenue function (Sopt2). We also evaluate the basic and advanced policies, without specifying any revenue function because in these policies we do not intentionally impact applications. Thus, in the simulations there is practically practically no deadline violation. We performed three simulations using each policy. In each simulation we submit a different workload instance. In the following we compare the obtained results from the eight polices according to the aforementioned evaluation metrics.

Each simulation presented here was performed five times and the results are presented as means ($\pm$ standard deviations in the bars).

7.2.2.1 Profit

Figure 7.2 shows the generated profit in the three workloads using the basic, advanced and Sopt2 policies. We show only the Sopt2 optimization policy in the figure for visibility reasons because it is the one that generates the best profit for the provider in three workloads, from 2.64% to 4.98% more profit than the advanced policy. The basic policy generates the worst provider profit. Then, the advanced policy generates from 1.91% to 6.32% more profit than the basic policy. The provider profit generated using the other optimization policies is between the one generated using the advanced policy and the one generated using the Sopt2 policy, as seen in Tables 7.2 and 7.3. In Table 7.2 we show the rate of profit gains obtained with all the polices compared to the basic one and in Table 7.3 we show the rate of profit gains obtained with all the optimization policies compared to the advanced one. The optimization rates differ from one workload to another one. The general observation is that the opt2 policies generate a little more profit than the opt1 policies using the same revenue function, from 0.06 % to 0.77%. Moreover, the optimization policies using a bounded linear revenue function generate more profit than the same policies using the linear revenue function, up to 1.02% in opt1 using workload1; and the optimization policies using a step revenue function generate more profit than the same policies using the bounded linear revenue function, up to 1.59% in opt2 using workload1. These differences between policies in the generated profit vary inversely with the amount of the used public cloud resources, as seen in Figure 7.3 and described in the next subsection.

7.2.2.2 VMs Usage Proportions

Figure 7.3 shows the used VMs proportions for each workload and policy. Each proportion in the figure represents a percentage of the used VMs compared with the total VMs used. Each of the total used VMs and the proportions are calculated as the number of VMs multiplied by the VMs usage duration. We clearly see in the figure that the policies using the less public cloud VMs in all workloads are ordered as follows: Sopt2, Sopt1, Bopt2, Bopt1, Lopt2, Lopt1, then the advanced policy. The basic policy is the one which uses the more public cloud VMs. The Sopt2 policy uses up to 82.61% less public

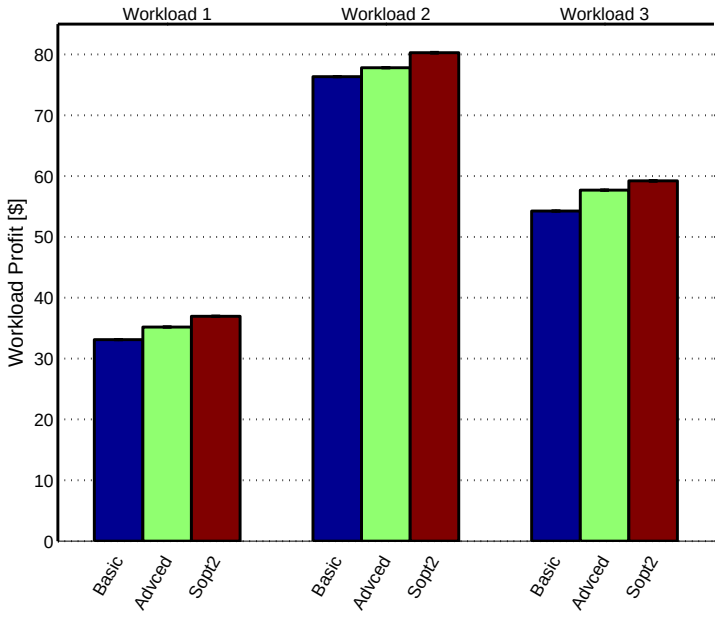


Figure 7.2: Workload profit comparison. Profit shown is the sum of profits of all jobs in the workload. (Simulations)

Table 7.2: Profit Rates of the advanced and the optimization policies compared to the basic policy. (Simulations)

	Workload 1	Workload 2	Workload 3
Advanced	6.29 %	1.91 %	6.32 %
Linear Opt 1	8.77 %	3.87 %	8.37 %
Linear Opt 2	9.54 %	4.37 %	8.44 %
Bounded Opt 1	9.79 %	4.68 %	8.57 %
Bounded Opt 2	10 %	4.77 %	8.64 %
Step Opt 1	11.06 %	4.95 %	9.08 %
Step Opt 2	11.59 %	5.14 %	9.14 %

cloud resources compared to the basic policy (find more details in Table 7.4). This usage scheme of public cloud VMs is directly reflected in the overall provider profit.

We notice that in the optimization policies the *donating VMs* are more used than the *waiting VMs*. The selection of one option, donating or waiting, depends on the loan duration, the runtime and SLA class of the new application and the remaining execution time of the running applications. In our evaluation the order and arrival time of the applications in the workloads as well as their corresponding SLA classes are random. However, the configured loan duration in these simulations (20 seconds) is relatively smaller than the average applications runtime (around 350 seconds). Thus, often the donating bid is cheaper than the waiting bid.

Table 7.3: Profit Rates of the optimization policies compared to the advanced policy. (Simulations)

	Workload 1	Workload 2	Workload 3
Linear Opt 1	2.33 %	1.92 %	1.92 %
Linear Opt 2	3.06 %	2.41 %	1.99 %
Bounded Opt 1	3.29 %	2.71 %	2.11 %
Bounded Opt 2	3.48 %	2.80 %	2.17 %
Step Opt 1	4.48 %	2.98 %	2.59 %
Step Opt 2	4.98 %	3.17 %	2.64 %

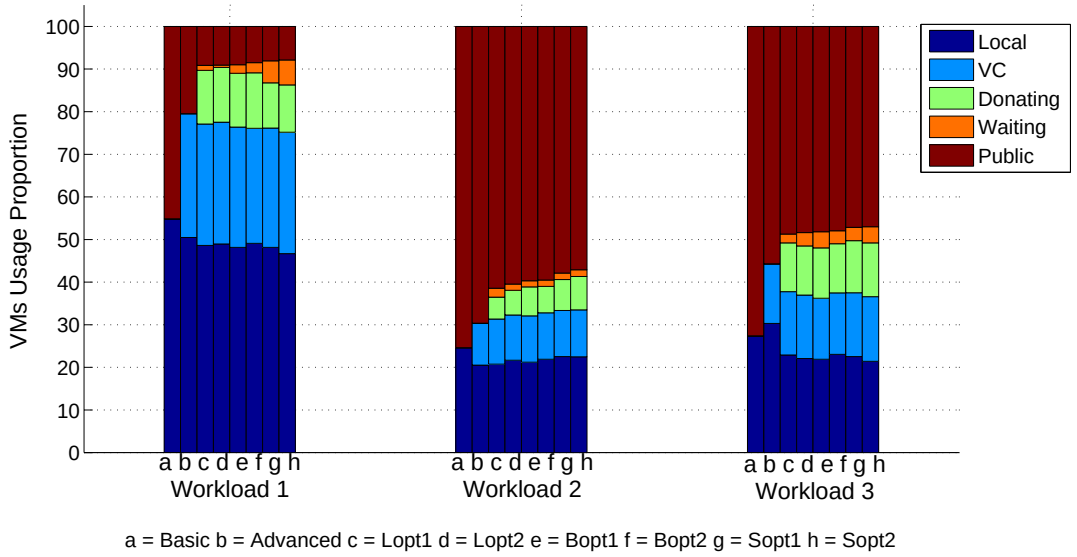


Figure 7.3: VMs usage proportion for each workload and policy, calculated as the number of the used VMs multiplied by the usage duration. (Simulations)

7.2.2.3 Completion Time

Figure 7.4 shows the completion time of the three workloads. We see that the workload completion time in the basic and advanced policies is almost the same in all the workloads. However, the workload completion time in the optimization policies is slightly higher, especially in workload 1 (up to 5.43% with the Sopt2 policy) and workload 2 (up to 2.13% with the Lopt1 policy). This overhead in workloads completion time is mainly due to the incurred delay of the impacted applications and the required time for calculating the bids.

Indeed, the required time for calculating the bids varies according to the number of requested VMs as well as the number of running applications and their used VMs. In our simulations the bid calculation time varied from 6 seconds to 20 seconds with outliers values up to 345 seconds. The median bid calculation time is 7 seconds which corresponds to the most frequent case.

Table 7.4: Percentage of the used public cloud VMs. (Simulations)

	workload 1	workload 2	workload 3
Basic	45.13 %	75.38 %	72.58 %
Advanced	20.46 %	69.62 %	55.68 %
Linear Opt 1	9.13 %	61.44 %	48.73 %
Linear Opt 2	9.07 %	60.46 %	48.37 %
Bounded Opt 1	8.98 %	59.68 %	48.16 %
Bounded Opt 2	8.51 %	59.51 %	47.95 %
Step Opt 1	8.06 %	57.87 %	47.14 %
Step Opt 2	7.85 %	57.11 %	47 %

7.2.2.4 Deadline Violations and Impact

As expected, in these simulations the deadlines of all applications in the three workloads were satisfied with the basic and the advanced policies. With the optimization policies, there are some applications that exceeded their deadlines in each workload, in order to improve the provider profit. Nevertheless in all the workloads the ratio of the delayed applications and the average delay percentage of delayed applications did not go beyond their respective defined thresholds. As it can be observed in Table 7.5, the ratio of the impacted applications is between 3.5% and 7.75% in the opt1 policy and between 2% and 7% in the opt2 policy. The average delay of the delayed applications is between 35.33% and 47% in opt1 policy and between 35.6% and 54.17% in the opt2 policy. We notice that both of the ratio of impacted applications as well as the average delay of delayed applications do not reach the respective defined thresholds. This is because impacting many applications with a high delay while considering the payment of penalties is not beneficial for optimizing the provider profit. We also notice that there are more impacts with the optimization policies using the step revenue function. This is because the penalty calculation in the step revenue function is more advantageous for the provider.

