Cooperative resource management in the cloud

Giang Son Tran

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :

M. GIANG SON TRAN

le mercredi 4 juin 2014

Titre :

COOPERATIVE RESOURCE MANAGEMENT IN THE CLOUD

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. DANIEL HAGIMONT

Rapporteurs :

M. DIDIER DONSEZ, UNIVERSITE GRENOBLE 1

M. JEAN-MARC MENAUD, ECOLE DES MINES DE NANTES

Membre(s) du jury :

M. NOËL DE PALMA, UNIVERSITE GRENOBLE 1, Président

M. ALAIN TCHANA, INP TOULOUSE, Membre

M. DANIEL HAGIMONT, INP TOULOUSE, Membre

To my family...

No one's born a computer scientist,
but with a little hard work,
and some math and science,
just about anyone can become one.
Barrack Obama.

Firstly, I would like to express my gratitude to all the members of the jury who spent invaluable time to evaluate my work. I would like to thank especially the reviewers Mr. Jean-Marc Menaud and Mr. Didier Donsez who have given me the constructive critics. I also would like to thank Mr. Daniel Hagimont, Professor at Institute National Polytechnique de Toulouse for his supervision throughout my work. His advices were important and essential for my research during these years.

I was very happy to swork within the SEPIA research team at IRIT-ENSEEIHT. I deeply felt the working spirit of our team in each weekly meeting. Thanks Alain Tchana for a lot of your welcome and help in my personal life when I initially joined the group, and for our joint work as a background of my thesis. Thanks Larissa Mayap - my office mate - for our discussions and works during our thesis. I also would like to thank Laurent Broto, Suzy Temate, Aeiman Gadafi and Ekane Brice for all of your help.

I would like to thank the Vietnamese Government, the Institute of Information Technology (IOIT) and the University of Science and Technology of Hanoi (USTH) for the grant to work in France. Many thanks to the laboratory IRIT for funding all of my trips to the conferences.

Finally, I would like to express my deep gratitude to my parents who have raised me, covered me and educated me for such a long time. Thank you my little sister for all the fun we have had and shared together. Last but not least, I cannot finish my work without everyday help and encourgement from my family members, my wife and my little princess. They are my source of love and strength to continue my pursuit of science.

Recent advances in computer infrastructures encourage the separation of hardware and software management tasks. Following this direction, virtualized cloud infrastructures are becoming very popular. Among various cloud models, Infrastructure as a Service (IaaS) provides many advantages to both provider and customer. In this service model, the provider offers his virtualized resource, and is responsible for managing his infrastructure, while the customer manages his application deployed in the allocated virtual machines. These two actors typically use autonomic resource management systems to automate these tasks at runtime.

Minimizing the amount of resource (and power consumption) in use is one of the main services that such cloud model must ensure. This objective can be done at runtime either by the customer at the application level (by scaling the application) or by the provider at the virtualization level (by migrating virtual machines based on the infrastructure's utilization rate). In traditional cloud infrastructures, these resource management policies work uncoordinated: knowledge about the application is not shared with the provider. This behavior faces application performance overheads and resource wasting, which can be reduced with a cooperative resource management policy.

In this research work, we discuss the problem of separate resource management in the cloud. After having this analysis, we propose a direction to use elastic virtual machines with cooperative resource management. This policy combines the knowledge of the application and the infrastructure in order to reduce application performance overhead and power consumption. We evaluate the benefit of our cooperative resource management policy with a set of experiments in a private IaaS. The evaluation shows that our policy outperforms uncoordinated resource management in traditional IaaS with lower performance overhead, better virtualized and physical resource usage.

L'évolution des infrastructures informatiques encourage la gestion séparée de l'infrastructure matérielle et de celle des logiciels. Dans cette direction, les infrastructures de cloud virtualisées sont devenues très populaires. Parmi les différents modèles de cloud, les Infrastructures as a Service (IaaS) ont de nombreux avantages pour le fournisseur comme pour le client. Dans ce modèle de cloud, le fournisseurs fournit ses ressources virtualisées et il est responsable de la gestion de son infrastructure. De son coté, le client gère son application qui est déployée dans les machines virtuelles allouées. Ces deux acteurs s'appuient généralement sur des systèmes d'administration autonomes pour automatiser les tâches d'administration.

Réduire la quantité de ressources utilisée (et la consommation d'énergie) est un des principaux objectifs de ce modèle de cloud. Cette réduction peut être obtenue à l'exécution au niveau de l'application par le client (en redimensionant l'application) ou au niveau du système virtualisé par le fournisseur (en regroupant les machines virtuelles dans l'infrastructure matérielle en fonction de leur charge). Dans les infrastructures de cloud traditionnelles, les politiques de gestion de ressources ne sont pas coopératives: le fournisseur ne possède pas d'informations détaillées sur les applications. Ce manque de coordination engendre des surcoùts et des gaspillages de ressources qui peuvent être reduits avec une politique de gestion de ressources coopérative.

Dans cette thèse nous traitons du problème de la gestion de ressources séparée dans un environnement de cloud virtualisé. Nous proposons un modèle de machines virtuelles élastiques avec une politique de gestion coopérative des ressources. Cette politique associe la connaissance des deux acteurs du cloud afin de réduire les coûts et la consommation d'énergie. Nous évaluons les bénéfices de cette approche avec plusieurs expériences dans un IaaS privé. Cette évaluation montre que notre politique est meilleur que la gestion des ressources non coordonnée dans un IaaS traditionnel, car son impact sur les performances est faible et elle permet une meilleure utilisation des ressources matérielles et logicielles.

Contents

1 Introduction **1**

2 Cloud and Resource Management **4**
2.1 Cloud Computing . . . 5
2.1.1 Introduction and Definition . . . 5
2.1.2 Classifications . . . 7
2.1.3 Advantages and Disadvantages . . . 11
2.1.4 Synthesis . . . 14
2.2 Virtualization . . . 15
2.2.1 Definitions . . . 15
2.2.2 Classifications . . . 17
2.2.3 Advantages and Disadvantages . . . 20
2.2.4 Synthesis . . . 20
2.3 Resource Management in a IaaS . . . 22
2.3.1 Resource Management by the Provider . . . 23
2.3.2 Resource Management by the Customer . . . 24
2.4 Synthesis . . . 24

3 Motivation and Orientation **25**
3.1 Fixed Size Virtual Machine . . . 26
3.1.1 Resource Holes . . . 27
3.1.2 Performance overhead . . . 28
3.2 Elastic Virtual Machine . . . 31
3.3 Cooperative IaaS . . . 36
3.3.1 Approach overview . . . 36
3.3.2 Characteristics . . . 39
3.3.3 Comparison with a conventional IaaS or a PaaS . . . 40
3.4 Synthesis . . . 43

4 State of the Art **44**
4.1 Resource Management with Static Virtual Machines . . . 45
4.1.1 Traditional Resource Management . . . 45
4.1.1.1 The Customer Side . . . 45
4.1.1.2 The Provider Side . . . 48

4.1.2 Multi-Level Resource Management . . . 51
4.1.2.1 Uncoordinated Policies . . . 52
4.1.2.2 Cooperative Policies . . . 53
4.2 Elastic Virtual Machines . . . 56
4.3 Synthesis . . . 58

5 Contribution: Cooperation Design 60
5.1 Cooperation Protocol . . . 61
5.1.1 Tier Subscription . . . 61
5.1.2 Changing Tier Resource . . . 61
5.1.3 Splitting or Merging Tier Instances . . . 63
5.2 Quota Management . . . 65
5.3 Cooperation Calls . . . 70
5.3.1 Upcall . . . 71
5.3.1.1 Proposals of Virtual Machine Allocation . . . 71
5.3.1.2 Notifications of Virtual Machine Allocation . . . 72
5.3.1.3 Elasticity Proposals . . . 72
5.3.2 Downcall . . . 74
5.3.2.1 Requests . . . 74
5.3.2.2 Confirmations . . . 75
5.4 Synthesis . . . 75

6 Contribution: Cooperative Resource Management System 76
6.1 jTune Framework . . . 77
6.1.1 System Representation . . . 77
6.1.2 Application Deployment . . . 80
6.1.3 Control Loop . . . 81
6.1.4 Communication between Components . . . 82
6.1.5 Runtime Management . . . 82
6.1.6 Usages . . . 84
6.2 jCoop: Cooperative Resource Manager . . . 87
6.2.1 XenManager . . . 87
6.2.2 AmpManager . . . 91
6.3 Synthesis . . . 93

7 Evaluations 95
7.1 Experimental Setup . . . 96
7.1.1 Hardware Testbed . . . 96
7.1.2 Customer Application: RUBiS . . . 96
7.1.3 Metrics . . . 98
7.1.4 Workload Profile . . . 98
7.2 Evaluations . . . 99

7.2.1 Scalability and Elasticity . . . 101
7.2.2 Performance Overhead . . . 102
7.2.3 Virtual Machine Occupation . . . 104
7.2.4 Physical Server Utilization . . . 105
7.3 Synthesis . . . 106

8 Conclusion and Perspectives **107**
8.1 Conclusion . . . 107
8.2 Perspectives . . . 108
8.2.1 Short Term Perspectives . . . 109
8.2.2 Long Term Perspectives . . . 110

Bibliography **112**

Chapter 1

Introduction

Computing systems are continuously becoming more and more complex. These structures evolved from a single machine (personal computer or large mainframe) to clusters, to grids, and recently to hosting centers with complicated distributed systems. The rapid increase in number of machines leads to the complexity of administration. This is often considered as an error-prone and costly task: the administrator not only deploys the applications into the system, but also maintains its state and repairs as failure occurs. Maintaining such large clusters or grids needs to be automated in order to be cost- and time-effective.

Autonomic administration was proposed as a potential option to solve the complex problem of managing clusters and grids [40]. In an autonomic administration system, the system self-manages with given high-level objectives from the administrator, such as application deployment or reactions to failures. As a result, the intervention from the administrator is greatly reduced. TUNe [22] was developed as an autonomic administration system with a high-level formalism for the specification of deployment and management policies. TUNe has been experimented with a variety of application domains: web applications, grid computing systems, and cloud computing systems.

Cloud computing is becoming a global trend for companies to externalize their hardware resources instead of managing themselves. The companies managing the hardware, so-called *providers*, are expected to ensure quality of service for their *customers* while minimizing costs. Power consumption in data centers in 2011 was predicted at 100 billion kWh with peak load at 12GW, equivalent to 25 baseload power plants. Additionally, American Society of Heating, Refrigerating and Air Conditioning Engineers (ASHRAE) estimated that by 2014, infrastructure and energy would contribute 75% to the total IT cost of companies [16]. Many solutions

are proposed and applied to address this power issue. Virtualization, being one of the answers, allows resource mutualization of users on the same resource pool. With virtualization, hardware resources are divided and encapsulated inside virtual machines. The usage of virtual machines in cloud computing increases utilization rate of data centers and speedups application deployments.

To effectively manage resources in cloud infrastructures, the provider and the customer are interested in using autonomic administration systems to handle their resource management tasks. Using such a system, the customer dynamically allocates and deallocates his virtual machines to fulfill his resource needs at runtime, to deal with different load situations of his application and to minimize resource cost. To do this, the customer defines his resource management policy before deployment and instructs his autonomic administration system to follow this predefined plan. On the other hand, the provider specifies his resource management policy so that a minimum amount of physical resources is used. This strategy aims at reducing energy cost for the provider. These resource management policies are complementary and should be coordinated. Very few works in literature focused on exploiting the benefit of a cooperative management policy between these two actors. From this point of view, the objectives of this research work are: (1) pointing out the missing potential cooperation between the customer and the provider, and (2) contributing to the exploration and confirmation of this benefit.

This document presents the work done in the domain of autonomic administration in a cooperative resource management policy. This dissertation is organized as follows.

- **Part 1: Thesis context**

  This part consists of chapter 2, describing the context that motivates this work. This chapter gives an overview of cloud computing in section 2.1. Section 2.2 reviews the base of cloud computing: the virtualization technology. Section 2.3 briefs resource management policies in cloud infrastructures.