Table 7.5: Percentage of (A) delayed applications and (B) average delay of delayed applications with the optimization policy. (Simulations)

	workload 1		workload 2		workload 3	
	A	B	A	B	A	B
Lopt1	4.5 %	35.33 %	3.5 %	45.41 %	7.5 %	36.6 %
Bopt1	5 %	35.9 %	4 %	46.74 %	6 %	36.75 %
Sopt1	5.75 %	36 %	5.5 %	47 %	7.75 %	36.94 %
Lopt2	2 %	38.8 %	2.5 %	54.17 %	4 %	35.6 %
Bopt2	6 %	36 %	4.75 %	48.8 %	7 %	36.87 %
Sopt2	5 %	35.73 %	6.5 %	45 %	6 %	35.66 %

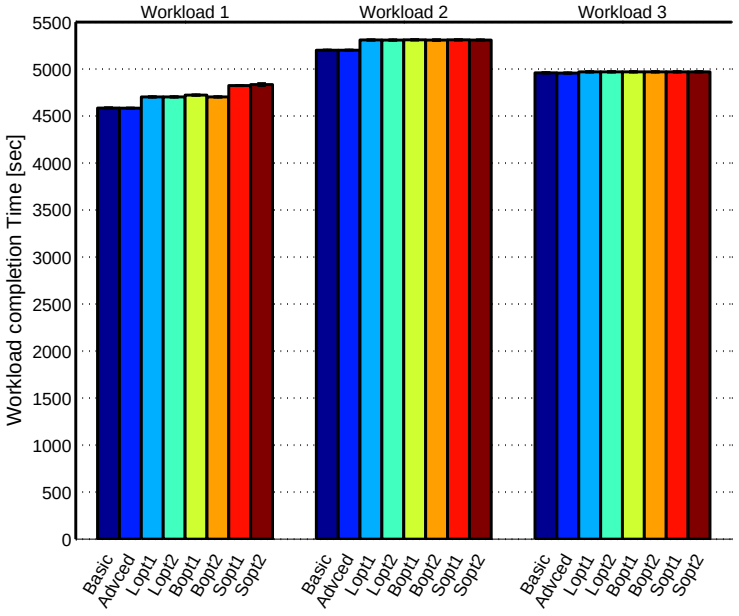


Figure 7.4: Workloads completion time (seconds), from the submission of the first job to the completion of the final job. (Simulations)

7.2.2.5 Conclusion

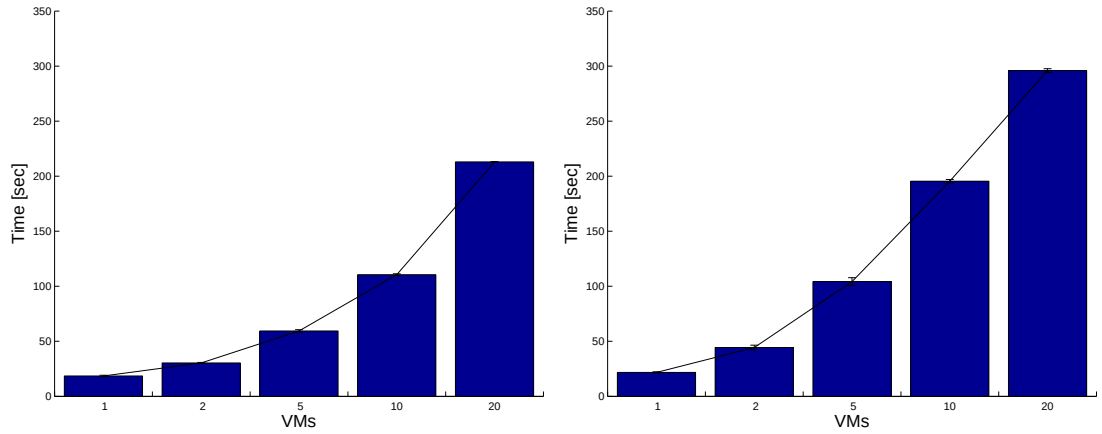
In this section we evaluated our optimization policies using different revenue functions through a set of simulations and compared them to the basic and the advanced policies. Our results show that the optimization policies generates more profit for the provider than the basic and the advanced policies with a small overhead on the workloads completion time. The comparison between to the instances of the optimization policies, opt1 and opt2, shows that the possibility to impact more applications and with more delay is not highly exploited because of the corresponding payment of penalties. Finally, the comparison between the optimization policies using different revenue functions shows that when the penalty calculation is more advantageous for the provider, the profit is greater and the applications are more impacted.

7.3 Experiments

In this section we evaluate our profit optimization policies through a series of experiments. The experiments' goal is to see the behavior of our policies in a real PaaS environment. In the following we present some measurements and describe the experimental environment setup, then we describe and analyze the results.

7.3.1 Measurements

In this section we measure the time for creating and configuring private and public VMs. Then, we measure the time for processing the submission of a batch and MapReduce applications on (1) available private resources, (2) private resources transferred from another VC, and (3) public cloud resources. The objective of these measurements is to configure our experimental environment.



(a) Private VMs (b) Public VMs

Figure 7.5: Creation and configuration of OGE VMs

To perform these measurements, we deployed a private instance of Snooze to host Meryn on 10 nodes from the *paradent* and *parapluie* clusters of the Rennes site, and deployed a public instance of Snooze on 14 nodes from the *griffon* and *graphene* clusters of the Nancy site.

7.3.1.1 VM(s) Creation and Configuration

Figure 7.5 shows the required time for creating and configuring 1, 2, 5, 10, and 20 private and public VMs of the OGE VC. Figure 7.6 shows the required time for creating and configuring 1, 2, 5, 10, and 20 private and public VMs of the Hadoop VC. We used a VM instance model similar to the Amazon EC2 medium instance[3], which consists of 2 CPUs and 3.75 GB of memory. With the two frameworks OGE and Hadoop, the required time for creating public cloud VMs is greater than the required time for creating private VMs. This is mainly because of the network latency between the two remote sites.

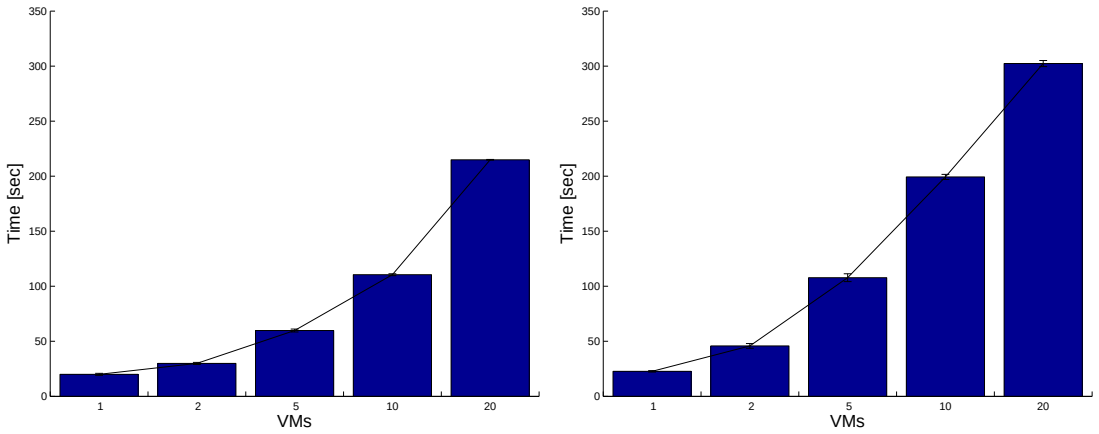
We tried to parallelize most of the operations to create and configure multiple VMs. However, for consistency reasons, many configuration operations consist in modifying specific shared configuration files which requires the use of mutual exclusion. Thus, the required time for creating multiple VMs is approximately linear.

We also measured the required time for removing separately a local and a public cloud VM from both the OGE and the Hadoop programming frameworks and shutting the VM down. With both frameworks and with the private and the public VM, this operation takes around 2 seconds. We didn't measure the removal of multiple VMs because the framework's remove operation requires the name or the IP address of the specific VM to remove. Thus, if we need to remove multiple VMs we should do it sequentially.

7.3.1.2 Job Submission

We measure the submission time of batch and MapReduce applications on (1) available private resources (*local VMs*), (2) private resources transferred from another VC (*VC*

[3]http://aws.amazon.com/ec2/instance-types/ [Online; accessed February 2013]



(a) Private VMs (b) Public VMs

Figure 7.6: Creation and configuration of Hadoop VMs

VMs), and (3) public cloud resources (*public VMs*). We define the submission time as the required time from the submission of an application in the PaaS system to the time the application really starts running.

Figure 7.7 shows the submission time of batch applications requiring 1, 2, 5, 10 and 20 VMs, on *local VMs* (Figure 7.7(a)), *VC VMs* (Figure 7.7(b)) and *public VMs* (Figure 7.7(c)). Figure 7.8 shows the submission time of MapReduce applications requiring 1, 2, 5, 10 and 20 VMs, on *local VMs* (Figure 7.8(a)), *VC VMs* (Figure 7.8(b)) and *public VMs* (Figure 7.8(c)).

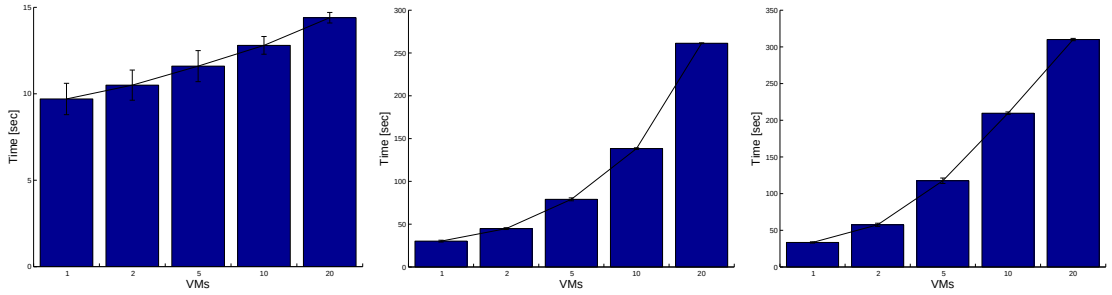
Our first observation is that with both types of application and the three sources of VMs (local, VC and public) the applications submission time increases with the increase of the number of VMs required by the applications. Our second observation is that the applications submission time varies according to the source of VMs and the greatest application submission time values are obtained with the use of *public VMs*.

7.3.2 Environment Setup

Our first setup concerns the batch applications. As it is difficult to find real batch applications matching the applications properties described in the used batch workload model, we implemented a simple sleep MPI application taking as input a number of processes and a sleep duration.

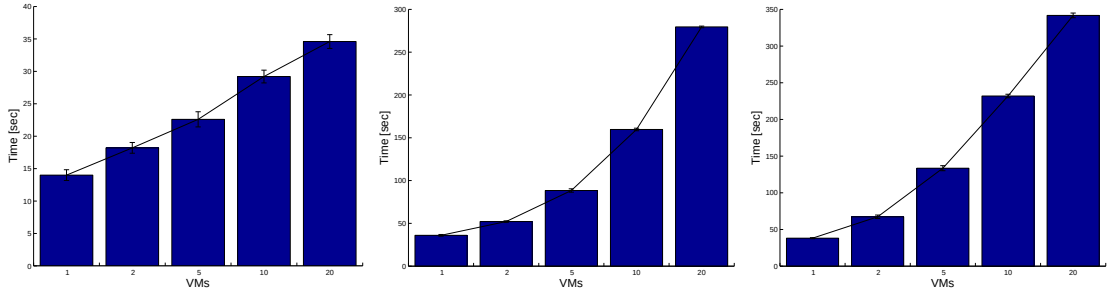
Our second setup concerns the definition of a function for computing applications processing time which is used to compute the application deadline, as seen in Section 5.3. The difference between the submission time and the processing time of an application is that the processing time should be the same whatever is the source of the used VMs, because in this thesis we consider that the deadline proposed to a specific application requiring a specific number of VMs and a specific SLA class is always the same. Thus, for all applications we calculate the processing time according to the submission time values obtained with the source of VMs requiring the maximum time, namely with the *public VMs*.

The average application submission time on *public VMs* of batch and MapReduce applications requiring from 1 to 20 VMs is shown in Table 7.6. To define a function



(a) Local VMs (b) VC VMs (c) Public VMs

Figure 7.7: Submission time of batch applications on (a) local VMs, (b) VC VMs, and (c) public VMs



(a) Local VMs (b) VC VMs (c) Public VMs

Figure 7.8: Submission time of MapReduce applications on (a) local VMs, (b) VC VMs, and (c) public VMs

for calculating the applications processing time we make use of these values and the polyfit function of the NumPy Python library [46]. The polyfit function gets a sample of points (x: number of VMs, y: submission time) and returns a vector of coefficient for the polynomial function. Thus, we define a processing time function for batch and MapReduce applications as seen in Equations 7.1, where A_R is the number of resources required by the application.

$$Batch: \ A_{processing} = -0.4877 \times (A_R)^2 + 25.34 \times A_R + 8.482 \tag{7.1a}$$

$$MapReduce: \ A_{processing} = -0.5462 \times (A_R)^2 + 27.72 \times A_R + 13.83 \tag{7.1b}$$

Our final setup concerns the definition of a function for computing the VM loan duration. This duration depends if the loan is between applications from the same VC or between applications from different VCs.

In the first scenario the VM loan duration includes (1) the time for suspending/shrinking the *source* application, (2) the time for cleaning up the VM(s), (3) the submission time of the new application, (4) the time for cleaning up the VM(s) after the end of the new application (e.g., removing the new application files), and (5) the time for resuming/extending the source application. We measure the required time for each step for both batch and MapReduce applications, as seen in Table 7.7. The time for suspending a batch application is around 2 seconds and the time for shrinking or suspending a MapReduce application is around 1 second. The time for cleaning up an OGE VM is null (because of our simple sleep MPI applications).The time for cleaning up a Hadoop TaskTracker VM is around 1 second (for stopping the corresponding daemons and reconfiguring the application sub-cluster). The submission time of the new application on local VMs is measured in the previous section and the average values are shown in Table 7.8. Based on these values and using the polyfit function of the NumPy library we define a function for calculating the submission time of batch and MapReduce applications on local VMs, see Equations 7.2. The required time for cleaning up the VM(s) after the end of a batch application is null and after the end of a MapReduce application is around 1 second. Finally, the time for resuming a batch application follows the same batch local submission function and the time for extending a MapReduce application also follows the same MapReduce local submission function where the function depends on the number of VMs to add to the impacted application A_{lend_R} rather than A_R. Therefore, we use Functions 7.3 to calculate the required time for loaning VMs between applications from the same VC.

$$\text{Batch local submission time} = -0.007576 \times (A_R)^2 + 0.4025 \times A_R + 9.944 \tag{7.2a}$$

$$\text{MapReduce local submission time} = -0.0625 \times (A_R)^2 + 2.349 \times A_R + 12.973 \tag{7.2b}$$

$$\text{Batch local\_VMs\_loan} = 2 + 2 \times (-0.007576 \times (new\_A_R)^2 + 0.4025 \times new\_A_R + 9.944) \tag{7.3a}$$

$$\text{MapReduce local\_VMs\_loan} = 3 + (-0.0625 \times (new\_A_R)^2 + 2.349 \times new\_A_R + 12.973) + (-0.0625 \times source\_A_{lend\_R})^2 + 2.349 \times source\_A_{lend\_R} + 12.973) \quad (7.3b)$$

In the second scenario the VM loan duration between applications from different VCs includes (1) the time for suspending/shrinking the source application (2) the time for removing the VM(s) from the source VC and creating new VM(s) in the destination VC and submitting the new application, and (3) removing the VM(s) from the destination VC after the end of the new application, creating VM(s) in source VC and resuming/extending the source application. We measure the required time of each step for both batch and MapReduce applications, as seen in Table 7.9. The first step is similar to the one in the first scenario. The second and the third steps are similar to processing the submission of an application on private VMs from a different VC (*VC VMs*), which was reported in Section 7.3.1 and is shown in Table 7.10. Based on these values and using the polyfit function of the NumPy library we define a function for calculating the required time for submitting batch and MapReduce applications on VC VMs, as seen in Equations 7.4. Therefore, we use functions 7.5 to calculate the required time for loaning VMs between applications from the different VCs. Note that Function 7.5a is used by MapReduce applications lending VMs for a new batch application and inversely in Function 7.5b.

$$\text{Batch VC submission time} = 0.02187 \times (A_R)^2 + 11.8 \times A_R + 20.22 \quad (7.4a)$$

$$\text{MapReduce local submission time} = -0.0894 \times (A_R)^2 + 14.97 \times A_R + 21.68 \quad (7.4b)$$

$$\text{Batch vc\_VMs\_loan} = 1 + (0.02187 \times (new\_A_R)^2 + 11.8 \times new\_A_R + 20.22) + (-0.0894 \times (source\_A_R)^2 + 14.97 \times source\_A_R + 21.68) \quad (7.5a)$$

$$\text{MapReduce vc\_VMs\_loan} = 2 + (-0.0894 \times (new\_A_R)^2 + 14.97 \times new\_A_R + 21.68) + (0.02187 \times (source\_A_R)^2 + 11.8 \times source\_A_R + 20.22) \quad (7.5b)$$

To carry out these experiments, we deployed a private instance of Snooze to host Meryn on 20 nodes from the *parapluie* cluster of the Rennes site, and deployed a public instance of Snooze on 70 nodes from the *griffon* cluster of the Nancy site.

7.3.3 Results

Due to the high number of resources required to carry out the experiments, here we evaluate only the opt2 instance optimization policy and compare it with the advanced the basic policies. We evaluate three instances of the opt2 policy, each instance uses a specific revenue function: opt2 with linear revenue function (Lopt2), opt2 with bounded linear revenue function (Bopt2), and opt2 with step revenue function (Sopt2). We do not specify a revenue function in the basic and the advanced polices because there would be

Table 7.6: Average applications submission time on public VMs.

	1 VM	2 VMs	5 VMs	10 VMs	20 VMs
Batch [sec]	33.4	57.5	117.5	209.6	310
MapReduce [sec]	38.1	67.3	133.5	231.9	341.8

Table 7.7: Required time for processing a local VM(s) loan.

	(1)	(2)	(3)	(4)	(5)
Batch [sec]	2	null	equation 7.2a	null	equation 7.2a
MapReduce [sec]	1	1	equation 7.2b	1	similar to equation 7.2b

Table 7.8: Average applications submission time on local VMs.

	1 VM	2 VMs	5 VMs	10 VMs	20 VMs
Batch [sec]	9.7	10.5	11.6	12.8	14.4
MapReduce [sec]	14	18.2	22.6	29.2	34.6

Table 7.9: Required time for processing a VC VM(s) loan.

	(1)	(2)	(3)
Batch [sec]	2	equation 7.4a	equation 7.4b
MapReduce [sec]	1	equation 7.4b	equation 7.4a

Table 7.10: Average applications submission time on VC VMs.

	1 VM	2 VMs	5 VMs	10 VMs	20 VMs
Batch [sec]	30.2	44.8	79	138.3	261.4
MapReduce [sec]	36	52.1	88.4	159.7	279.5

no deadline violation with the used environment setup. We perform three experiments using each policy. In each experiment we submited a different workload instance. In the following we compare the obtained results from the six policies according to the aforementioned evaluation metrics.

Each experiment presented here was performed three times and the results are presented as means ($\pm$ standard deviations in the bars).

7.3.3.1 Profit

Figure 7.9 shows the profit generated in the three workloads using each policy. The results are quite similar to the ones obtained with simulations. The main noticed difference is that with the experiments all the policies generate less profit than with simulations For instance in the basic policy we generate 32.74$, 75.15$, and 53.46$ rather than 33.11$, 76.35$, and 54.26$ in respectively workload 1, workload 2 and workload 3. This is because in a real environment the VMs usage is longer due to the required configurations for hosting applications, which increases the costs of hosting applications. In Table 7.11 we show the rate of profit gains obtained with all the polices compared to the basic one and in Table 7.12 we show the rate of profit gains obtained with all the optimization policies compared to the advanced one. The advanced policy generates from 1.90% to 5.06% more profit than the basic policy and the Sopt2 policy generates from 2.99% to 4.29% more profit than the advanced policy. Similarly to simulations, these differences vary inversely with the amount of the used public cloud resources, as seen in Figure 7.10 and described in the next subsection.

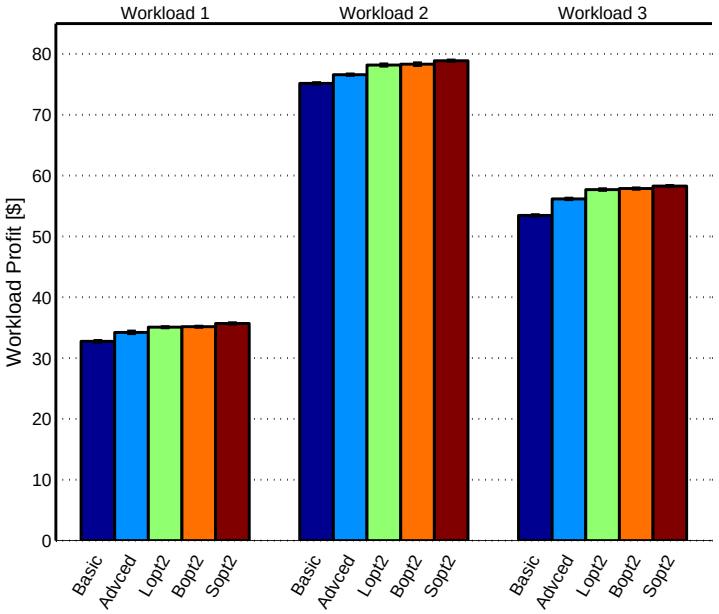


Figure 7.9: Workload profit comparison. Profit shown is the sum of profits of all jobs in the workload. (Experiments)

Table 7.11: Profit Rates of the advanced and the optimization policies compared to the basic policy. (Experiments)

	Workload 1	Workload 2	Workload 3
Advanced	4.53 %	1.90 %	5.06 %
Linear Opt 2	7.12 %	4.01 %	7.93 %
Bounded Opt 2	7.38 %	4.20 %	8.24 %
Step Opt 2	9.01 %	4.96 %	9.02 %

Table 7.12: Profit Rates of the optimization policies compared to the advanced policy. (Experiments)

	Workload 1	Workload 2	Workload 3
Linear Opt 2	2.48 %	2.07 %	2.73 %
Bounded Opt 2	2.72 %	2.26 %	3.03 %
Step Opt 2	4.29 %	2.99 %	3.77 %

7.3.3.2 VMs Usage Proportion

Figure 7.10 shows the used VMs proportions for each workload and policy. The optimization policies uses clearly less public cloud VMs than the basic and advanced policies. For instance, the Sopt2 policy uses up to 80.55% and 57.14% less public VMs than respectively the basic and the advanced policies. More details are shown in Table 7.13. Similarly to simulations, the usage scheme of public cloud VMs is directly reflected in the overall provider profit.

We notice that with the experiments in the optimization policies *waiting VMs* are more used than *donating VMs*. The reason is related to two factors. First, the VM loan duration is more significant than in the simulations which highly affects the completion time of the impacted applications with a donating bid as well as their penalties. Therefore the proposed donating bid is often higher than the waiting bid, and thus it is not selected. Second, the time for exchanging VMs between VCs is comparable to the average applications runtime in our workloads. Thus, waiting the end of the execution of running applications often takes a comparable time and with a more attractive cost.