- **Part 2: Thesis position**

  This part covers the problem analysis, approach and state of the arts with the above management policies. It includes chapter 3 and 4. Chapter 3 motivates our work by discussing the problems of fixed-size virtual machine (section 3.1) and analyzing elastic virtual machine as a straightforward solution (section 3.2). It then presents the general orientation of our research in section 3.3. Finally, chapter 4 presents the related works with respect to this orientation.

- **Part 3: Contributions**

The main contributions of this research work are presented in this part, including an in-depth explanation of the specifications for a cooperative resource management policy in chapter 6. Chapter 6 details the design and implementation of the *jTune* autonomic administration framework, and *jCoop* as the implementation the cooperation specification. This part also covers the evaluation of our policy with jTune and jCoop in chapter 7. Finally, we conclude our work and discuss the future works in chapter 8.

Chapter 2

Cloud and Resource Management

Contents

2.1 Cloud Computing 5

2.1.1 Introduction and Definition 5

2.1.2 Classifications 7

2.1.3 Advantages and Disadvantages 11

2.1.4 Synthesis 14

2.2 Virtualization 15

2.2.1 Definitions 15

2.2.2 Classifications 17

2.2.3 Advantages and Disadvantages 20

2.2.4 Synthesis 20

2.3 Resource Management in a IaaS 22

2.3.1 Resource Management by the Provider 23

2.3.2 Resource Management by the Customer 24

2.4 Synthesis 24

2.1 Cloud Computing

2.1.1 Introduction and Definition

The difficulties of self-managing infrastructures.

Computer infrastructure has been evolving very quickly, from a single machine to clusters and grids. Large companies usually require large amount of machines to host their business applications (for example: web servers, application servers, database servers, file servers, email servers, authentication services, load balancers, etc). These servers must also be secured from intrusions. On the other hand, small companies need to reduce initial investment for IT to focus on their business. These requirements lead to the following difficulties in order to build and maintain the company's computer infrastructure [15]:

- **Increasing human power**. More complicated infrastructures also mean more requirements on the deployment, configuration, launch, and reconfiguration at runtime. These tasks are either manually performed or automatically managed (but still need to be supervised) by the IT department of the company. However, the first deployment of the whole infrastructure (setting up servers, networks, cooling systems) still must be manually done. Human power investment for the IT department is usually underestimated.
- **Waste of resources**. After deployment, the infrastructure must be well utilized (i.e. it must have load and not being idle). Idle machines not only contribute to the initial investment but also waste power at runtime. Electricity for operating the whole computer infrastructure is always one of the highest parts contributing to the Total Cost of Ownership (TCO). The average cost for powering the servers and their cooling systems accounts for 20% of the total cost [48]. The company must ensure that it provides enough power to keep these servers running. As a result, the company usually needs to improve the usage of the infrastructure, to reduce resource and energy waste.
- **Difficulties to dimension the IT infrastructure**. The infrastructure's workload at runtime cannot be perfectly predicted and allocated at deployment time. There are idle and peak load phases (e.g. during the night and in business hours, respectively). If being over-dimensioned to deal with peak loads, the infrastructure will be more under-utilized during idle periods and contribute to the resource waste. Therefore, it must have the ability to self-adapt to the current load by increasing or decreasing the number of active servers to

handle application's needs. To do this, the design of the infrastructure must be flexible enough to deal with such various situations.

What is cloud computing.

Cloud computing is now a current trend for companies to externalize their computing infrastructure into another type of company. The first actor is called *customer*, and the later is *provider*. This movement is to improve the concentration of each actor: the customers only focus on their business and leave the infrastructure management to the providers. By improving the dedication of each actor, cloud computing model connects the customer's needs with the provider's services.

Since there was no exact formal definition of cloud computing, we can quote a proposal definition, which is rather widely accepted in the research community, from National Institute of Standards and Technology (NIST[1]): ***"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"*** [42]. From this definition, we can summary the following characteristics of cloud computing:

- On-demand Self-service: the customer can request more or less computing resources at runtime. These requests will be automatically served by the service provider *without any human intervention*.
- Remote Access: the resource is available and can be accessed remotely over the network. With the vast improvement of network infrastructures, accessing resource over the network is not a challenge for the customers.
- Resource Pooling: the provider's resource is shared by multiple customers. These resources are dynamically assigned and reassigned to different customers at runtime, when requested. The customer generally does not have any knowledge about the physical location of his allocated resources.
- Rapid Elasticity: the provided resource can be elastically expanded and collapsed *rapidly* at runtime, according to the customer's request. From the customer's view, this resource pool often appears with unlimited amount and can be requested at any time.

[1] http://csrc.nist.gov

- Monitored and Measured: the cloud manager monitors its resource usage with various metrics, and tries to optimize the resource pool in an efficient way. These activities are often transparent to the customer.

Cloud computing is the combination and adoption of many existing technologies. Cloud computing is similar to *grid computing* in terms of hardware deployment and management, but different as cloud computing mostly provides its resource by utilizing *virtualization* [33]. This is the main technology behind cloud infrastructure's manageability and portability of resources. Additionally, *utility computing* concepts are widely used in cloud computing [34]: resources (such as CPU, memory, storage and bandwidth) provided to the customers are metered and billed. The pay-as-you-go billing model is very popular in cloud services nowadays.

In cloud computing, resource sharing is the nature and the main factor bringing the benefit: the provider switches off unused resources while sharing his resource pool among the customers to satisfy their resource needs. Therefore, *Service Level Agreements* (SLAs) are offered by the provider to the customer. A typical type of SLA consists of a set of metrics regarding the *Quality of Service* (QoS) that the provider must ensure at runtime. These metrics can be the number of allocated processors or FLOPS[2], dedicated memory, storage or network bandwidth, etc. Without SLA, the provider may overcommit a lot of customer-booked resources into a small set of physical resources, therefore the customers would not have their desired computing capabilities. SLA is a way to protect the customers to certainly have the resources they booked.

This section briefly shows the overview of cloud computing. Various ways to classify different cloud computing models and their characteristics will be described in the next section.

2.1.2 Classifications

Cloud computing has a long time of evolution. During this time, there appeared many types of cloud, some of which overlapped some others. In the literature, cloud computing has two major official ways of categorizing: by Service Model or by Deployment Model.

When classifying cloud computing by **Service Model**, we use the type of service that the cloud infrastructure provides to the customer. There exists three main types

[2]FLoating-point Operations Per Second

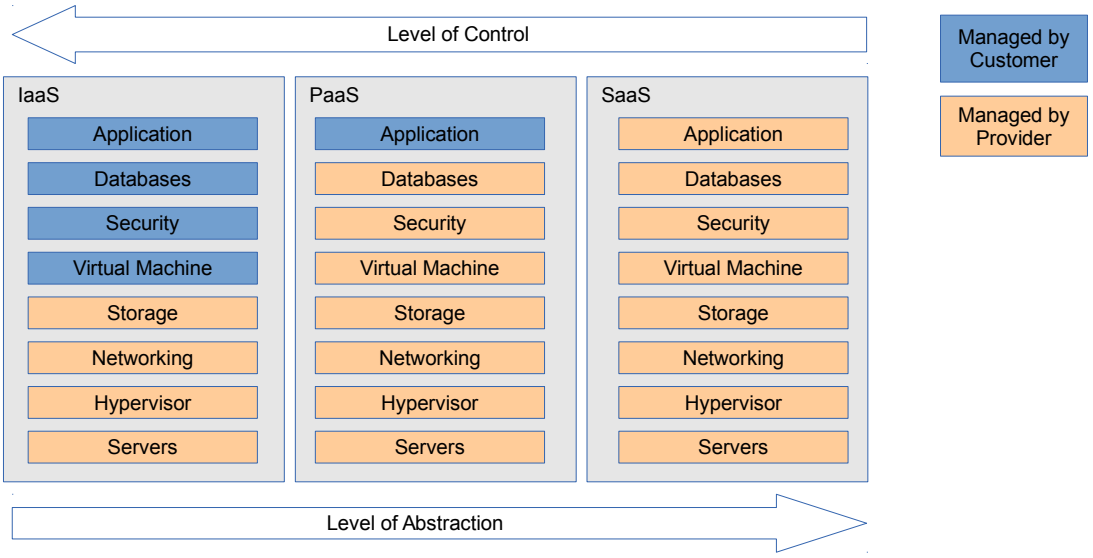

Figure 2.1: Components of each Cloud Service Model

of cloud services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Other models are also proposed as cloud service: Network as a Service (NaaS) and Communication as a Service (CaaS). However, this section only briefly describes the three major types of cloud computing. Figure 2.1 summarizes the management role of each cloud actor, level of control and level of abstraction of each service model.

Infrastructure-as-a-Service is the most basic model of the cloud service. The IaaS provider offers physical machines or virtual machines (the later is more popular, will be described in section 2.2) and other basic resources like network, load balancer, storage. In a virtualized cloud, virtual machine is the primitive form of resources being provided to the customer. The customer is responsible for controlling the operating systems in the provided virtual machines and deploying his own applications on it. In this model, the customer is typically billed on a utility computing basis, meaning based on the amount of allocated resources. The most notable examples of IaaS providers are Amazon Elastic Compute Cloud [1], Google Compute Engine [5], Windows Azure Cloud Services [12].

Platform-as-a-Service is on a higher level when compared to a IaaS: the PaaS offers a virtualized execution platform with predefined set of application programming interfaces (APIs), libraries, services and other tools. The customer develops using these APIs and deploys his application on to the PaaS execution environment. After being deployed, the application is executed in the provider's cloud infrastructure. The customer does not have control of the cloud platform, including network,

[3]Source: wikimedia.

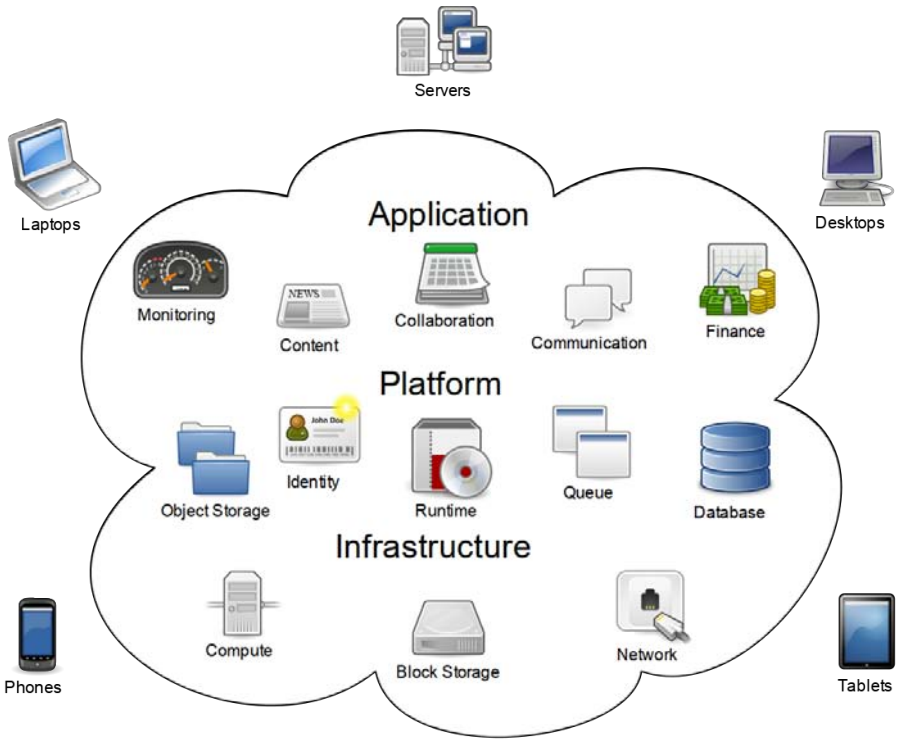

Figure 2.2: Cloud Service Models in Vertical Stack[3]

servers, operating systems, etc. The process of balancing and scaling the customer application is generally performed by the provider's cloud manager and is transparent to the customer. In this model, the customer can focus more on the application business without the need to manage balancing and resizing of his application at runtime. Google App Engine [3], Amazon Elastic Beanstalk [2] and IBM SmartCloud Application Services [8] are major examples of PaaS.