Table 7.13: Percentage of the used public cloud VMs. (Experiments)

	workload 1	workload 2	workload 3
Basic	45.04 %	75.28 %	72.60 %
Advanced	20.44 %	69.59 %	55.78 %
Linear Opt 2	10.12 %	60.59 %	50.05 %
Bounded Opt 2	9.80 %	60.41 %	49.49 %
Step Opt 2	8.76 %	58.63 %	48.07 %

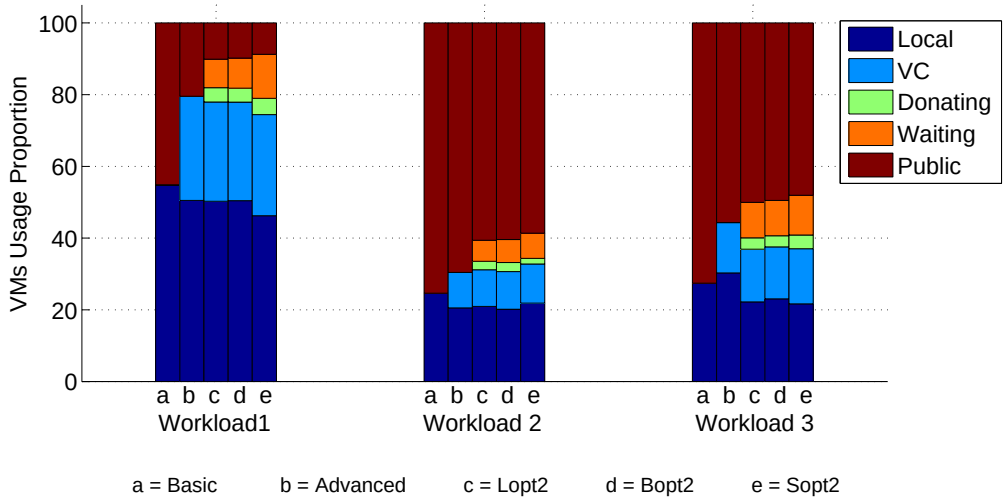


Figure 7.10: VMs usage proportion for each workload and policy, calculated as the number of the used VMs multiplied by the usage duration. (Experiments)

7.3.3.3 Completion Time

Figure 7.11 shows the completion time of the three workloads. Similarly to the simulations, the completion time of the workloads with the basic and the advanced policies are almost the same. However, workload completion time with the optimization policies is slightly higher, especially in workload 1 (up to 4.17% with the Sopt2 policy) and workload 2 (up to 6.63% with the Sopt2 policy). The workloads completion time overhead is due to the incurred delay of the impacted applications, the required time for calculating the bids, and the mechanisms for transferring VMs between VCs. In these experiments the bid calculation time varied from 14 seconds to 50 seconds with outliers values up to 876 seconds. The median bid calculation time is 23 seconds which corresponds to the most frequent case.

7.3.3.4 Deadline Violations

As expected, in the these experiments the deadlines of all applications in the three workloads were satisfied with the basic and the advanced policies. With the optimization policies some applications exceeded their deadlines. However, the ratio of delayed applications and the average delay percentage of delayed applications do not go beyond their respective thresholds. As it can be observed in Table 7.14, the ratio of the impacted applications is between 4% and 8.75% and the average delay of the delayed applications is between 39% and 65.52%. We notice that in the experiments the impact of applications is higher than the one in the simulations, mainly because of the overhead of the mechanisms for handling VMs.

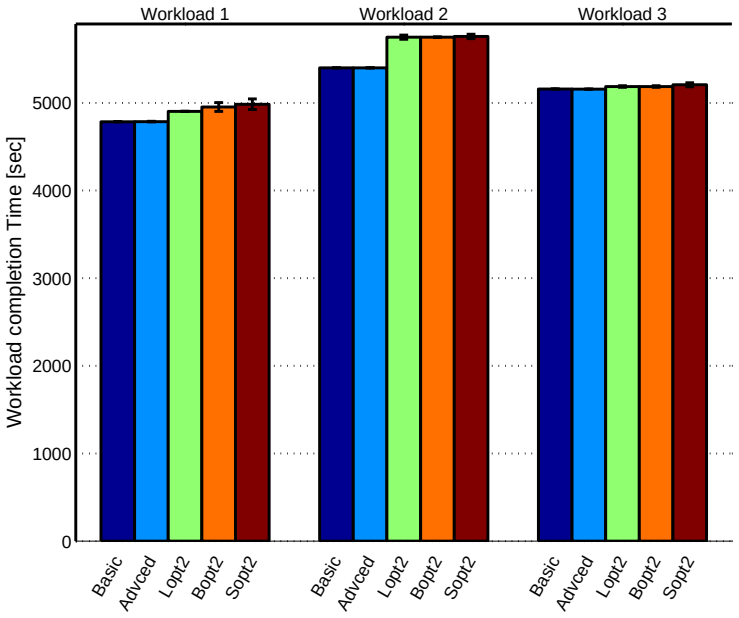


Figure 7.11: Workloads completion time (seconds), from the submission of the first job to the completion of the final job. (Experiments)

Table 7.14: Percentage of (A) delayed applications and (B) average delay of delayed applications with the optimization policy. (Experiments)

	workload 1		workload 2		workload 3	
	A	B	A	B	A	B
Lopt2	4 %	46.5 %	4 %	59.07 %	4.5 %	40.65 %
Bopt2	6.5 %	39 %	6.75 %	49.23 %	8 %	46.93 %
Sopt2	7 %	55.63 %	7.5 %	49 %	8.75 %	65.51 %

7.3.3.5 Conclusion

In this section we evaluated the opt2 optimization policy using different revenue functions through a set of experiments and compared it to the basic and the advanced policies. Our results show that the optimization policy generates more profit for the provider than the basic and the advanced policies. As a cost of this profit increase, the workloads completion time with the optimization policy is slightly higher and the deadline of some applications is violated. The results also show that all the policies generate less profit in the experiments than in the simulations.

7.4 Summary

In this chapter we presented the evaluation of our contributions through a series of simulations and experiments on the Grid'5000 testbed. We have used different workloads, SLA classes and revenue functions. We compared the efficiency of the optimization, basic and advanced policies in optimizing the provider profit and satisfying SLAs. The

results show that the optimization policies are able to improve the provider profit and significantly reduce the number of the used public cloud VMs. Moreover, the number of applications undergoing an impact in their QoS properties does not exceed the predefined thresholds. Furthermore, the results of the simulations and the experiments agree with each other, which validates the evaluation.

Chapter 8

Conclusions and Perspectives

Contents

This chapter summarizes the contributions of this PhD thesis and presents some future research directions.

8.1 Contributions

Cloud computing is an emerging paradigm revolutionizing the usage and marketing of Information Technology (IT). Nowadays, the socio-economic impact of cloud computing and particularly PaaS services becomes critical since the number of PaaS users and providers is growing. PaaS providers want to generate the maximum profit from the services they provide. This requires them to face a number of challenges such as efficiently managing the underlying resources and satisfying the SLAs of customer applications. This thesis addresses these challenges and focuses on optimizing the PaaS provider profit while considering the payment of penalties if the QoS level promised to the hosted applications is violated. The contributions of this thesis are summarized in the following.

Profit Optimization Model. In order to address the challenges related to optimizing the provider profit, this thesis considered a cloud-bursting PaaS environment and proposed a profit optimization policy that tries to find the cheapest resources for hosting applications. After each request submission, our optimization policy estimates the cost of hosting the new application using the private and the public resources and chooses the most profit-efficient option. During peak periods and unavailability of private resources, the policy tries to optimize the use of private resources with the consideration of two more options. The first option consists in taking resources from running applications while considering the payment of penalties if their promised QoS is impacted. The second option consists in evaluating the possible penalty of the new application due to waiting for private resources to become available. This policy cooperates with a set of application type-specific heuristics that provide the required information about the hosted applications.

Application of the Optimization Model. To show the applicability of our profit optimization model to a particular class of applications, we investigated rigid and elastic computational applications. We defined a deadline-based revenue function and presented two heuristics that collaborate with the profit optimization policy for maximizing the PaaS provider profit. The first heuristic provides a *waiting bid*, the smallest possible penalty of the new application due to waiting for private resources to become available. The second heuristic provides a *donating bid*, the smallest possible penalty of running applications due to providing some of their resources during their execution. We defined three possible forms that may be used by computational applications for providing resources during their execution. The first form consists in suspending the applications during the execution of the new application and resuming them once the new application finishes its execution and gives back the resources. The second form consists in lending a part of their resources to the new application and continuing to run with the remaining resources until the new application finishes and gives back the borrowed resources. The third form consists in giving a part of their resources to the new application and continuing their execution only with the remaining resources. The second and third forms may be used only by elastic applications. Each form leads applications to a different penalty. Thus, the heuristic selects the form of impact leading to the smallest possible penalty.

Meryn: an Open Cloud-Bursting PaaS. We designed and implemented a PaaS architecture, called *Meryn*, able to incorporate the profit optimization policy as well as different application type-specific heuristics. The Meryn architecture relies on virtualized private resources, using an IaaS manager solution, and supports bursting into public clouds when it is necessary. To support extensibility regarding application types, Meryn makes use of independent virtual clusters and manages each virtual cluster using an existing programming framework (e.g., OGE, Hadoop). Each programming framework is dedicated to a specific application type (e.g., batch, MapReduce). The private resources are dynamically shared between the virtual clusters according to specific policies and objectives, namely in our context according to the profit optimization policy. An application type-specific manager is hosted on each virtual cluster in order to provide and manage application SLAs.

To validate our contributions, we implemented a prototype that supports batch and MapReduce applications and used workload models based on real world traces. We performed a set of simulations and experiments on the Grid'5000 testbed. The results show that the proposed profit optimization model enables the provider to generates up to 11.59% and 9.02% more profit for the provider in respectively the simulations and experiments compared to a basic approach. As a cost of such an optimization, the completion time of workloads using our optimization model is slightly higher compared to the basic approach and the expected QoS level of some applications is impacted.

8.2 Perspectives

The work presented in this thesis, as any research work, is far from being complete or finished. Thus, several research directions are open for future work. Some of them aim at improving the proposed contributions, others at exploring new research areas. In the following we present the main identified perspectives.

Meryn Software. The Meryn software prototype was tested only on the Grid'5000 testbed. To make it available and useful for potential external users two improvement directions are possible. First, the Client Manager requires two additional features to be usable in a productive environment. The first feature consists in storing and managing user identities and their authentication credentials. The second feature consists in making the Client Manager more user-friendly through the implementation of a graphical web submission interface.

Second, the Meryn prototype should be tested on top of further IaaS managers, such as Open Stack, Eucalyptus and Nimbus in order to make it available to a larger community. For that reason, the Resource Manager needs to interact with IaaS managers through standard interfaces, such as the EC2 interface.

Optimization Model. The proposed profit optimization model makes three simplifications concerning PaaS environments. First, it considers that the PaaS system is built on top of homogenous private and public resources. However, both private and public resources may be heterogeneous, for example in the form of multiple VM instance types like in Amazon. Thus, the model should be extended to take into account the possibility of having resources with different capacities, which is more realistic. As a

result, the Cluster Manager component requires changes particularly for predicting the performance of applications and estimating their possible penalties for providing the donating and waiting bids.

Second, the proposed profit optimization model considers a per second resource billing model. However, the majority of current commercial providers propose a per hour resource pricing. Considering such pricing in our model leads to revise the price and cost functions for hosting applications. Moreover, the model should be extended to take into account the possibility to use idle public resources, which are already rented.

Finally, this thesis did not investigate the optimization of providers' profits during idle periods where the private resources are underutilized. The current behavior of our system in such a situation is to keep the resources idle. However, other actions are possible such as shutting down the idle resources to reduce their energy consumption and possibly their cost (as it was investigated in [122]), or selling the idle resources in the form of IaaS services (as it was investigated in [141]). Thus to complete our profit optimization model, a future direction would be to investigate the impact of such actions on the PaaS provider profit.

Evaluation. We have identified two possible future evaluations. The first evaluation consists in integrating an existing solution that predicts the runtime of computational applications such as the work presented in [133] and [73] (see Chapter 5). Then, we could measure the accuracy of predicting the runtime of real applications, and evaluate the impact of this accuracy on the PaaS provider profit. The second evaluation consists in measuring the effectiveness of the proposed policy in optimizing the profit with the use of several real public clouds with different characteristics and pricing models.

Application Types. The PaaS system and profit optimization approach proposed in this thesis are designed to be open and easily extensible to support new application types. In this thesis, we investigated rigid and elastic computational applications. A future direction would be to investigate further application types and associated programing frameworks and QoS properties. For instance to add support for workflow applications, it is possible to use existing workflow management systems such as Xerox [52], Apache ODE [51], or Activiti [50] for managing the workflow virtual cluster. In addition, the application type-specific part of our Cluster Manager component may implement the QoS specification and prediction of workflow applications proposed in research work, such as [71] and [96].

Energy Consumption. The energy consumption in cloud computing environments is growing and has become a major concern in the cloud computing community. A possible future direction of our work in this context is evaluating the impact of energy-saving actions on the performance and availability of virtual machines used by the Meryn system. Energy-saving actions, such as dynamically consolidating private VMs in the least number of physical machines and transitioning the idle physical machines into a lower power-state, may be provided by a VM manager such as Snooze [85]. Furthermore, it would be interesting to study PaaS management policies for minimizing the energy consumption of the hosted applications while meeting their SLAs, as already proposed for IaaS cloud environments [92]. Then, it would be possible to measure how the optimization of the energy consumption of applications impacts their costs.

Bibliography

[1] Microsoft Windows Azure Infrastructure. http://www.windowsazure.com/en-us/solutions/infrastructure/, 2013. [Online; accessed 30-November-2013]. 26, 35, 37, 39, 58, 137

[2] Rackspace: The Open Cloud Company. http://www.rackspace.com/, 2013. [Online; accessed 30-November-2013]. 26, 37

[3] Flexiant. http://www.flexiant.com/, 2013. [Online; accessed 30-November-2013]. 26

[4] The Apache Software Foundation. CloudStack: Open Source Cloud Computing. http://cloudstack.apache.org/, 2013. [Online; accessed 30-November-2013]. 26, 37

[5] OpenStack. http://www.openstack.org/, 2013. [Online; accessed 30-November-2013]. 26, 37, 88

[6] Microsoft Private Cloud. http://www.microsoft.com/en-us/server-cloud/solutions/virtualization-private-cloud.aspx#fbid=vbMp209OQ4l, 2013. [Online; accessed 30-November-2013]. 26

[7] Red Hat CloudForms. http://www.redhat.com/products/cloud-computing/cloudforms/, 2013. [Online; accessed 30-November-2013]. 26

[8] IBM SmartCloud Orchestrator. http://www-03.ibm.com/software/products/en/smartcloud-orchestrator/, 2013. [Online; accessed 30-November-2013]. 26

[9] HP CloudSystem. http://www8.hp.com/us/en/business-solutions/solution.html?compURI=1079455#tab=TAB2, 2013. [Online; accessed 30-November-2013]. 26, 37

[10] Amazon AWS Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/, 2013. [Online; accessed 16-Decembre-2013]. 14, 27, 36, 39, 137

[11] Amazon AWS Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/, 2013. [Online; accessed 16-Decembre-2013]. 27, 36, 39, 137

[12] Hitachi Data Systems' Virtual Storage Platform (VSP). http://www.hds.com/products/storage-systems/hitachi-virtual-storage-platform.html, 2013. [Online; accessed 30-November-2013]. 26

[13] Microsoft Windows Azure Cloud Services. http://www.windowsazure.com/en-us/services/cloud-services/, 2013. [Online; accessed 16-Decembre-2013]. 14, 27

[14] Google App Engine. https://cloud.google.com/products/app-engine/, 2013. [Online; accessed 16-Decembre-2013]. 27, 35, 39, 58

[15] Heroku. https://www.heroku.com/, 2013. [Online; accessed 16-Decembre-2013]. 14, 27, 36, 39, 137

[16] SalesForce Platform (Force.com). http://www.salesforce.com/platform/overview/, 2013. [Online; accessed 16-Decembre-2013]. 27, 36, 39

[17] Amazon AWS. Cloudbursting - Hybrid Application Hosting. http://www.appscale.com/, 2014. [Online; accessed 20-January-2014]. 27, 42, 44

[18] Cloudify. http://www.cloudifysource.org/, 2014. [Online; accessed 20-January-2014]. 27, 42, 44, 58

[19] VMware Cloud Foundry. http://www.cloudfoundry.com/, 2013. [Online; accessed 17-Decembre-2013]. 27, 41, 44, 58

[20] WSO2 Stratos. http://wso2.com/cloud/stratos/, 2014. [Online; accessed 20-January-2014]. 27

[21] RightScale. http://www.rightscale.com/, 2013. [Online; accessed 16-Decembre-2013]. 27, 37, 39, 58

[22] Scalr. http://www.scalr.com/, 2013. [Online; accessed 16-Decembre-2013]. 27

[23] IBM SAN Volume Controller (SVC). http://www-03.ibm.com/systems/storage/software/virtualization/svc/, 2013. [Online; accessed 30-November-2013]. 26

[24] Dell Cloud Manager. https://www.enstratius.com/, 2013. [Online; accessed 17-Decembre-2013]. 27

[25] RedHat OpenShift. https://www.openshift.com/, 2013. [Online; accessed 17-Decembre-2013]. 27, 42, 44, 58

[26] Google Apps. http://www.google.com/intl/en/about/products/, 2013. [Online; accessed 17-Decembre-2013]. 27

[27] SalesForce CRM. https://www.salesforce.com/crm/, 2013. [Online; accessed 17-Decembre-2013]. 27

[28] iCloud. https://www.icloud.com/, 2013. [Online; accessed 17-Decembre-2013]. 27

[29] High-Performance Computing Cluster (HPCC). http://hpccsystems.com/Why-HPCC/How-it-works, 2013. [Online; accessed 27-Decembre-2013]. 18

[30] European Grid Infrastructure (EGI). http://www.egi.eu/, 2013. [Online; accessed 27-Decembre-2013]. 19

[31] Globus. https://www.globus.org/, 2013. [Online; accessed 28-Decembre-2013]. 19

[32] Engine Yard. https://www.engineyard.com/, 2014. [Online; accessed 15-January-2014]. 37, 39

[33] DataCore SANsymphony. http://www.datacore.com/Software/Products/SANsymphony-V.aspx, 2013. [Online; accessed 30-November-2013]. 26

[34] CloudBees. http://www.cloudbees.com/platform/, 2014. [Online; accessed 15-January-2014]. 14, 37, 40, 137

[35] dotCloud. https://www.dotcloud.com/index.html, 2014. [Online; accessed 15-January-2014]. 14, 38, 40, 137

[36] AppFog. https://www.appfog.com/, 2014. [Online; accessed 15-January-2014]. 38, 40

[37] DATAPIPE. http://www.datapipe.com/, 2014. [Online; accessed 15-January-2014]. 37

[38] VMware Workstation. http://www.vmware.com/fr/products/workstation/, 2013. [Online; accessed 30-November-2013]. 26

[39] AppScale. http://aws.typepad.com/aws/2008/08/cloudbursting-.html, 2008. [Online; accessed 05-February-2014]. 49

[40] Grid5000. https://www.grid5000.fr/, 2014. [Online; accessed 20-March-2014]. 16, 98, 140, 141

[41] Tsuru. http://www.tsuru.io, 2014. [Online; accessed 20-January-2014]. 43, 44

[42] Paasmaker. http://paasmaker.org/, 2014. [Online; accessed 20-January-2014]. 42, 44

[43] VMware NSX. http://www.vmware.com/products/nsx/, 2013. [Online; accessed 30-November-2013]. 26

[44] Ubuntu Juju. https://juju.ubuntu.com/, 2014. [Online; accessed 11-February-2014]. 43

[45] Docker. http://www.docker.io/, 2014. [Online; accessed 12-February-2014]. 38, 43

[46] NumPy library. http://docs.scipy.org/doc/numpy/reference/index.html, 2014. [Online; accessed 11-April-2014]. 109

[47] CISCO Easy Virtual Network (EVN). http://www.cisco.com/en/US/products/ps11783/products_ios_protocol_option_home.html, 2013. [Online; accessed 30-November-2013]. 26

[48] Hive performance benchmarks. https://issues.apache.org/jira/browse/HIVE-396, 2014. [Online; accessed 01-April-2014]. 97

[49] Google Compute Engine. https://cloud.google.com/products/compute-engine, 2013. [Online; accessed 30-November-2013]. 26, 35, 37

[50] Activiti. http://activiti.org/index.html, 2014. [Online; accessed 30-April-2014]. 120

[51] Apache ODE. http://ode.apache.org/, 2014. [Online; accessed 30-April-2014]. 120

[52] Xerox. http://www.xerox.com/, 2014. [Online; accessed 30-April-2014]. 120

[53] Google App Engine Billing. https://developers.google.com/appengine/kb/billing#time_granularity_instance_pricing, 2014. [Online; accessed 06-May-2014]. 97

[54] Amazon AWS EC2. http://aws.amazon.com/ec2/, 2013. [Online; accessed 30-November-2013]. 26, 37

[55] Abdelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Andréea Chis, Yves Caniou, Eddy Caron, Pushpinder-Kaur Chouhan, Gaël Le Mahec, Holly Dail, Benjamin Depardon, et al. Diet: New developments and recent results. In *Euro-Par 2006: Parallel Processing*, pages 150–170. Springer, 2007. 19

[56] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web services agreement specification (ws-agreement). In *Global Grid Forum*, volume 2, 2004. 23

[57] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. 24, 29

[58] Adnan Ashraf, Benjamin Byholm, and Ivan Porres. Cramp: Cost-efficient resource allocation for multiple web applications with proactive scaling. *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 0:581–586, 2012. 52, 54, 57, 58

[59] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991. 21

[60] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4):361–372, 1998. 19

[61] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. 26

[62] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association. 19

[63] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, Mei-Hui Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10, 2008. 21

[64] Tekin Bicer, David Chiu, and Gagan Agrawal. Time and cost sensitive data-intensive computing on hybrid clouds. *Cluster Computing and the Grid, IEEE International Symposium on*, pages 636–643, 2012. 14, 52, 54, 57, 138

[65] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, 20(4):481–494, November 2006. 19

[66] M. Boniface, B. Nasser, J. Papay, S.C. Phillips, A. Servin, Xiaoyu Yang, Z. Zlatev, S.V. Gogouvitis, G. Katsaros, K. Konstanteli, G. Kousiouris, A. Menychtas, and D. Kyriazis. Platform-as-a-service architecture for real-time quality of service management in clouds. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 155–160, 2010. 45, 48, 58

[67] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing sla management based upon business objectives. *IBM Syst. J.*, 43(1):159–178, January 2004. 22

[68] Junwei Cao, Kai Hwang, Keqin Li, and Albert Y. Zomaya. Optimal multiserver configuration for profit maximization in cloud computing. *IEEE Trans. Parallel Distrib. Syst.*, 24(6):1087–1096, June 2013. 54, 55, 57, 58

[69] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 776–783, Washington, DC, USA, 2005. IEEE Computer Society. 18

[70] Jorge Carapinha and Javier Jiménez. Network virtualization: A view from the bottom. In *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, VISA '09, pages 73–80, New York, NY, USA, 2009. ACM. 26

[71] Jorge Cardoso, Amit Sheth, and John Miller. Workflow quality of service. In *Enterprise Inter-and Intra-Organizational Integration*, pages 303–311. Springer, 2003. 120

[72] Alexandra Carpen-Amarie, Djawida Dib, Anne-Cécile Orgerie, and Guillaume Pierre. Towards Energy-Aware IaaS-PaaS Co-design. In *SMARTGREENS: International Conference on Smart Grids and Green IT Systems, colocated with International*