Software-as-a-Service is the highest level of service compared to the above two: the SaaS offers well-defined, pre-developed software to the client. The application is accessible from various client devices with a thin client interface (e.g. a web browser) or the client-side application interface. The provider manages his infrastructure (network, servers, balancing) and the application itself. The provider also develops and maintains the provided application. The customer acts like an end user: he only uses the software from the cloud. The most popular SaaS applications are Google Documents [6], Google Apps [4] and Microsoft Office 365 [9].

Figure 2.2 summarizes the basic components of each service model (IaaS, PaaS, SaaS, respectively) in a bottom-up point of view: the IaaS is the basic foundation of the cloud, the PaaS is usually built on top of a IaaS, and finally applications are then developed on the provided platform.

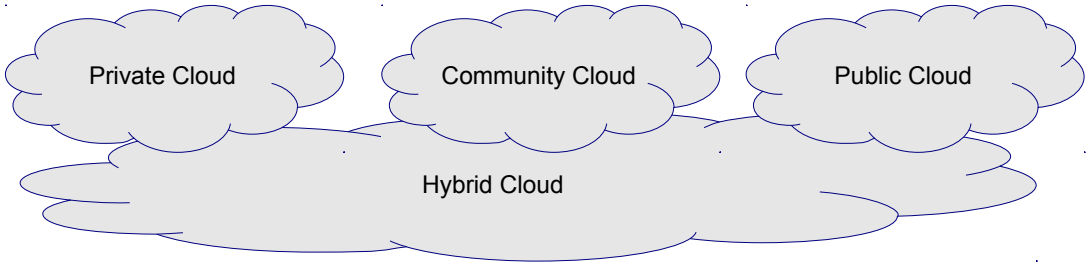

Figure 2.3: Classification based on Deployment Model

On the other hand, when cloud computing is classified by **Deployment Model**, we use the scope of the cloud elements as the main criterion (figure 2.3). This scope also takes into account the customer type. In this type of classification, we identify the following kinds of cloud computing: private cloud, community cloud, public cloud and hybrid cloud.

A **private cloud** is a cloud infrastructure that is only used by one specific organization (e.g. a company, a military or a nonprofit organization). The private cloud is usually deployed, managed and accessed by the organization itself. This type of cloud has more control compared to the others, because the owner have access to both physical and logical resources. There are also virtual private cloud providers (like Amazon, Google, or HP), offering secured and isolated private clouds in a public cloud environment. Although with better security and isolation, virtual private cloud still does not bring full control like a physical private cloud.

A bigger cloud environment, **community cloud**, is targeted by a type of organization (e.g. academic institutions in a country) with a common concern. The goal of a community cloud is to provide the organization with the benefit of a public cloud (such as pay-as-you-go billing model) but with the added level of privacy, security and other requirements which are usually satisfied with a private cloud. The community cloud is owned, deployed and managed by one or more organizations in the community. Its infrastructure is shared among the users.

Being the biggest cloud environment, **public cloud** is open to public use without any limitation on the type of the customer (either personal, business, military or academic institution). A cloud provider has full ownership of the cloud platform with his own policy and shares his resource pool between the public cloud users through network access, mostly through the Internet. This cloud infrastructure serves any kind of applications, depending on the customers, and generally falls into PaaS or IaaS. This is the most popular type of cloud service nowadays. Beside big

and popular providers (Google App Engine, Amazon EC2), there exists more and more smaller providers like Terremark[4], DreamHost[5], EMC[6], etc.

A **hybrid cloud** solution is usually proposed to meet the requirements of the customer. The cloud infrastructure in this case is a combination of two or more smaller infrastructures (private, community or public). These small infrastructures are bound together by the requirements of the customer, either his goals, security concerns, or proprietary technologies. A typical application of hybrid cloud solutions is to extend the capacity of a small cloud during peak load, by requesting additional resources from a public cloud. This method is called cloud bursting. By using hybrid cloud, organizations can obtain both performance (no Internet connectivity is required when the private cloud is used locally) and availability (more resources are available on the community/public cloud when the private cloud is out of resource).

It is also notable that the software required to power these types of cloud infrastructures is still a topic of research and enhancement. The cloud software is categorized as proprietary (e.g. VMWare vCloud) or open source (e.g. OpenStack, CloudStack). It depends on the customer and the provider to choose what cloud software is best appropriate for their cloud requirements.

This section summarized the various kinds of cloud computing, based on the classification model (Service Model or Deployment Model). The next section discusses their benefits and drawbacks to each actor of cloud computing.

2.1.3 Advantages and Disadvantages

Cloud computing, like many other computing systems, has its benefits and drawbacks. This section summarizes the main advantages and disadvantages of this popularity-gaining [15] computing model.

Advantages

Cloud computing brings many benefits to both the customer and the provider, mainly because of each actor's dedication. On the customer side, he simply uses the provided application (SaaS), or is freed from taking care of the availability and maintainability of the computing infrastructure, and focuses on his business application instead (PaaS or IaaS). On the other side, the provider is dedicated to deploying

[4] http://www.terremark.com/
[5] http://www.dreamhost.com/
[6] http://www.emc.com/

and managing the infrastructure (IaaS), developing his platform with more services (SaaS), or developing the application (SaaS). This dedication improves each actor's performance, increases hardware utilization rate and reduces power waste.

For the customer, the computing capability of the whole infrastructure often appears to be unlimited. That means the client can request as much as he wants to deal with the computational needs. Generally, the client needs to deal with idle or peak loads at runtime of his application. To do that, an autonomic resource management system (ARMS) is implemented. In a peak load, the ARMS deploys many instances of the customer's application on the cloud infrastructure to keep the application response time as low as acceptable. The cloud computing capability allows the customer to allocate resources for his application as much as requested.

Combined with the pay-as-you-go billing model, this type of on-demand resource allocation brings benefit to the client because he only pays for what is used: only booked and used resources are billed. By effectively using the cloud infrastructure from the provider, the customer does not need to spend on the computing infrastructure himself, and therefore reduces the initial investment cost. This reduction is critical to many startups and small companies. Large corporations also benefit because their computing infrastructure is huge. Deploying, maintaining and extending such infrastructures are costly, both in terms of human resources and equipments.

The provider shares his own resource pool between various cloud customers. By doing this, the provider has better infrastructure utilization rate: the same hardware can be used to serve many clients at the same time (by providing virtual machines to each customer). After initial investment to deploy the cloud infrastructure, it is mainly the customers who pay for the hardware being used. This model is similar to renting the hardware: the provider benefits from charging his customers for each fraction of his hardware.

Not only does cloud computing benefit in terms of cost for the provider and customer, it also helps in protecting the environment as a form of green computing. By mutualizing the computing resources, both actors have lower physical hardware needed to run the customer applications when compared with non-virtualized, non-cloud infrastructure. Minimizing the amount of occupied physical hardware also means minimizing the energy powering this hardware, and reducing the amount of emitted CO_2 used for generating electricity.

Issues

While cloud computing continues to gain popularity among enterprises, it also raises concerns about its various drawbacks. The main drawbacks include (but not

limited to) data privacy, system security, standardization and resource management. They affect both the provider and the customer in their operations.

A typical and one of the most important concerns of the customer about cloud computing is his data privacy. As cloud computing is a way of outsourcing data and applications, the customer needs to put his data into the cloud, in which the provider shares resources with other customers. The customer naturally raises a question about their privacy of his data in the provider's cloud infrastructure. Therefore, there is a need for implementing appropriate mechanisms to isolate data between customers. Additionally, the cloud provider should be prevented from exploiting the customer's data.

The cloud infrastructure's security is also one of the main concerns. The security issues for cloud computing are generally divided into two categories, according to the point of view of the cloud provider or the customer. On the customer side, physical control of a private cloud (in a local building) is usually considered more secure than having the equipment at the provider's data center. Physical control also means the ability to have direct visibility of the equipments, and is easier to ensure that the infrastructure is not compromised. In the case of a IaaS, the customer needs to enforce better security to his provided virtual machines (including kernel, operating system, virtual network, and the deployed application itself) than in the case he deploys his application in a private data center. A higher security level is required because many customers share the same physical network in the provider's infrastructure. On the provider side, beside the usual security policies employed in a typical data center, an additional level of security must be enforced: the Virtual Machine Monitor (or *hypervisor*, the virtualization software that separates hardware and the virtual machines, see section 2.2.1). It poses more tasks for the provider: the virtualization level must be properly configured, managed and secured [52]. If a Virtual Machine Monitor in one host is compromised, the attacker has access to all the virtual machines running on that host, regardless of the network security system implemented on these virtual machines. Therefore, both actors need to ensure security at their own levels.

Another problem blocking the adoption of cloud computing for enterprises is the lack of standardization. Each cloud provider offers his own APIs for the clients, applications and users to interact with the cloud. This obstructs the development of the cloud ecosystem by preventing users from easily switching between different providers. For example, there is currently no standardized way to seamlessly translate security requirements and policies across cloud offerings [49]. More importantly, proprietary APIs overburden the integration process to the cloud from the company's legacy system (i.e. traditional servers). Additionally, virtualization technology among cloud platforms is not standardized: each cloud provider has his

own choice of virtualization platforms (e.g. Xen [17], OpenVZ [10], VMware [11], etc). These hypervisors do not interoperate because they do not share the same virtualization techniques, file formats, network and storage architectures.

The research community made many efforts to bring open standards to the cloud. IEEE Standards Association (IEEE-SA) is one of the main organizations (beside Distributed Management Task Force, Open Grid Forum and National Institute of Standards and Technology) contributing to the open standards of cloud computing [49]. For example, IEEE-SA has published two working drafts for standardizing the portability and interoperability profiles, as well as for intercloud interoperability and federation. Open Cloud Computing Interface develops the various standards for cloud management tasks, enabling interfacing between IaaS cloud implementations. However, these proposals are not yet finished, accepted and implemented. Cloud computing still needs a long time to have full standards.

Last but not least, there is no unified resource management policy in the cloud. The provider and the customer have their own ways of implementing resource management to solve the cost and scalability problems. The customer tends to reduce the number of allocated virtual machines, because he is billed according to the number of requested VMs in the cloud. The provider wants to minimize the number of running physical servers, because doing this will reduce resource waste and energy consumption. The lack of coordination in the cloud prevents these resource management policies at both levels (the virtualized level and the application level) from fully bringing benefits to all cloud actors.

2.1.4 Synthesis

We summarized the basis of cloud computing, its definition, characteristics and two ways to classify cloud models. We also briefed the benefits and drawbacks when adopting cloud computing to the enterprises. As previously discussed, cloud computing is a means to connect the need of customers with the services of providers. This connection not only improves the dedication and performance of each actor, but also brings benefits to them. The success of cloud computing is backed by virtualization technology. We will describe virtualization in the next section.

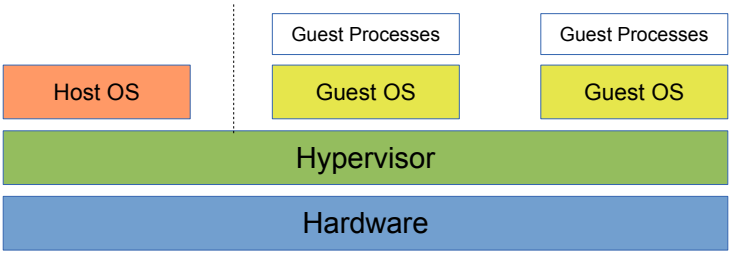

Figure 2.4: Bare Metal Hypervisor *(Type 1)*

2.2 Virtualization

Virtualization is the most important technology in cloud computing because it provides isolation and flexibility for managing computing resources. With virtualization, the provider can easily offer different shares of his hardware to the customers, allowing these resources to be allocated, relocated and deallocated at runtime. This section gives an overview of virtualization, its various types, benefits and drawbacks.