Conference on Cloud Computing and Services Science (CLOSER),, Barcelona, Spain, 2014. 135

[73] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Gener. Comput. Syst.*, 2006. 74, 120

[74] N. M. Mosharaf Kabir Chowdhury and Raouf Boutaba. Network virtualization: State of the art and research challenges. *Comm. Mag.*, 47(7):20–26, July 2009. 26

[75] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A B Silva, C.O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: the mygrid approach. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 407–416, 2003. 21

[76] Stefania Victoria Costache, Nikos Parlavantzas, Christine Morin, and Samuel Kortas. Merkat: A Market-based SLO-driven Cloud Platform. In *5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013)*, Bristol, Royaume-Uni, December 2013. 46, 48, 58

[77] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. 21

[78] Ewa Deelman, Gurmeet Singh, Miron Livny, G. Bruce Berriman, and John Good. The cost of doing science on the cloud: the montage example. In *SC*, page 50. IEEE/ACM, 2008. 29

[79] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Generation Computer Systems*, 29(4):973 – 985, 2013. Special Section: Utility and Cloud Computing. 14, 51, 53, 58, 138

[80] Djawida Dib. Adaptation dynamique des fonctionnalités d'un système d'exploitation large échelle, May 2011. Publication de la journée ADAPT 2011. 135

[81] Djawida Dib, Nikos Parlavantzas, and Christine Morin. Towards multi-level adaptation for distributed operating systems and applications. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part II*, ICA3PP'12, pages 100–109, Berlin, Heidelberg, 2012. Springer-Verlag. 135

[82] Djawida Dib, Nikos Parlavantzas, and Christine Morin. Meryn: Open, sla-driven, cloud bursting paas. In *Proceedings of the First ACM Workshop on Optimization Techniques for Resources Management in Clouds*, ORMaCloud '13, pages 1–8, New York, NY, USA, 2013. ACM. 135

[83] Djawida Dib, Nikos Parlavantzas, and Christine Morin. SLA-based Profit Optimization in Cloud Bursting PaaS. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, États-Unis, 2014. 135

[84] Dmitry Duplyakin, Paul Marshall, Kate Keahey, Henry Tufo, and Ali Alzabarah. Rebalancing in a multi-cloud environment. In *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*, Science Cloud '13, pages 21–28, New York, NY, USA, 2013. ACM. 49, 53

[85] E. Feller, C. Rohr, D. Margery, and C. Morin. Energy management in iaas clouds: A holistic approach. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 204–212, June 2012. 120

[86] Eugen Feller, Louis Rilling, and Christine Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 482–489, Washington, DC, USA, 2012. IEEE Computer Society. 26, 88

[87] A.L. Freitas, N. Parlavantzas, and J-L Pazat. Cost reduction through sla-driven self-management. In *Web Services (ECOWS), 2011 Ninth IEEE European Conference on*, pages 117–124, Sept 2011. 54, 55, 57, 58

[88] Arijit Ganguly, Abhishek Agrawal, P. Oscar Boykin, and Renato Figueiredo. Ip over p2p: Enabling self-configuring virtual ip networks for grid computing. In *In Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS-2006*, pages 1–10, 2006. 26

[89] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001. 18

[90] I. Goiri, J. Guitart, and J. Torres. Characterizing cloud federation for enhancing providers' profit. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 123–130, July 2010. 14, 50, 51, 53, 58, 138

[91] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(9):34–45, September 1974. 26

[92] Hadi Goudarzi, Mohammad Ghasemazar, and Massoud Pedram. Sla-based optimization of power and migration cost in cloud computing. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid*, 2012. 55, 57, 120

[93] Hadi Goudarzi and Massoud Pedram. Multi-dimensional sla-based resource allocation for multi-tier cloud computing systems. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11. IEEE Computer Society, 2011. 52, 57, 58

[94] Andrew S. Grimshaw, Wm. A. Wulf, and CORPORATE The Legion Team. The legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1):39–45, January 1997. 19

[95] NIST Cloud Computing Standards Roadmap Working Group, Michael Hogan, Fang Liu, Annie Sokol, and Jin Tong. NIST Cloud Computing Standards

Roadmap. Technical report, National Institute of Standards and Technology, July 2011. 24, 25

[96] L Guo, AS McGough, A Akram, D Colling, J Martyniak, and M Krznaric. Qos for service based workflow on grid. In *Proceedings of UK e-Science 2007 All Hands Meeting, Nottingham, UK*, 2007. 120

[97] Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. Seagull: Intelligent cloud bursting for enterprise applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 33–33, Berkeley, CA, USA, 2012. USENIX Association. 50, 51, 53

[98] Peer Hasselmeyer, Henning Mersch, Bastian Koller, HN Quyen, Lutz Schubert, and Philipp Wieder. Implementing an sla negotiation framework. In *Proceedings of the eChallenges Conference (e-2007)*, volume 4, pages 154–161, 2007. 24

[99] M. Reza HoseinyFarahabady, Young Choon Lee, and Albert Y. Zomaya. Pareto-optimal cloud bursting. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2013. 51, 53, 58

[100] Jan Hungershofer. On the combined scheduling of malleable and rigid jobs. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '04, pages 206–213, Washington, DC, USA, 2004. IEEE Computer Society. 21

[101] Anca Iordache, Christine Morin, Nikos Parlavantzas, Eugen Feller, and Pierre Riteau. Resilin: Elastic mapreduce over multiple clouds. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:261–268, 2013. 46, 48, 58

[102] Xuxian Jiang and Dongyan Xu. Violin: Virtual internetworking on overlay infrastructure. In Jiannong Cao, LaurenceT. Yang, Minyi Guo, and Francis Lau, editors, *Parallel and Distributed Processing and Applications*, volume 3358 of *Lecture Notes in Computer Science*, pages 937–946. Springer Berlin Heidelberg, 2005. 26

[103] Gueyoung Jung, N. Gnanasambandam, and T. Mukherjee. Synchronous parallel processing of big-data analytics services to optimize performance in federated clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 811–818, June 2012. 49, 50, 53

[104] Steffen Kächele, Jörg Domaschka, and Franz J. Hauck. Cosca: An easy-to-use component-based paas cloud system for common applications. In *Proceedings of the First International Workshop on Cloud Computing Platforms*, CloudCP '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM. 45, 48

[105] S. Kailasam, N. Gnanasambandam, J. Dharanipragada, and N. Sharma. Optimizing ordered throughput using autonomic cloud bursting schedulers. *Software Engineering, IEEE Transactions on*, 39(11):1564–1581, Nov 2013. 49, 53, 58

[106] Mira Kajko-Mattsson. Sla management process model. In *Proceedings of the 2Nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 240–249, New York, NY, USA, 2009. ACM. 22

[107] Mohammad Mahdi Kashef and Jörn Altmann. A cost model for hybrid clouds. In *Proceedings of the 8th international conference on Economics of Grids, Clouds, Systems, and Services*, GECON'11, pages 46–60, Berlin, Heidelberg, 2012. Springer-Verlag. 29

[108] Kate Keahey, Tim Freeman, Jerome Lauret, and Doug Olson. Virtual workspaces for scientific applications. *Journal of Physics: Conference Series*, 78(1):012038, 2007. 26

[109] S. Kibe, S. Watanabe, K. Kunishima, R. Adachi, M. Yamagiwa, and M. Uehara. Paas on iaas. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 362–367, 2013. 45, 48

[110] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007. 26

[111] André Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. An Integrated Approach for Specifying and Enforcing SLAs for Cloud Services. In *The IEEE 5th International Conference on Cloud Computing (CLOUD 2012)*, Honolulu, États-Unis, June 2012. 47, 48

[112] E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall, M. Craig, A. Di Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, and A. Wilson. Programming the grid with glite. In *Computational Methods in Science and Technology*, page 2006, 2006. 19

[113] Sonja Lehmann and Peter Buxmann. Pricing strategies of software vendors. *Business & Information Systems Engineering*, 1(6):452–462, 2009. 23

[114] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar. Cost-efficient and application sla-aware client side request scheduling in an infrastructure-as-a-service cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 213–220, June 2012. 52, 57, 58

[115] Philipp Leitner, Zabolotnyi Rostyslav, Alessio Gambi, and Schahram Dustdar. A framework and middleware for application-level cloud bursting on top of infrastructure-as-a-service clouds. *6th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2013)*, 2013. 49, 50, 53

[116] Siew Huei Liew and Ya-Yunn Su. Cloudguide: Helping users estimate cloud deployment cost and performance for legacy web applications. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, CLOUDCOM '12, pages 90–98, Washington, DC, USA, 2012. IEEE Computer Society. 30

[117] Shuo Liu, Shaolei Ren, Gang Quan, Ming Zhao, and Shangping Ren. Profit aware load balancing for distributed cloud data centers. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 611–622, May 2013. 55, 57

[118] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105 – 1122, 2003. 97

[119] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P King, and Richard Franck. Web service level agreement (wsla) language specification. *IBM Corporation*, pages 815–824, 2003. 23

[120] Maciej Malawski, Kamil Figiela, and Jarek Nabrzyski. Cost minimization for computational applications on hybrid cloud infrastructures. *Future Generation Computer Systems*, 29(7):1786 – 1794, 2013. 14, 52, 54, 57, 58, 138

[121] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 43–52, Washington, DC, USA, 2010. IEEE Computer Society. 51, 52, 53

[122] Michele Mazzucco and Dmytro Dyachuk. Optimizing cloud providers revenues via energy efficient server allocation. *Sustainable Computing: Informatics and Systems*, 2(1):1 – 12, 2012. 55, 57, 120

[123] Dejan Miloji?i?, I.M. Llorente, and Ruben S. Montero. Opennebula: A cloud management tool. *Internet Computing, IEEE*, 15(2):11–14, 2011. 26, 88

[124] Christine Morin. Xtreemos: a grid operating system making your computer ready for participating in virtual organizations. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, pages 393–402. IEEE, 2007. 19

[125] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis Rilling. Kerrighed: a single system image cluster operating system for high performance computing. In *Euro-Par 2003 Parallel Processing*, pages 1291–1294. Springer, 2003. 19

[126] C. Müller, O. Martín-Díaz, A. Ruiz-Cortés, M. Resinas, and P. Fernández. Improving temporal-awareness of ws-agreement. In BerndJ. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 193–206. Springer Berlin Heidelberg, 2007. 23

[127] Alek Opitz, Hartmut König, and Sebastian Szamlewska. What does grid computing cost? *Journal of Grid Computing*, 6(4):385–397, 2008. 29

[128] Pradeep Padala and JosephN. Wilson. Gridos: Operating system services for grid architectures. In TimothyMark Pinkston and ViktorK. Prasanna, editors, *High Performance Computing - HiPC 2003*, volume 2913 of *Lecture Notes in Computer Science*, pages 353–362. Springer Berlin Heidelberg, 2003. 19

[129] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. Managing elasticity across multiple cloud providers. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, MultiCloud '13, pages 53–60, New York, NY, USA, 2013. ACM. 45, 48

[130] G. Pierre and C. Stratan. Conpaas: A platform for hosting elastic cloud applications. *Internet Computing, IEEE*, 16(5):88–92, 2012. 27, 41, 44, 58

[131] Omer Rana, Martijn Warnier, Thomas B. Quillinan, and Frances Brazier. Monitoring and reputation mechanisms for service level agreements. In *Proceedings of the 5th International Workshop on Grid Economics and Business Models*, GECON '08, pages 125–139, Berlin, Heidelberg, 2008. Springer-Verlag. 23