2.2.1 Definitions

Virtualization is a software- and/or hardware-based solution for building and running many operating systems simultaneously on the same bare hardware. Most current implementations of virtualization platform use a software to separate physical hardware and the executing operating systems. These virtualization systems are usually supported by the underlying hardware: the hardware provides architectural support that facilitates the performance, management and isolation of virtual machines.

A virtualized system uses its Virtual Machine Monitor (*VMM*), the virtualizing software responsible for hardware emulation and communication, to share its hardware resources among the guest operating systems (*Guest OS*). The VMM guarantees the independence and isolation among the guest OSes, and therefore provides better security for applications running inside the guest OSes than the same applications deployed in an unvirtualized environment. The VMM is also called *hypervisor*. Most hypervisors support many instances of various operating system (e.g. Unix, Linux, Windows). There exists two main types of hypervisor [46]:

- **Type 1**: also called *native* or *bare metal* hypervisor. This type of hypervisor runs directly on the host's hardware as an intermediate level between hardware

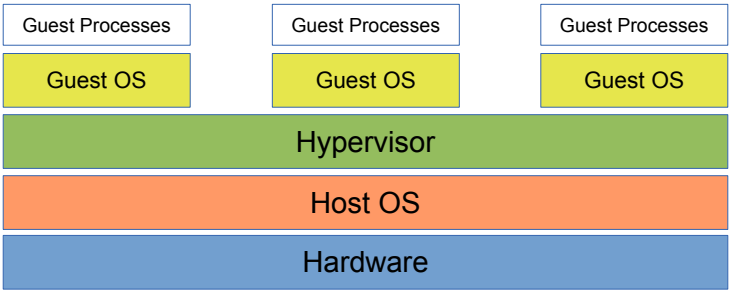

Figure 2.5: Hosted Hypervisor *(Type 2)*

and the guest OSes. Therefore, the guest OS runs at another level above the hypervisor (figure 2.4). This model represents the classic implementation of virtual machine systems. The main benefit of bare metal hypervisor is having less overhead: only does the hypervisor layer stay between the guest OSes and the physical hardware. However, this type of hypervisor is hard to deploy on an existing (and running) system: many type-1 hypervisors require a full hard drive repartition and format during their installs. Most popular examples of type 1 hypervisors include XenServer and VMware ESX/ESXi.

- **Type 2**: also called *hosted* hypervisor. These hypervisors run on top of a normal operating system. The guest OSes suffer from two levels overhead: the host operating system and the hypervisor (figure 2.5). Therefore, hosted hypervisors generally have lower performance than bare metal ones. However, they are easier to adopt in a running system: their installation usually includes kernel drivers and a normal front-end application. They do not severely affect the host operating system and do not require full system format. VMware Workstation and VirtualBox are two main examples of hosted hypervisors.

To support the execution of each guest OS, the VMM emulates a *virtual machine*, providing complete system platform with a complete set of resource (similar to a physical machine): CPUs, memory, graphic cards, audio cards, storage drives, network interfaces, etc. These virtual machines are usually based on an existing architecture, such as Intel x86, ARM, PowerPC. After finishing virtual machine allocation (i.e. allocate physical resources for the virtual machine), the VMM starts the guest OS inside this virtual machine. During the guest OS runtime, the VMM can dynamically allocate additional resources to the virtual machine.

Virtualization ensures *isolation* among the guest OSes by wrapping each guest OS in a separated virtual machine, similar to real operating systems running on different servers. The VMM isolates its virtual machines both in terms of resources and performance. Accessing resource of another virtual machine (storage, memory, etc) is strictly prohibited (and generally there is no way to perform such tasks from a

virtual machine). The VMM also ensures the performance of each virtual machine by protecting resources from other virtual machines' abuse. For example, each virtual machine cannot use more than the defined amount of CPU cycles, memory limit or network bandwidth. The VMM balances the physical resources being used between virtual machines while ensuring performance for applications running on top of these guest OSes.

Virtual machine live migration [26] enables the modification of the customer's virtual machine placements in the provider's data center. It helps the provider to satisfy customers and to reduce energy cost. In a cloud infrastructure with dynamic virtual machine allocation, live migration can pack or spread running virtual machines among physical servers, based on their resource needs. Idle virtual machines can be gathered into as few servers as possible, enabling the possibility to switch off or to suspend the freed servers. On the other hand, overloaded virtual machines can be distributed to less busy servers to ensure their performance to decrease the risk of SLA violation. All of these migrations are executed at runtime, and most likely do not affect the performance of the applications inside these virtual machines. More importantly, the application is guaranteed that there will be almost no message loss during the migrations [26]. As a result, server utilization is greatly improved and power consumption is reduced with live migration.

2.2.2 Classifications

As virtualization has a long time of evolution (dated back to 1960s [18]), there exists many different types of virtualization. In this section we categorize various virtualization technologies in two directions: hardware level and operating system level.

At the **hardware level**, the hypervisor is responsible for *emulating complete virtual machines with all types of hardware resources*: CPU, memory, storage, I/O devices, etc. In this virtualization architecture, *the hypervisor and the host operating system are separated*. Access to all hardware resources is provided by the hypervisor through virtual devices. Additionally, this type of virtualization requires a complete copy of an additional operating system running in each virtual machine, including the kernel, device drivers and all basic system services (e.g. modules to handle different file systems). Note that each operating system instance consume a portion of the storage and memory of each virtual machine. This part can be considered as memory and disk overhead of hardware-level virtualization.

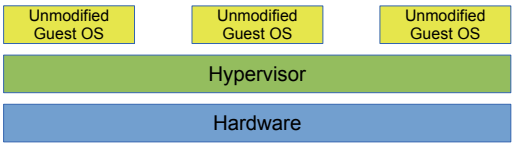

Figure 2.6: Full Virtualization

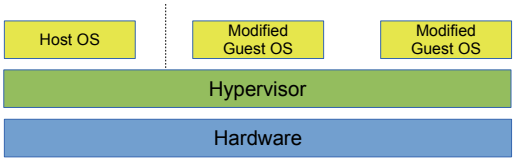

Figure 2.7: Paravirtualization

The most straightforward solution in this level is *full virtualization*. A full virtualization architecture emulates all necessary hardware to allow an *unmodified* guest OS to be executed and isolated (figure 2.6). This includes the emulation of CPU (along with its architecture and instruction sets), storage (hard drive or optical drive, with its controller), graphic card, etc. Full virtualization is usually based on a technique called binary translation, in which all CPU instructions being executed in the virtual machine are parsed, translated and executed in the host CPU (which may be different from the emulated CPU). One of the key challenges for full virtualization is the interception and simulation of privileged operations, such as I/O instructions. As a result, full virtualization is complicated to implement and has low performance when compared to a physical, unvirtualized system.

However, one of the main advantages of full virtualization is the ability to keep the guest OSes unmodified: they are fully abstracted from the underlying hardware by the virtualization layer. In this case, the guest OSes are not aware of being virtualized and do not need to be modified. This advantage is important for running proprietary operating systems (such as Windows, Solaris). Open source operating systems do not need to be run in full virtualization, because they can be modified to meet the virtualized requirements.

Paravirtualization (or OS Assisted Virtualization) exploits such modifications to the guest OS to allow it to be virtualized (figure 2.7). It is different from full virtualization in the sense that the guest OS is aware of being virtualized. The changes to the guest OSes to support paravirtualization are generally minimal and non-intrusive. Because of the possibility to modify the guest OS, paravirtualization has better performance, less overhead and easier implementation when compared with full virtualization. However, paravirtualization is limited only to modifiable operating system, and this limitation greatly reduces the number of supported platforms.

Hardware Assisted Virtualization is a form of full virtualization which uses the support from the hardware (primarily from the host's processor) to ease and increase performance of the guest OSes. As virtualization gained popularity, hardware vendors started supporting it by incorporating features in their hardware to simplify

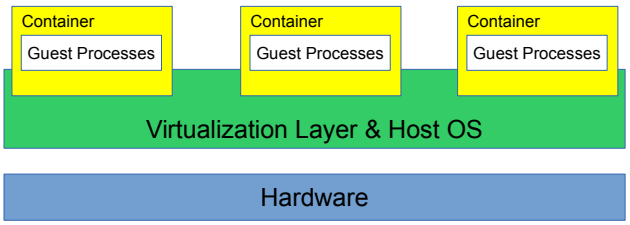

Figure 2.8: Operating System Level Virtualization

virtualization techniques. This support reduces performance overhead when executing virtual machines and reduces changes to the guest OS. Intel and AMD have their own instruction set extensions for virtualization, called VT-x and AMD-V, respectively. However, not all processors and hardware have this capability, and this limits the adoption of hardware assisted virtualization. Most popular virtualization platforms use hardware assisted virtualization when the host hardware is capable of it.

There exists both open source and proprietary implementations of hardware-level virtualization: qemu, VirtualBox, VMware Workstation (full virtualization); Xen, Sun's Logical Domains, VMware ESX/ESXi (paravirtualization).

At the **operating system level**, virtualization is integrated inside the operating system. The *guest processes* are encapsulated into entities called *containers* (figure 2.8). These containers share the operating system kernel with the host and take parts of the hardware resources. Note that each container has its own view of available resources: the number of CPU cores, the amount of available memory, storage size, etc. In this category, *the hypervisor and the host operating system are merged*. This type of virtualization supports a certain form of isolation, but less separated than hardware-level virtualization. With the nature of sharing the same operating system instance among containers, OS-Level virtualization has less performance overhead when compared with hardware level virtualization: there is one layer less in the virtualization stack. The customer's applications running in the containers can have doubled throughput when compared with hardware-level virtualization solutions [53]. Live migration for containers is also possible in OS-Level virtualization. Another advantage of OS-Level virtualization is the ability to easily manage the size of allocated resources on-demand. However, security issue is the main concern in OS-Level virtualization, because all containers share the same operating system kernel with the host. If the attacker can exploit a vulnerability in the guest kernel, it also means that he has exploited the host kernel and theoretically has access to the whole physical system. Major examples of OS-Level virtualization include OpenVZ, Linux Container (LXC) and Solaris Containers.

2.2.3 Advantages and Disadvantages

Virtualization brings many benefits to the hosting industry. One of the main advantages of virtualization is cost reduction, both for hardware cost and for maintenance cost. Many server applications can share the same physical hardware to improve hardware utilization rate, without compromising each server's security. For instance, one physical server can be used for a web server in VM1, a database server in VM2, an application server in VM3. If one server has security breach, the other two are still safe in other virtual machines. In an unvirtualized system, these systems take 3 different physical servers (wasting power when underutilized) to have the same security level. Additionally, virtualized resources are easier to manage with autonomic management systems. A virtual machine acts both as a physical entity (a machine) and a logical entity (a manageable software). Autonomic management systems can interact with the virtualization software to allocate, migrate, or stop virtual machines at runtime. This helps in reducing resource usage and cost. Furthermore, virtualization helps the customer in deployment and dependency management: a prebuilt image of a virtual machine disk containing the application with its dependencies can be easily deployed and replicated into the cloud. The usage of prebuilt virtual machine images can reduce deployment time for additional application instances to serve load peaks. Finally, the ability to migrate virtual machines provides the flexibility of placing them in the infrastructure, according to the actual load status of physical servers.

However, similar to cloud computing, virtualization still lacks of open standards to help interoperability among virtualization platforms. This fact leads to the vendor lock-in problem: when a virtual machine is created using one virtualization platform, it is not easy to port it to another one seamlessly. Furthermore, virtualization introduces performance overhead when compared to a unvirtualized system: the hypervisor itself consumes system resources, as well as all access to the resources from the virtual machines needs to be processed and translated by the hypervisor. In ideal cases, the computing performance overhead is as low as 3% [17], but real world applications and benchmarks showed much larger values, up to 46% [13] when compiling a Linux kernel inside a virtual machine.