[132] James E Reuter, David W Thiel, Richard F Wrenn, and Andrew C St Martin. System and method for managing virtual storage, June 1 2004. US Patent 6,745,207. 26

[133] Nikzad Babaii Rizvandi, Albert Y. Zomaya, Ali Javadzadeh Boloori, and Javid Taheri. On modeling dependency between mapreduce configuration parameters and total execution time. *CoRR*, 2012. 74, 120

[134] Rizos Sakellariou and Viktor Yarmolenko. On the flexibility of ws-agreement for job submission. In *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, MGC '05, pages 1–6, New York, NY, USA, 2005. ACM. 23

[135] Lutz Schubert, Keith G Jeffery, and Burkard Neidecker-Lutz. *The Future of Cloud Computing: Opportunities for European Cloud Computing Beyond 2010:–expert Group Report*. European Commission, Information Society and Media, 2010. 24, 25

[136] B. Sharma, R.K. Thulasiram, P. Thulasiraman, S.K. Garg, and R. Buyya. Pricing cloud compute commodities: A novel financial economic model. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 451–457, 2012. 30

[137] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005. 26

[138] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. 18

[139] Sen Su, Jian Li, Qingjia Huang, Xiao Huang, Kai Shuang, and Jie Wang. Cost-efficient task scheduling for executing large programs in the cloud. *Parallel Computing*, 39(4-5):177–188, 2013. 54, 55, 57

[140] Byung Chul Tak, Bhuvan Urgaonkar, and Anand Sivasubramaniam. To move or not to move: The economics of cloud computing. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association. 29

[141] Adel Nadjaran Toosi, Rodrigo N. Calheiros, Ruppa K. Thulasiram, and Rajkumar Buyya. Resource provisioning policies to increase iaas provider's profit in a federated cloud environment. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, HPCC '11, pages 279–287, Washington, DC, USA, 2011. IEEE Computer Society. 14, 50, 51, 53, 58, 120, 138

[142] Johan Tordsson, Rubén S. Montero, Rafael Moreno-Vozmediano, and Ignacio M. Llorente. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future Generation Computer Systems*, 28(2):358 – 367, 2012. 50, 51, 53

[143] Hong-Linh Truong and Schahram Dustdar. Composable cost estimation and monitoring for computational applications in cloud computing environments. *Procedia Computer Science*, 1(1):2175 – 2184, 2010. <ce:title>ICCS 2010</ce:title>. 29

[144] Radu Tudoran, Alexandru Costan, Gabriel Antoniu, and Luc Bougé. A performance evaluation of azure and nimbus clouds for scientific applications. In *Proceedings of the 2Nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pages 4:1–4:6, New York, NY, USA, 2012. ACM. 97

[145] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008. 24

[146] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: a software platform for .net-based cloud computing. *High Speed and Large Scale Scientific Computing*, pages 267–295, 2009. 47, 48, 58

[147] Bruce J Walker. Open single system image (openssi) linux cluster project. *2005*, 2008. 19

[148] Cheng Wang, Bhuvan Urgaonkar, Qian Wang, George Kesidis, and Anand Sivasubramaniam. Data center cost optimization via workload modulation under real-world electricity pricing. *arXiv preprint arXiv:1308.0585*, 2013. 55

[149] Jon Watson. Virtualbox: Bits and bytes masquerading as machines. *Linux J.*, 2008(166), February 2008. 26

[150] Barry B White. Virtual storage system and method, August 21 1984. US Patent 4,467,421. 26

[151] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012. 18, 36, 39

[152] Linlin Wu and Rajkumar Buyya. Service level agreement (sla) in utility computing systems. *CoRR*, abs/1010.2881, 2010. 22, 24

[153] Linlin Wu, S.K. Garg, and R. Buyya. Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, 2011. 54, 55, 57

[154] S. Yangui and S. Tata. Cloudserv: Paas resources provisioning for service-based applications. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 522–529, 2013. 45, 48

[155] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM. 97

[156] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. 25

[157] Jian Zhao, Hongxing Li, Chuan Wu, Zongpeng Li, Zhizhong Zhang, and F Lau. Dynamic pricing and profit maximization for clouds with geo-distributed datacenters. *Proc. of IEEE INFOCOM to appear*, 2014. 54, 57, 58

[158] Jian Zhao, Chuan Wu, and Zongpeng Li. Cost minimization in multiple iaas clouds: A double auction approach. *CoRR*, abs/1308.0841, 2013. 14, 54, 57, 138

Appendix A

Publications

- **International Conferences**
  - Djawida Dib, Nikos Parlavantzas, and Christine Morin. SLA-based Profit Optimization in Cloud Bursting PaaS. *In Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014)*, May 2014. [83]
  - Alexandra Carpen-Amarie, Djawida Dib, Anne-Cécile Orgerie, and Guillaume Pierre. Towards Energy-Aware IaaS-PaaS Co-design. *In SMARTGREENS: International Conference on Smart Grids and Green IT Systems, colocated with International Conference on Cloud Computing and Services Science (CLOSER)*, April 2014. [72]
  - Djawida Dib, Nikos Parlavantzas, and Christine Morin. Towards Multi-Level Adaptation for Distributed Operating Systems and Applications. *In Proceedings of the 12th international conference on Algorithms and Architectures for Parallel Processing - Volume Part II (ICA3PP'12), Yang Xiang, Ivan Stojmenovic, Bernady O. Apduhan, Guojun Wang, and Koji Nakano (Eds.), Vol. Part II. Springer-Verlag, Berlin, Heidelberg, 100-109.*, September 2012. [81]
- **International Workshops**
  - Djawida Dib, Nikos Parlavantzas, and Christine Morin. Meryn: Open, SLA-driven, Cloud Bursting PaaS. *In Proceedings of the first ACM workshop on Optimization techniques for resources management in clouds (ORMaCloud '13). ACM, New York, NY, USA, 1-8.*, June 2013. [82]
- **Other**
  - Djawida Dib. Adaptation dynamique des fonctionnalités d'un système d'exploitation large échelle. *In Journée Adapt, Saint-Malo, France* May 2011. [80]

Annexe B

Résumé en Français

B.1 Introduction

L'apparition de l'informatique dans les nuages (*cloud computing*) ces dernières années a ouvert de nouvelles perspectives à l'utilisation et la commercialisation des ressources informatiques. Le concept de cloud computing permet aux utilisateurs du monde entier d'accéder à de très grandes capacités de calcul en utilisant une simple connexion Internet et en ne payant que pour les ressources réellement utilisées. Ce modèle de tarification à l'usage (*pay-as-you-go*) attire beaucoup de consommateurs et de petites entreprises ayant l'objectif de réduire le coût d'utilisation des ressources informatiques, et offre aux fournisseurs de nouvelles opportunités pour commercialiser les ressources informatiques. Les services de cloud computing sont fournis selon trois modèles fondamentaux : infrastructure en tant que service *(IaaS)*, plate-forme en tant que service *(PaaS)*, et logiciel en tant que service *(SaaS)*. Le modèle de service IaaS fournit des ressources de base, tels que des processeurs, de l'espace de stockage et du réseau. Le modèle de service PaaS fournit un environnement de développement et d'exécution prêt à être exploité par les applications. Le modèle de service SaaS fournit des applications hébergées prêtes à être utilisées.

Dans cette thèse nous nous intéressons au modèle de service PaaS fondé sur des ressources du niveau IaaS. Les systèmes PaaS dispensent leurs utilisateurs d'acquérir des compétences techniques de programmation et d'administration pour déployer leurs applications et de gérer les ressources sous-jacentes. L'interaction entre les clients et les fournisseurs de PaaS est gouvernée par des contrats de service SLA (*Service Level Agreement*), spécifiant les obligations de chaque partie ainsi que les paiements et les pénalités associés. L'impact socio-économique des services de PaaS devient essentiel puisque le nombre d'utilisateurs et de fournisseurs des clouds PaaS est en pleine croissance.

L'objectif principal des fournisseurs de cloud PaaS est de générer le maximum de profit des services qu'ils fournissent. Cela les oblige à faire face à un certain nombre de défis. Ils doivent gérer efficacement le placement et l'isolation des applications des clients sur les ressources et satisfaire les SLAs des applications afin d'éviter le paiement de pénalités. Les grands fournisseurs de clouds PaaS commerciaux, tels que Amazon [10][11] et Microsoft [1], utilisent leurs propres ressources. Cependant, les fournisseurs émergeants, tels que Heroku [15], CloudBees [34] et dotCoud [35], louent à la demande des ressources de clouds IaaS publics. Dans cette thèse nous considérons un environnement PaaS hybride de *cloud bursting*, où le système PaaS est fondé sur un

nombre limité de ressources privées et est capable de s'étendre sur des ressources de clouds IaaS publics. Un tel environnement représente le cas général et offre plusieurs options au fournisseur de cloud PaaS pour sélectionner les ressources qui hébergent les applications.

Récemment, plusieurs travaux de recherche se sont intéressés aux optimisations économiques dans des environnements composés de plusieurs systèmes de cloud. Cependant, la plupart de ces travaux visent des types d'application spécifiques [120][79][64] ou des types d'environnement spécifiques [90][141][158]. C'est pourquoi l'optimisation du profit d'un fournisseur de cloud PaaS exploitant la fonctionnalité de *cloud bursting* reste un sujet de recherche ouvert. Les travaux de cette thèse s'inscrivent dans cette thématique.

B.2 Objectifs

L'objectif principal de cette thèse est de proposer *une solution d'optimisation du profit des fournisseurs PaaS permettant le support de SLA et du cloud bursting*. Afin d'atteindre cet objectif, cette thèse étudie quatre sous-objectifs.

- **Optimisation du profit.** L'optimisation du profit est l'objectif principal de chaque fournisseur de cloud PaaS. D'un côté, ceci implique l'acceptation et la satisfaction des requêtes des clients que ce soit dans des périodes de forte ou de faible demande. D'un autre côté, la compétition entre les fournisseurs contraint chaque fournisseur à proposer des prix raisonnables et compétitifs par rapport aux prix pratiqués sur le marché, ce qui limite leurs revenus. Par conséquent l'optimisation du profit du fournisseur doit passer par l'optimisation des coûts induits pour l'hébergement des applications. Cependant, si l'optimisation de tels coûts conduit le fournisseur à ne pas tenir ses engagements concernant la qualité de service (QoS) promise aux applications, les utilisateurs seront déçus et peuvent se tourner vers d'autres fournisseurs. Ainsi notre objectif ici est de *fournir une politique qui cherche les ressources les moins chéres pour le fournisseur pour héberger les applications des clients, tout en prenant en compte la réputation du fournisseur et le profit sur le long terme.*

- **Support de SLA.** Les consommateurs de clouds PaaS s'attendent à ce que les fournisseurs proposent des contrats de SLA qui garantissent un certain niveau de QoS à leurs applications. Une partie importante d'un accord de SLA consiste à compenser les utilisateurs si le niveau de QoS promis à leurs applications n'est pas tenu. De telles compensations peuvent pénaliser le profit des fournisseurs mais sont nécessaires pour gagner la confiance des clients. Ainsi notre but est de *fournir aux applications des contrats de SLA fondés sur la QoS et de prendre en compte le paiement de pénalités si le fournisseur n'arrive pas a fournir aux applications de ses clients le niveau de QoS promis.*

- **Support de plusieurs types d'applications.** Une solution de cloud PaaS facilement extensible pour supporter de nouveaux types d'applications est attrayante pour les consommateurs et les fournisseurs. D'un côté, elle offre plus de flexibilité aux consommateurs ayant des besoins variés. D'un autre côté, elle permet aux fournisseurs d'attirer des clients de différents domaines professionnels et d'ajouter facilement le support aux nouvelles applications rentables, ce qui les aide à

augmenter leur chiffre d'affaire. Pour cela, notre objectif est de *fournir une solution d'optimisation de profit de fournisseur PaaS générique qui soit indépendante d'un type d'application spécifique.*

- **Système PaaS avec cloud bursting** Un système PaaS avec *cloud bursting* permettant le déploiement des applications simultanément sur des ressources privées et publiques offre au fournisseur de cloud PaaS plusieurs options de déploiement pour optimiser le coût ou la performance des applications. Le support du *cloud bursting* nécessite un travail d'ingénierie important pour permettre au système de PaaS d'utiliser plusieurs systèmes IaaS et de garantir les termes de SLA des applications déployées simultanément sur plusieurs clouds IaaS. À cet effet, notre but est de *concevoir un système PaaS avec cloud bursting permettant le déploiement des applications sur plusieurs clouds.*

B.3 Contributions

L'objectif de cette thèse est de fournir une solution d'optimisation du profit du fournisseur PaaS sous des contraintes de SLA dans un environnement de *cloud bursting*. Cette thèse aborde cet objectif avec les contributions suivantes.