2.2.4 Synthesis

Despite of the various drawbacks, virtualization is still widely used in the hosting centers, mostly because its advantages outweigh the disadvantages: hosting centers and customers benefit from cost reductions. Key factors contributing to this success

include: domain isolation, virtual machine migration (server consolidation) and dynamic resource allocation. Isolation improves security in application server. Virtual machine live migration reduces the number of running physical machines. Dynamic resource allocation minimizes allocated resources and reduces cost for the customers.

Resource allocation is further enhanced by various resource management policies, both at the physical level and at the virtualized level. The next section describes in-depth these policies.

2.3 Resource Management in a IaaS

Resource management is one of the most important tasks in cloud computing. Inefficient resource management has a direct negative effect on performance and cost. Ensuring performance and effective use of the resources is a challenge for both provider and customer. In this research work we consider a cloud as a IaaS, in which the provider offers his resources with virtual machines to the customer. This section summarizes various resource management tasks in the cloud for the provider and the customer.

The primitive resource container provided to the customer in a IaaS is in the form of virtual machines. As described in the previous section, each virtual machine has all typical resource types: computing power (CPUs), memory (virtual memory), storage (hard drives), connectivity (virtual network interfaces, IP addresses). Note that most current IaaS providers offer *fixed-size virtual machines*: the amount of allocated resources for each virtual machine cannot be changed at runtime without stopping the virtual machine first. Resource allocation in cloud infrastructures is mostly based on the allocation, relocation and deallocation of the virtual machines.

In cloud computing, various metrics are defined when allocating resource for a customer. A *resource reservation* is defined as a minimal threshold for the resource that the provider must ensure. That means the reservation is used to specify a minimum guaranteed amount of resources: the customer's allocated resource is prevented from being heavy overcommited. This type of threshold is typically defined in terms of absolute units, such as Megahertz for CPU, the number of CPU cores, or Megabytes for dedicated memory of each virtual machine. On the other hand, a *resource limit* is used as a means to restrict the customer from resource abuse. A limit is used to define an upper bound on resource consumption that the customer cannot exceed. Most virtualization platforms support this ability on all major types of resources: CPU cap, memory quota, storage limit, network bandwidth restrictions, etc. Similar to reservation, a limit is also expressed in terms of absolute units, such as Megabytes or Megabits per second.

Resource allocation in a IaaS needs to take into account a lot of various factors, including the heterogeneity of machines. As cloud infrastructure grows, it is difficult for cloud providers to have a large number of identical machines: they need to be added over time. The differences in architectures, hardware specifications and computing capabilities increase the complexity of deciding which physical resource will be allocated upon a resource allocation request. These differences also need to be considered when migrating a virtual machine from one physical server to another. For example, a VM running on a physical machine at 3GHz CPU with 50% quota

must have a different quota value when being migrated to another server at 2.0GHz CPU. This leads to the resource allocation problem: how to effectively distribute the virtual machines, which contain the customer's application instances, among physical servers using the predefined metrics?

This problem is considered at two levels: virtual resource management and application instance management (i.e. at the IaaS layer and at the application layer, respectively).

2.3.1 Resource Management by the Provider

There exists two types of resources that the IaaS provider manages: physical resources and virtualized resources, in the forms of physical servers and virtual machines. The provider is responsible for managing these resources effectively to reach his goal: minimizing operation cost.

Physical servers has only few basic and manageable tasks: power off, power on, suspend and resume. These actions allow the provider to dynamically start and stop the physical machines, and therefore resize the server pool at runtime. Notice that the main difference between power on/off and suspend/resume is the time to execute the operation: starting up a server is much slower than resuming it from sleep state. A cold boot often takes up to several minutes, while a resume task takes less than 10 seconds. The provider needs to take into consideration this difference when deciding to decrease the pool size (during idle periods). Powering off implies a bigger latency when a peak load occurs, so most providers suspend their unused physical servers for saving energy instead of powering them off.

Virtual machines has more options for the provider to deal with peak loads or idle states. The provider can migrate his virtual machines: they can be placed and moved anywhere, on any physical server, without any interruption. This ability ensures performance for the customer and server consolidation for the provider.

To reach the goal of reducing cost, the provider manages his physical servers and allocated virtual machines at the same time, by (1) relocating virtual machines, in order to span as few servers as possible, then (2) switching off or suspending the unused servers to save energy.

2.3.2 Resource Management by the Customer

On the customer side, allocated resources can also be managed: the more unused virtual machines, the more cost for the customer. The ultimate goal for the customer is also to minimize operation cost. To achieve this goal, the customer tends to minimize the number and size of his allocated virtual machines.

To do this, an autonomic administration system (AAS), a branch of autonomic computing [40], is implemented to monitor and manage the customer application instances. The customer's AAS usually includes a set of probes, a decision core, and an effector. The probes monitor each application instance, see if it is idle or is dealing with high load. The information gathered is then submitted to the decision core. This component, in turns, takes into consideration all loads of all instances and selects a choice among (1) keeping the current application instances as is, or (2) reducing the number of application instances in idle states, or (3) increasing the number of application instances to deal with peak loads. This decision is then transmitted to the effector. This last component, depending on the architecture of the application, interacts with the cloud infrastructure to allocate or deallocate virtual machines according to the decision.

2.4 Synthesis

After presenting the resource allocation in the IaaS from different actor's point of views, we conclude that the goal of both actors is to minimize costs, but their way to achieve this goal is different. The provider minimizes the number of physical machines by migrating the allocated virtual machines on as few physical machines as possible, then switching off the freed machines. On the other hand, the customers minimizes his number of running virtual machines with his own AAS.

However, a problem is raised when these resource management policies are operating separately. This problem and our approach toward solving it will be presented in the next chapter.

Chapter 3

Motivation and Orientation

Contents

3.1 Fixed Size Virtual Machine 26

3.1.1 Resource Holes 27

3.1.2 Performance overhead 28

3.2 Elastic Virtual Machine 31

3.3 Cooperative IaaS 36

3.3.1 Approach overview 36

3.3.2 Characteristics 39

3.3.3 Comparison with a conventional IaaS or a PaaS 40

3.4 Synthesis 43

The previous chapters showed the goal and approaches of each cloud actors toward minimizing operating costs in a conventional IaaS. In summary, the customer *dynamically requests* the allocation and deallocation of *fixed-size virtual machines* to serve his application. On the other side, while satisfying the customer's demand for computing resources, the provider ensures *server consolidation* to minimize power consumption by live migrating allocated virtual machines and suspending freed servers.

There exists several ways to optimize the infrastructure utilization rate. This chapter discusses various issues in resource management in a conventional IaaS and describes our proposal to solve these problems.

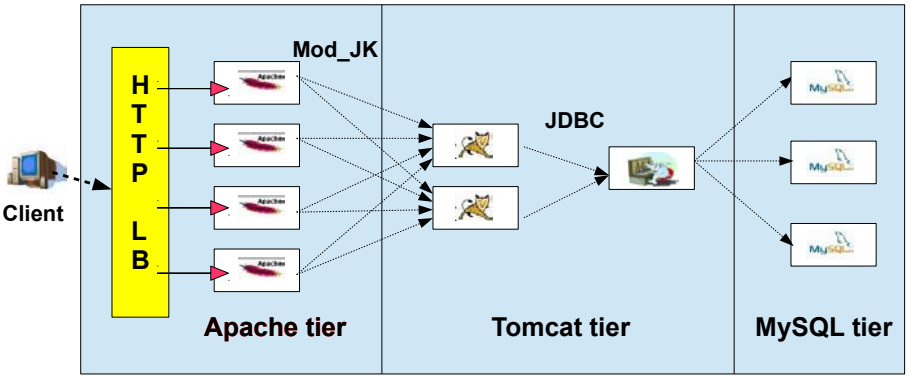

Figure 3.1: Typical J2EE Architecture

3.1 Fixed Size Virtual Machine

Most IaaS providers currently offer a set of *fixed size virtual machines*: any allocated virtual machines are assigned with a fixed set of resources and cannot be resized at runtime. To resize a virtual machine, the customer firstly need to stop it, change the required resource size and finally restart it. This limitation poses inconvenience to the customer when he wants to resize his virtual machines according to the application load.

A naive approach to have dynamic application size at runtime with fixed size virtual machines is to use application *replication*: instead of having a single application instance, the customer uses multiple instances. When the workload increases, new application instances are dynamically added. In contrast, running instances are removed when workload decreases. Depending on the application type, parts or all of the application instances can be replicated. This mechanism is effective with master-slave applications using load balancers.

For instance, a typical web application in Java 2 Platform Enterprise Edition (J2EE) is a popular example of a master-slave application being deployed in the cloud (figure 3.1). Such application represents the commonly hosted applications in cloud platforms. Its design consists of a web server tier (e.g. Apache), an application server tier (e.g. Tomcat) and a database server tier (e.g. MySQL). When a HTTP request is received, it refers either to a static web document (e.g. HTML, CSS), in which case the web server directly returns the requested document to the client; or to a dynamically generated document, in which case the web server forwards the request to the application server. In turn, the application server executes requested application components (e.g. Servlets, EJBs), creating queries to a database through a JDBC driver (Java DataBase Connection driver). Finally, queried data from the

database (e.g. MySQL) is processed by the application server to generate a web document which is returned to the client.

In this example, the customer application is replicable, consisting of 3 tiers: the web server tier, the application tier and the database server tier. Each of these tiers can be *replicated* to deal with important of work load: to serve a large number of static files (e.g. static images at the web server), to finish a complicated task in the application (calculations at the application server), or to execute a long query in the database (making reports from multiple large tables in the database server). Each load balancer distributing incoming requests to application tiers should be able to take into account the weight of each managed tier instances: instances deployed with more resources is forwarded with more requests than instances with less resources. When being deployed in a cloud environment, each tier instance can be wrapped inside a virtual machine to ease the management of the tier instances.

3.1.1 Resource Holes

In a conventional IaaS, the provider usually offers fixed size virtual machines from a set of different instance types (specified size for each type of resource). These instance types allow the provider to have a wide range of customers. When fixed size virtual machines are created and released dynamically at runtime, the provider's infrastructure gradually becomes fragmented, similar to filesystems in storage disks. As a consequence, resource holes appear at runtime in the cloud and the provider has less flexibility for server consolidation. The resource holes at runtime depend on the allocated virtual machine sizes. From the provider's point of view, the ability to move virtual machines (also called the **flexibility** of virtual machines) between physical machines to ensure server consolidation is the most important factor. Small virtual machines are more flexible for migration than big ones, because they are more likely to fit available slots in physical machines. As a result, small virtual machines are better to fill the holes.

An example to show the flexibility of small virtual machines over big ones is described on figure 3.2. This illustration considers memory as the main resource-constraint factor. The left part shows two occupied physical machines (PM1 and PM2) with associated virtual machines on each. The allocated memory for each virtual machine and free memory are also noted. PM3a denotes a case when the customer uses a big virtual machine. PM3b shows another situation when the same customer uses two smaller virtual machines, each consumes 5GB of memory, instead of a big 10GB one.

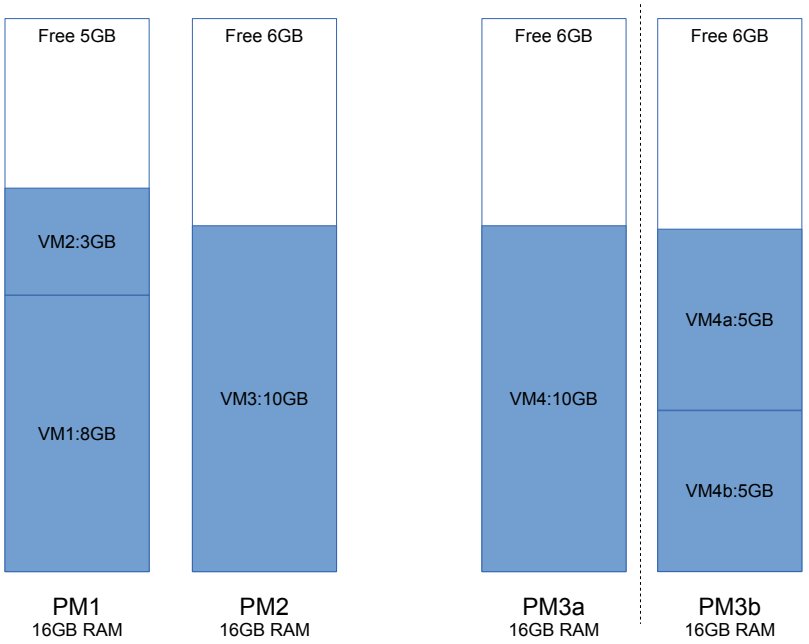

Figure 3.2: Example of Flexibility from Small Virtual Machines

If one big virtual machine is allocated (PM3a), the provider does not have the ability to migrate it to PM1 or PM2, because the free memory on PM1 or PM2 is not enough for this big virtual machine (requiring 10GB of memory). In this situation, although the total free memory (11GB) fits VM4 (10GB), the provider still need to keep all 3 PMs running. In contrast, if the customer chooses the second placement (two virtual machines, each consuming 5GB of memory), like in PM3b, the provider has the ability to migrate these virtual machines to PM1 and PM2 to shut down PM3b.

From this example, it is clear that fixed size virtual machines do not bring flexibility to the provider for the best possible optimization regarding server consolidation. Another issue with fixed size virtual machine is performance overhead, which will be described in the next section.

3.1.2 Performance overhead

Because a fixed-size virtual machine cannot be resized after its allocation, the customer needs to add or to remove application instances (wrapped into virtual machines) to deal with workload fluctuation at runtime. He typically requests to allocate new virtual machines to deal with the increase of the application's workload. As a result, there exists situations where multiple virtual machines of the same

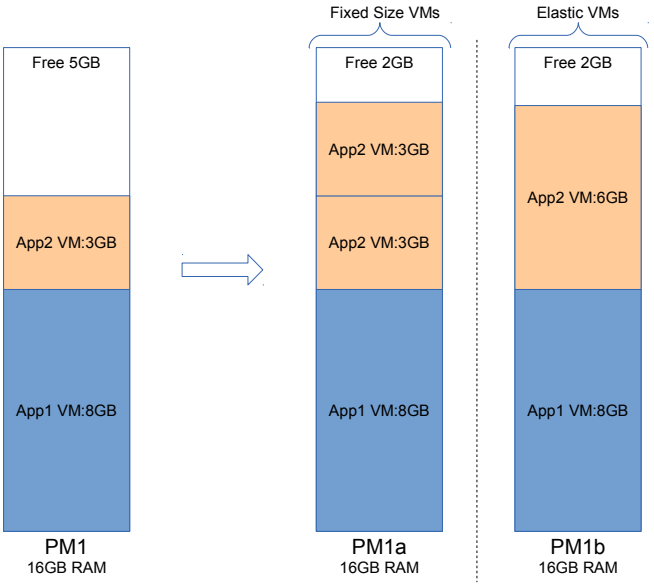

Figure 3.3: Dealing with Load Increase

application are running on the same physical machine. An example is illustrated in figure 3.3: to deal with a load peak, App2 allocates another fixed-size virtual machine. In this example, two virtual machines of the same application 2 are running on PM1a. This placement (PM1a) generates several types of performance overhead:

- *Virtualization overhead* is the performance penalty caused by the hypervisor. Since the hypervisor is a software itself, it consumes system resources, including CPU cycles and memory. CPU is used in the hypervisor for doing all virtualization tasks, including making decision for the next virtual machine time slice, processing hypercalls, looking up memory pages for guest OS memory access, etc. Xen, a typical hypervisor, introduces 3% to 45% CPU overhead in benchmarks [13, 17]. Hypervisors use a part of the host's memory for managing virtual machine, hash tables in content-based page sharing, etc. Having more virtual machines on a host means more hypercalls, more pagetable lookups, more allocated memory for virtual machine's page mapping, etc. As a result, the more virtual machine shares the same host, the more virtualization overhead is introduced.

- *Balancer overhead* occurs when requests to the customer's applications are passed through a load balancer. A balancer needs to maintain a list of managed tier instances and distributes its requests to the worker instances. Introducing a new layer (balancer) into the application architecture also adds one more step in the request flow, further increasing application response time.

Reducing the number of application instances acts as an important factor to reduce performance overhead. For *the hypervisor*, a reduced number of virtual machines leads to less virtual machine's context switches, less physical-to-machine memory mapping tables, more cache hit, etc. Similarly, with a smaller number of virtual machines, the IO virtualization system has less network interfaces and disk images to virtualize, reducing overhead. At the highest level, the balancer is less stressed with less number of tier instances.

To summary, this section described the two main problems introduced in the cloud infrastructure when using fixed size virtual machine, namely resource holes and performance overhead. A straightforward solution is to use variable size virtual machine for resource allocation and deallocation. This direction will be discussed in the next section.

3.2 Elastic Virtual Machine

Elastic virtual machines are the ones with the ability to be resized at runtime, without the need of being stopped first. This section discusses the benefits and issues of using elastic virtual machines in the direction of optimizing resource usage (reducing resource holes and performance overhead) in cloud infrastructures. There exists several dimensions when considering a size for virtual machines: CPU, memory, network bandwidth and virtual storage.

CPU resource management in hypervisors, being one of the most important tasks in virtualization, has been well researched and implemented with various types of hypervisor's CPU schedulers. Note that these schedulers distribute CPU resources between virtual machines, which is *at a lower level than the operating system's CPU schedulers* (the latter balances CPU power between processes inside guest operating systems). Each hypervisor's CPU scheduler has its specific parameters: weight, vcpu, affinity, credit, reserve, limit, etc. Most of these parameters can be dynamically reconfigured at runtime. This ability provides the *elasticity* of CPU resources for virtual machines.

Memory is one key resource limiting server consolidation rate in cloud infrastructures. Each virtual machine, even being idle, consumes a certain amount of physical memory. This memory region cannot be used by any other virtual machines. Additionally, guest operating systems expect to be executed in a physical machine, with a fixed amount of memory. Current virtualization platforms implement dynamic memory mechanism by pretending that the guest operating system has the maximum amount of memory allocated (the limit that it can own) at its boot time. A special driver in the guest operating system, named *ballooning driver*, then inflates a balloon of memory to claim the unused memory and passes this region to the hypervisor. As a result, the amount of available memory for the guest operating system is reduced. By inflating and deflating the balloon of memory, the hypervisor can dynamically change effective memory size for its managed virtual machines at runtime.

Network throughput is a less important resource than the two types of resources above in cloud infrastructures. Most hypervisors support throttling their virtual machine's virtual network interfaces to ensure Quality of Service (QoS). More importantly, they also support the elasticity of network bandwidth, i.e. the bandwidth can be dynamically increased or decreased at runtime. This ability helps the administrator to resize his allocated virtual machines in the network dimension on the fly.

Lastly, **virtual storage** is another dimension of virtual machine size. The most straightforward approach to change the size of virtual storage of a virtual machine is to attach or detach virtual hard drives (similar to resizing a multi-tier application by adding or removing application instances). Additionally, most modern hypervisors support real dynamic resizing for storage of running virtual machine, i.e. to change the size of a running virtual hard drive. The ability of hypervisors to resize virtual storage complements the elasticity of virtual machines. Using all of these capabilities, the provider can fully support resizing virtual machines in all dimensions: CPU, memory, network and storage. In our work, we consider only elastic CPU and memory resource management because they are the most concerned resources in a cloud infrastructure.

Majorities of the public cloud providers (Google, Microsoft, Amazon, HP, etc.) do not allow their customers to get involve in the physical resource management process (e.g. resize their virtual machines at runtime). Fixed size virtual machines are easier to manage (packing and migrating) because all resource dimensions are well defined: each value (size) in each dimension (CPU, memory, bandwidth and storage) does not change at runtime. The resource management policy on the provider side is therefore *less complicated* and *executes faster*.

Improvements

If elastic virtual machines are used in cloud infrastructures, they will bring significant benefits to both actors.

On the **customer side**, since the virtual machines can be resized at runtime, the number of virtual machine allocations and deallocations to deal with workload variation is dramatically reduced. Furthermore, elastic virtual machines provides a finer-grain resource management to the customer (e.g. adding 200MB of memory to an existing virtual machine), in contrast with the coarse-grain resource management of fixed-size virtual machines (each can consume at least 500MB of memory). Finally, time for resizing a virtual machine typically is much shorter than allocating a new one, because a virtual machine allocation needs to take into account its considerable boot time. As a result, the fluctuation of application load can be instantly dealt by resizing its virtual machines. An example of this benefit can be illustrated in figure 3.3: App2 can simply resize its elastic virtual machine to 6GB, resulting in PM1b. In this example, only one application instance is used to deal with the load peak. This behavior *improves application's responsiveness* and *reduces performance overhead* when compared with a fixed-size virtual machine allocation (in PM1a).

On the **provider side**, if he offers elastic virtual machines to the customer, finer-grain resources will reduce resource fragmentation when compared with providing

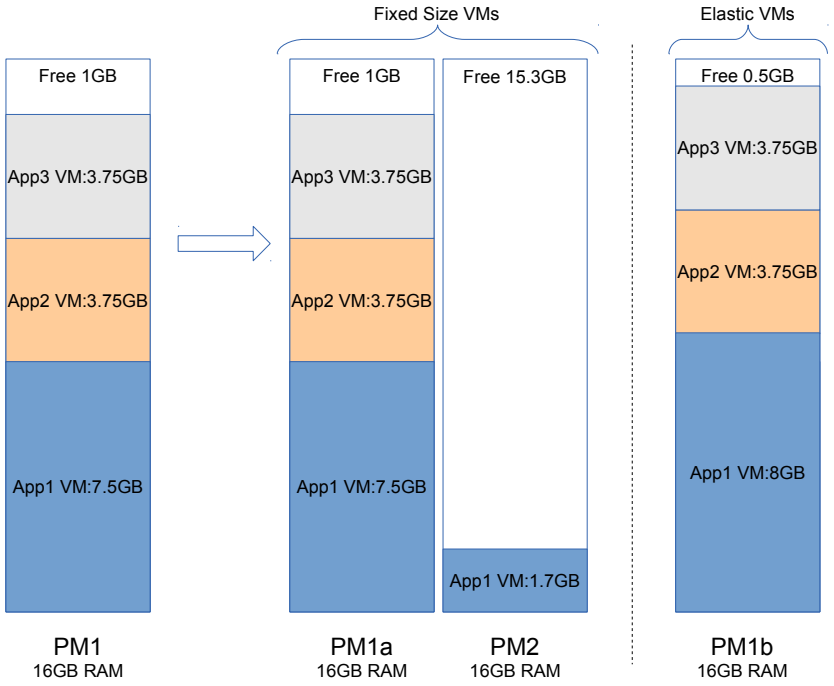

Figure 3.4: Finer-Grain Resource of Elastic Virtual Machine

fixed-size virtual machines: a size-up of a virtual machine can use memory that would not have been sufficient to host another virtual machine. As a result, there will be *less resource holes* (in both number and size of holes) for the provider.

Figure 3.4 considers a case when a IaaS EC2-like provider offers virtual machines with the instance types having 1.7GB, 3.75GB, 7.5GB, 15GB of memory (similar to EC2's m1.small, m1.medium, m1.large and m1.xlarge, respectively). The left side of this figure represents a scenario when 3 virtual machines of 3 different applications are collocating on a single server PM1. We assume that the application 1 is dealing with a load peak: it requires an additional 500MB of memory.

If this is a conventional IaaS, i.e. the IaaS provider offers fixed-size virtual machines, this application will need to allocate another virtual machine for its resource requirement. Since the smallest instance type takes 1.7GB of memory, the provider must start a new physical machine, PM2, for this allocation (figure 3.4, PM1a and PM2). As a result, PM1a's 1GB RAM free is unable to be used, leaving a resource hole in PM1a. Additionally, a new server must be switched on and consumes energy.