B.3.1 Modèle d'optimisation de profit

Nous définissons un modèle générique de système PaaS avec la fonctionnalité de *cloud bursting* et nous proposons une politique d'optimisation du profit du fournisseur de cloud PaaS. La politique proposée essaye de trouver les ressources les moins chères pour héberger les applications des clients, tout en prenant en compte la satisfaction de leurs SLAs. Concrètement, à chaque requête d'un client la politique évalue le coût d'héberger son application en utilisant les ressources publiques et privées et choisit l'option qui génère le plus de profit. Pendant les périodes de pointe et l'indisponibilité des ressources privées, la politique essaye d'optimiser l'utilisation des ressources privées en considèrant deux autres options. La première option consiste à emprunter quelques ressources aux applications en cours d'exécution tout en considérant le paiement de pénalités si leur niveau de QoS promis est affecté. La seconde option consiste à évaluer la pénalité possible de la nouvelle application due à l'attente de la libération de ressources privées. La politique proposée est générique et peut être appliquée sur n'importe quel type d'application avec une seule condition qui consiste à faire coopérer des entités spécifiques aux types d'application supportés. De telles entités fournissent des informations sur le modèle de performance et le contrat de SLA des applications hébergées.

B.3.2 Application du modèle d'optimisation

Pour montrer l'applicabilité de notre modèle d'optimisation de profit sur une classe particulière d'application, nous avons étudié les applications de calcul rigides et élastiques Nous avons défini des termes de SLA correspondants et présenté deux heuristiques qui collaborent avec la politique d'optimisation pour maximiser le profit du fournisseur de cloud PaaS. La première heuristique fournit une *offre d'attente* qui représente la plus petite pénalité possible pour la nouvelle application due à l'attente de la disponibilité de

ressources privées. La deuxième heuristique fournit une *offre de don* qui représente la plus petite pénalité possible pour des applications en cours d'exécution induite par la fourniture d'une partie de leurs ressources à la nouvelle application. Nous avons défini trois formes possibles qui peuvent être utilisées par les applications de calcul pour fournir des ressources durant leur exécution. La première forme consiste à suspendre des applications durant l'exécution de la nouvelle application et les reprendre une fois que la nouvelle application termine son exécution et rend les ressources. La deuxième forme consiste à prêter une partie des ressources des applications en cours d'exécution à la nouvelle application et de continuer leur exécution avec les ressources restantes jusqu'à ce que la nouvelle application termine son exécution et rende les ressources empruntées. La troisième forme consiste à donner une partie des ressources des applications en cours d'exécution à la nouvelle application et à continuer leur exécution avec les ressources restantes. La deuxième et troisième forme ne peuvent être utilisées qu'avec les applications de calcul élastiques. Chaque forme induit pour l'application une pénalité différente. Ainsi, l'heuristique sélectionne la forme d'impact qui conduit à la plus petite pénalité possible.

B.3.3 Meryn : un système de PaaS avec la fonctionnalité de cloud bursting

Nous avons conçu et implémenté une architecture de système PaaS, appelée Meryn, capable d'incorporer la politique d'optimisation de profit ainsi que les différentes heuristiques spécifiques aux types d'application. L'architecture du système Meryn repose sur des ressources privées virtualisées, en utilisant une solution de gestion de cloud IaaS, et supporte la fonctionnalité de *cloud bursting* dans des cloues publics quand cela est nécessaire. Pour supporter l'extensibilité par rapport aux types d'applications, Meryn fait usage de clusters virtuels et gère chaque cluster virtuel en utilisant un framework existant dédié à un type d'application spécifique. En l'occurrence, le framework OGE est dédié aux applications batch et le framework Hadoop est dédié aux applications MapReduce. Les ressources privées sont dynamiquement partagées entre les clusters virtuels suivant des politiques et des objectifs spécifiques, à savoir dans notre contexte suivant la politique d'optimisation de profit.

B.4 Évaluation

Pour valider nos contributions nous avons implémenté un prototype Meryn supportant les applications batch et MapReduce. Nous avons évalué la politique d'optimisation de profit proposée en utilisant le prototype Meryn et des modèles de workload fondés sur des traces du monde réel. Nous avons effectué un ensemble de simulations et d'expériences sur la plate-forme d'expérimentation Grid'5000 [40].

Nous avons comparé la politique d'optimisation proposée à une approche de base qui attribue un nombre fixe de ressources privées aux clusters virtuels et elle ne leur permet de s'étendre qu'avec des ressources de clouds publics. Les résultats montrent que notre politique d'optimisation permet au fournisseur de cloud PaaS d'augmenter son profit de 11.59% et 9.02% dans respectivement les simulations et les expériences par rapport à l'approche de base. Comme coût d'une telle optimisation, le temps de terminaison des workloads dans un système utilisant notre politique d'optimisation est

légèrement supérieure par rapport à l'approche de base. En outre, le niveau de QoS promis à certaines applications a été affecté.

B.5 Organisation du manuscrit

Ce manuscrit est organisé comme suit. Le chapitre 2 présente les définitions générales du contexte de notre travail. Plus précisément, ce chapitre fournit un aperçu des différentes infrastructures de calcul et des différents modèles d'applications. En outre, il définit les concepts de *Service Level Agreement* (SLA) et de *cloud computing*. Le chapitre 3 présente les différentes conditions requises pour atteindre les objectifs de cette thèse, il couvre l'état de l'art et il positionne les contributions de cette thèse. Les principaux axes couvèrt sont : les politiques d'optimisation économique, les mécanismes de *cloud bursting*, et les systèmes de type PaaS. Le chapitre 4 décrit la première contribution de cette thèse qui consiste en la proposition d'un modèle générique pour l'optimisation du profit d'un fournisseur de cloud OaaS dans un environnement de cloud bursting. Le chapitre 5 étudie les applications de calcul rigides et élastiques dans le but d'appliquer concrètement notre modèle générique d'optimisation de profit. Le chapitre 6 présente les principes de conception, l'architecture et la mise en œuvre du système Meryn, un cloud PaaS qui implémente le scénario de cloud bursting et les algorithmes d'optimisation que nous avons proposés. Le chapitre 7 montre les résultats d'évaluation de nos contributions à l'aide de simulations et expériences effectuées sur la plate-forme d'expérimentation Grid'5000 [40] avec notre prototype. Le chapitre 8 conclut ce manuscrit en résumant nos contributions et en présentant quelque perspectives.

Optimizing PaaS Provider Profit under Service Level Agreement Constraints

Abstract

Cloud computing is an emerging paradigm revolutionizing the use and marketing of information technology. As the number of cloud users and providers grows, the socio-economical impact of cloud solutions and particularly PaaS (platform as a service) solutions is becoming increasingly critical. The main objective of PaaS providers is to generate the maximum profit from the services they provide. This requires them to face a number of challenges such as efficiently managing the underlying resources and satisfying the SLAs of the hosted applications.

This thesis considers a cloud-bursting PaaS environment where the PaaS provider owns a limited number of private resources and is able to rent public cloud resources, when needed. This environment enables the PaaS provider to have full control over services hosted on the private cloud and to take advantage of public clouds for managing peak periods. In this context, we propose a profit-efficient solution for managing the cloud-bursting PaaS system under SLA constraints. We define a profit optimization policy that, after each client request, evaluates the cost of hosting the application using public and private resources and chooses the option that generates the highest profit. During peak periods the optimization policy considers two more options. The first option is to take some resources from running applications, taking into account the payment of penalties if their promised quality of service is affected. The second option is to wait until private resources become available, taking into account the payment of penalties if the promised quality of service of the new application is affected. Furthermore we designed and implemented an open cloud-bursting PaaS system, called *Meryn*, which integrates the proposed optimization policy and provides support for batch and MapReduce applications. The results of our evaluation show the effectiveness of our approach in optimizing the provider profit. Indeed, compared to a basic approach, our approach provides up to 11.59% and 9.02% more provider profit in, respectively, simulations and experiments.

Keywords

Cloud computing, Platform as a Service (PaaS), Service Level Agreement (SLA), economic optimization, computational applications.

Résumé

L'informatique en nuage (cloud computing) est un paradigme émergent qui révolutionne l'utilisation et la commercialisation des services informatiques. De nos jours, l'impact socio-économique de l'informatique en nuage et plus particulièrement des services de PaaS (plate-forme en tant que service) devient essentiel, puisque le nombre d'utilisateurs et de fournisseurs des cloud PaaS est en pleine croissance. L'objectif principal des fournisseurs de cloud PaaS est de générer le maximum de profit des services qu'ils fournissent. Cela les oblige à faire face à un certain nombre de défis, tels que la gestion efficace des ressources sous-jacentes et la satisfaction des SLAs (contrat de service) des applications hébergées.

Dans cette thèse, nous considérons un environnement PaaS hybride de *cloud bursting*, où le fournisseur PaaS possède un nombre limité de ressources privées et a la possibilité de louer des ressources publiques. Ce choix permet au fournisseur PaaS d'avoir un contrôle complet sur les services hébergés dans les ressources privées et de profiter de ressources publiques pour gérer les périodes de pointe. De plus, nous proposons une solution rentable pour gérer un tel système PaaS sous des contraintes de SLA. Nous définissons une politique d'optimisation de profit qui, à chaque requête d'un nouveau client, évalue le coût d'hébergement de son application en utilisant les ressources publiques et privées et choisit l'option qui génère le plus de profit. Pendant les

périodes de pointe la politique considère deux autres options. La première option consiste à emprunter quelques ressources aux applications en cours d'exécution tout en considérant le paiement de pénalités si leur qualité de service est affectée. La seconde option consiste à attendre que des ressources privées soient libérés tout en considérant le paiement de pénalités si la qualité de service de la nouvelle application est affectée. En outre, nous avons conçu et mis en œuvre une architecture de cloud PaaS, appelée Meryn, qui intègre la politique d'optimisation proposée, supporte le cloud bursting et héberge des applications du type batch et MapReduce. Les résultats de notre évaluation montrent l'efficacité de notre approche dans l'optimisation du profit du fournisseur. En effet, comparée à une approche de base, notre approche fournit jusqu'à 11.59 % et 9.02 % plus de profits pour le fournisseur dans respectivement les simulations et les expériences.

Mots clés

Informatique en nuage, plate-forme en tant que service, contrat de service, optimisation économique, applications de calcul.