On the other hand, if this IaaS provider offers elastic virtual machines, he can use the remaining 1GB of free memory for this resource requirement: a resize of virtual machine of this application, from 7.5GB to 8GB, is enough to meet the application's needs (PM1b). In this case, the provider does not need to start a new server to

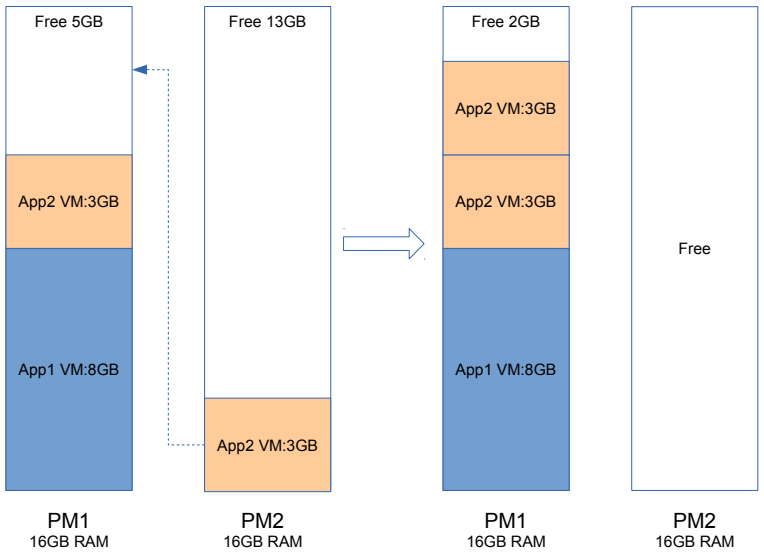

Figure 3.5: Multiple Instances of the same Application running on the same PM after a Migration

satisfy the customer's request, and his physical resource is more effectively used: both resource and power is not wasted.

Remaining Issues

Although elastic virtual machine bring improvements for both actors like discussed in the previous section, they do not completely solve the problems of *resource holes* and *performance overhead* when being used in cloud infrastructures.

Resource holes. There exists situations in which elastic virtual machine as a sole solution cannot optimally fill resource holes. We can reference to figure 3.2 as an example. In this scenario, PM1, PM2 and PM3 host four elastic virtual machines for four different applications. Although total available memory of PM1 and PM2 (11GB) are enough for the total need of VM4 (10GB), the provider cannot use these resources. A solution to fill this resource hole is to split VM4 into VM4a and VM4b. This shows an opportunity for the provider to ensure server consolidation by migrating VM4a and VM4b to PM1 and PM2 respectively, filling these remaining holes in his infrastructure. Elastic virtual machine alone is not capable of splitting virtual machines. The lack of ability to *split virtual machines* prevents the provider from further defragmenting his infrastructure.

Performance overhead. Similarly, elastic virtual machine alone is not enough for minimizing performance overhead. Naturally, the provider uses virtual machine migration to consolidate his servers and improve infrastructure's utilization rate.

The migration process can result in non-optimal virtual machine placement: there are possibilities that the customer's virtual machines of the same tier are running on the same physical server. Figure 3.5 shows an example of this situation, when a virtual machine of the application 2 is migrated from PM2 to PM1, so that the provider can shutdown or suspend PM2. This placement faces an application's performance overhead for the application, which could be solved by merging these two small virtual machines (3GB each) into a bigger one (6GB). However, this merge cannot be done with elastic virtual machines only as the customer is not aware of the fact that the 2 virtual machines got collocated, and the provider is not aware that the 2 virtual machines belong to the same tier. Note that figure 3.5 is different from figure 3.3: the former shows the collocation problem with a virtual machine migration (triggered by the provider) while the latter shows the same problem but with a virtual machine allocation (triggered by the customer). The lack of ability to *merge virtual machines* reduces the chances to optimize application's performance for the customer.

The missing point in this situation is the collaboration of the two cloud actors. The next section describes our approach to use elastic virtual machine with the cooperation of two cloud actors so that an optimal resource management is achieved, in terms of reducing resource holes and performance overhead.

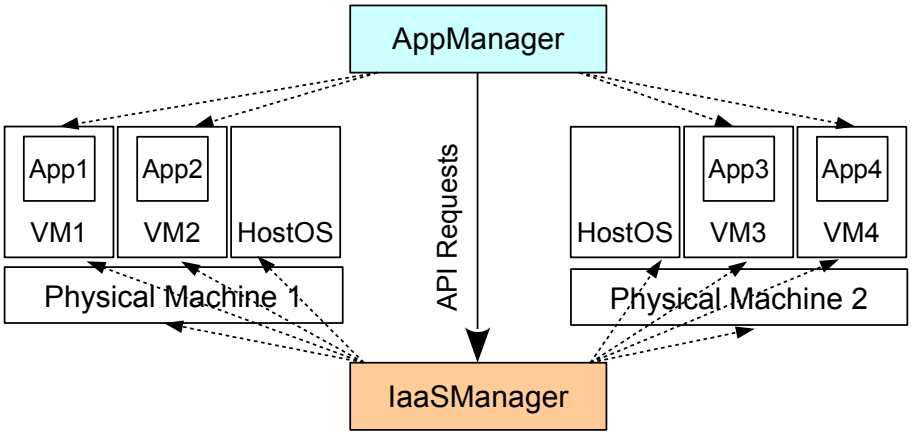

Figure 3.6: Traditional IaaS

3.3 Cooperative IaaS

The previous sections discussed the problems of resource holes and performance overheads, caused by fixed size virtual machines and uncoordinated resource management in most current IaaS. To resolve these issues, both cloud actors need to share their resource management knowledge to each other. Several methods can be applied to exploit the advantages of sharing knowledge between the actors. This section proposes an approach to improve resource usage and application performance by introducing the notion of **cooperative IaaS** with the ability to split and merge virtual machines.

After the approach description, we then make a comparison to study the similarities and differences between a cooperative IaaS and a traditional IaaS or a PaaS, which are the most two popular cloud models being provided currently. Various points of view of the cooperative IaaS are also discussed to have better perspectives about this work.

3.3.1 Approach overview

Most **conventional IaaS** systems work similar to figure 3.6. The customer has his own application manager (*AppManager*), while the provider has his infrastructure manager (*IaaSManager*), providing and managing fixed-size virtual machines. The *AppManager* can invoke services from the *IaaSManager* with various types of API calls, provided by the provider. The most popular calls include: allocate, deallocate, start, restart or stop virtual machines. Most providers also offer APIs to reconfigure a virtual machine (adding or removing resources) when it is in the

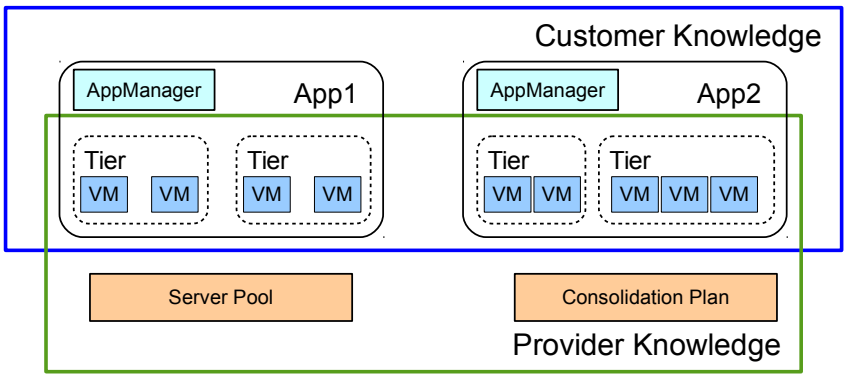

Figure 3.7: Cooperative IaaS: Knowledge of each Cloud Actor

stopped state, but using them causes unwanted service breakage to the customer: in order to resize a virtual machine, the customer needs to stop all running services, stop the virtual machine, resize, start, wait for it being ready and finally resume all services. The cloud API requests are usually implemented as RESTful API with HTTP or HTTPS as the protocol and JSON/XML for resource representation. As a result, the *AppManager* can access the cloud APIs from either inside or outside of the cloud. In current IaaS systems, the provider usually does not send any notifications about the infrastructure changes to the customer, e.g. virtual machines of the customer have been migrated from one machine to another machine. Hosts are transparent for the customer.

Unlike the traditional counterpart, we propose a **cooperative IaaS** with the insistence on sharing knowledge about applications and virtual machines between the two actors (figure 3.7), in order to improve mutual benefit and raise possibilities to improve resource management, particularly:

- **The customer** provides information about his application (workload characteristics, tiers, etc) to the provider; and
- **The provider** shares the knowledge about the placement of the allocated virtual machines to the customer.

In case of a multi-tier application, the shared knowledge includes **tier information** (which application instances are in each tier, this kind of information is typically not shared in a conventional IaaS). In our cooperative IaaS, once the information about application tier is shared, the provider can propose to split or to merge virtual machines to the customer at runtime, based on the current virtual machine placement. Splitting and merging virtual machines can help to further reduce resource holes and performance overheads, as concluded in the previous section.

In our cooperative IaaS, the resource management policy *shifts the decision to add or remove virtual machines from the customer to the provider*. That means instead of requesting the provider to allocate or deallocate individual virtual machines, the customer only needs to request the total computing power (amount of virtual CPU cores, amount of memory, etc.) that he really needs. According to these required parameters, the *IaaSManager* automatically decides how many virtual machines will be allocated and how big each virtual machine will be. Based on the actual placement and size of the application's virtual machines at runtime, the *IaaSManager* either scales the application tier *horizontally* (adding/removing more virtual machines), *vertically* (increasing/decreasing size of the existing virtual machines), or both. With the support of elastic virtual machines, this behavior brings much more flexibility in terms of resource management to the provider and manageability to the customer:

- **The provider** is able to make decision about when to allocate, deallocate, relocate and resize the virtual machines at runtime. This ability helps to improve the utilization rate of the infrastructure. Note that, the elasticity of virtual machines is important and is well used in the cooperative IaaS, unlike in a conventional IaaS where fixed size virtual machines are used.
- **The customer** is only responsible for requesting changes to the resources associated with the global tier, and deploying application instances to his allocated virtual machines. If the application has multiple tiers, he also needs to reconfigure his load balancer upon virtual machine allocation and deallocation.

Note that the cooperation of these two actors is optional. That means our proposed cooperative policy has backward compatibility and is able to work as a legacy resource management policy, in which the customer makes requests for allocating and deallocating virtual machines at runtime, and the provider does not create any notification about his virtual machine's relocation or resize to the customer. In this case, knowledge is not shared, and there is no collaboration between the two actors. This behavior is the same with most traditional IaaS.

In a cooperative IaaS, the two actors need to have a bi-directional channel to communicate with each other (figure 3.8). This channel is used for transmitting customer's requests to the provider (e.g. quota change for a tier), customer's notifications to the provider (e.g. an application tier has finished undeployment, so that the IaaS can stop the virtual machine), provider's notifications to the customer (e.g. a virtual machine's reconfiguration, resize, or started/stopped), etc. The two actors also need a common protocol and communication API for requests and responses. The communication channel and protocol will be described later in chapter 6.

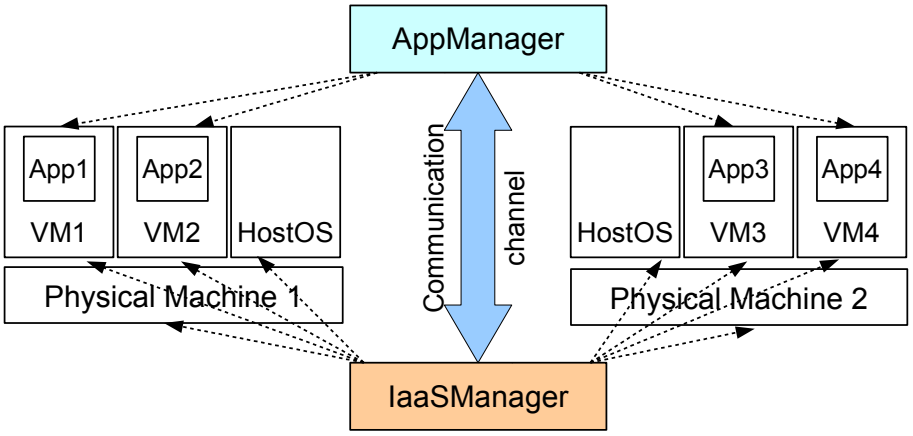

Figure 3.8: Communication Channel in a Cooperative IaaS

To summarize, a cooperative IaaS lets the customer decide the required amount of computing resources for his application. The provider is then responsible for organizing a suitable set of dynamic-size virtual machines, so that they satisfy the demanded resources. In order to do that, the two actors need to have a close collaboration.

3.3.2 Characteristics

Previous sections showed the importance of the cooperative resource management policy by raising two main problems: resource holes and performance overhead. The cooperation policy proposed in section 3.3.1 has the following characteristics:

- **Mutual Benefit**. The first and foremost goal of the policy is to bring benefits to both cloud actors. The cooperation policy must be able to improve benefits in terms of performance and resource utilization ratio: the provider helps the customer to lower system overhead by merging virtual machines, while the provider can reduce his physical resource usage with the support from the customer with the agreement for splitting virtual machines. Additionally, the customer has simplicity in resource management (by focusing on managing his application) while still using pay-as-you-go billing model. On the other hand, the provider has flexibility in determining virtual machine size and their placement, in order to filling the resource holes. As a result, he can lower price for the customer and be more competitive.

- **Fine-grain Resources**. The cooperative resource management system, as discussed above, takes the advantages of dynamically resizing virtual machines. This ability provides a finer-grain resource management to the customer than

the traditional virtual machine's allocate-deallocate mechanism, and allows the physical resource to be better used.

Additionally, our implementation of a cooperative resource management system is expected to have the following characteristics:

- **Modularity**. The cooperative resource manager is flexible and extensible for both actors at runtime: its functionality can easily be added and removed by reloading the modules. Modularity ensures that the policy can be customized according to the needs of each actor. Modularity also enables the developers on both sides to implement and reuse their components effectively. For example, the provider can replace his consolidation manager module at runtime, to have a better consolidation algorithm, without the needs of a whole system's redeployment.
- **Adaptability** is an important characteristic to ensure that the policy can be used in various environments. There exists many hypervisors available for the provider: VMware ESXi, XenServer, Microsoft Hyper-V, KVM, OpenVZ, etc. Additionally, the customer's application can range from typical multi-tiered web applications to high performance computing applications. The cooperation policy should be adaptable in order to fit various types of hypervisor (of the provider) and application (of the customer) architecture. This ensures that the policy can be widely implemented and bring benefit to as many scenarios as possible.

3.3.3 Comparison with a conventional IaaS or a PaaS

After the description of the cooperation policy in the previous section, we highlight the similarities and differences between a cooperative IaaS with a traditional IaaS or a PaaS (figure 3.9). Notice that our proposal is in a form of resource management policy, which is an extension of a traditional IaaS. In order to be implemented in a IaaS, the resource management behaviors of both customer and provider need to be extended.

When compared with a **conventional IaaS**, it is clear that a cooperative IaaS is at a higher level in terms of resource management abstraction. The customer is not responsible for managing the virtual machines anymore: he is provided with a set of virtual machines satisfying his desired computing power, and is responsible for handling the deployment, configuration, execution and undeployment of the application

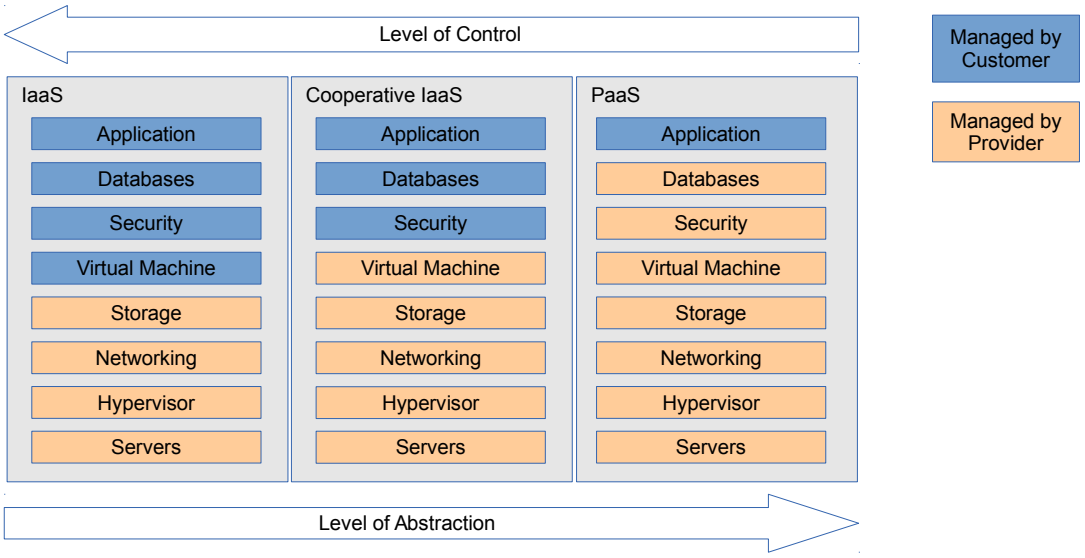

Figure 3.9: Comparison of a cooperative IaaS with traditional IaaS and PaaS

instances on this set. All management tasks regarding virtual machine placement are handled by the provider. These tasks include allocating, deallocating, migrating, resizing, splitting, merging virtual machines. Since the provider is the one managing both physical and virtual resources (servers and virtual machines), he has the required knowledge for optimizing resource usage. The missing piece of knowledge in resource management in the traditional IaaS is the placement and groups of tier instances.

This separation improves the dedication of each actor in resource management, increasing the effectiveness of resource management, in terms of reducing performance overhead and power consumption, as well as improving physical resource usage. The efficiency will be better analyzed in chapter 7.

A conventional IaaS also supports a tier notion, but in a different term than a cooperative one's. For example, Windows Azure has the definition of *Availability Sets*, and Amazon EC2 has *Auto Scaling Group*. These group notions provide the ability to automatically scale the group by actively monitoring the running instances' loads in the group. This group notion is different than the one's in a cooperative IaaS, in these terms:

- The virtual machines containing application instances in a *conventional IaaS group* must be pre-deployed, pre-configured and ready for startup. A cooperative IaaS does not need them to be ready, because they will be deployed, configured and added to application tier at runtime by the customer. This behavior is true on-demand resource allocation on both sides.

- The tier groups in a conventional IaaS are limited in size. They cannot be expanded over a number of pre-configured virtual machines. In contrast, a cooperative IaaS has virtually unlimited size for any tier, because additional virtual machines will be created by the provider, then instances will be deployed and configured by the customer.

Another major difference between a conventional IaaS and a cooperative IaaS is the usage of elastic virtual machines. Most traditional IaaS providers (Amazon, Google, Microsoft, Rackspace, etc) do not offer elastic virtual machines. These offered virtual machines are grouped into different *instance types* to support wide range of customer's needs. In contrast, our cooperative IaaS relies on elastic virtual machines to allow merges and splits.

On the other hand, similar to a **conventional PaaS**, the cooperative IaaS is also based on a IaaS. Most current PaaS cloud platforms are based on a self-managed or public IaaS. This frees the PaaS manager from having the complication of virtual machine management, so that he can focus on managing the platform with its services. Additionally, the notion of virtual machine is well hidden in a PaaS: the customer does not have any knowledge about the virtual machine that his application is running on. On the other side, the cooperative IaaS is also based on a IaaS with the extension of resource management policy, customer-provider communication channel and protocol, and takes care the virtual machine management for the customer.

However, the main difference between a cooperative IaaS and a PaaS is the amount of offered services. A cooperative IaaS provides much less services than a traditional PaaS does: it does not provide to the customer any application design, development environment, database, cross instance messaging services, security policies, etc. Because of these services, the customer in a PaaS has more vendor lock-in than one in a (cooperative) IaaS. The more provided services, the more dependency the customer has. As a result, substantial switching cost needs to be taken when he wants to change to another provider. In contrast, customer in a cooperative IaaS is provided with less services and has more freedom. He is still free to develop his application on any software development platform, deploy it in the IaaS, manage and undeploy as he wants, similar to a conventional IaaS.

A cooperative IaaS can be seen with various **point of views**. Firstly and primarily, it can be considered as *an extension of IaaS*, like analyzed above. Secondly, it can be viewed as *a hybrid IaaS-PaaS model*. A cooperative IaaS provides a set of small platform services, so that the customer can develop his application without worrying about the virtual machine placement and allocation. In this case, the

platform services include customer-provider communication and automatic virtual machine management.

Additionally, a cooperative IaaS can be considered as serving a set of *distributed virtual machines* (DVM). Each application tier (each tier consists of a set of instances) is represented by a distributed virtual machine. The customer sees his whole tier as a logical and distributed virtual machine. In this view, each logical virtual machine has virtually unlimited amount of resources (CPU, memory, etc), and can be resized dynamically at anytime. The customer only needs to specify the desired amount of resources, and the provider will organize this distributed virtual machine internally as long as the resource constraints are met. From the distributed virtual machine point of view, the J2EE example in the previous section (figure 3.1) can be seen as an application with three distributed virtual machines: one DVM for Apache, one DVM for Tomcat and another DVM for MySQL.

3.4 Synthesis

In this chapter, we discussed the problem of resource holes and performance overheads in IaaS clouds. We analyzed this problem with both fixed-size and elastic virtual machines. We studied several scenarios to convince that using elastic virtual machines is a step forward from its fixed-size counterpart. However, elastic virtual machines alone do not fully minimize resource holes and performance overheads. We then proposed a new direction with cooperative resource management in a IaaS. This so-called cooperative IaaS extensively uses the dynamic capability (or elasticity) of virtual machines, along with resource management awareness from both actors in the cloud, to achieve better performance and power save. The communication between the two actors is transmitted by a bi-directional communication channel. The cooperative IaaS stands between a traditional IaaS and a PaaS in terms of level of controls and level of abstractions.

In the next chapter, we study other existing research works having the same goal addressed in this thesis: achieving further optimization of performance and power consumption. This is to better compare our approach with current works in the research community, in order to have a clear picture of our work in the literature.

Chapter 4

State of the Art

Contents

4.1 Resource Management with Static Virtual Machines . . 45

4.1.1 Traditional Resource Management 45

4.1.1.1 The Customer Side 45

4.1.1.2 The Provider Side 48

4.1.2 Multi-Level Resource Management 51

4.1.2.1 Uncoordinated Policies 52

4.1.2.2 Cooperative Policies 53

4.2 Elastic Virtual Machines 56

4.3 Synthesis 58

This thesis emphasizes on the improvement of performance and reduction of power consumption in cloud infrastructure, with the combination of a **cooperative resource management policy** and **elastic virtual machines**. We proposed our approach to use this combination in the previous chapter: we attempt to exploit the benefit of splitting and merging virtual machines. We also reviewed our direction with several points of view (IaaS extension, hybrid PaaS-IaaS and distributed virtual machine) so that we have better perspectives for our research work.

This chapter gives an overview of the related work from the research community's perspectives, mainly on two sides: resource management with fixed size virtual machines and the research direction of elastic virtual machines. According to our work's characteristics that we described in the previous chapter, we consider each related work as a resource management system with the following criteria:

- **Server Consolidation**: the capability to ensure that physical servers are well used, resulting in energy saving for the provider.
- **Horizontal Scaling**: the ability to add or remove application instances to deal with fluctuation of workload, ensuring application performance and reducing cost for the customer.
- **Vertical Scaling**: the ability to dynamically change virtual machine's size at runtime, further ensuring application performance and minimizing cost for the customer.
- **Cooperation**: the collaboration between the two actors of the cloud to achieve both better performance and power saving. Note that this cooperation does is limited to splits and merges of virtual machines.

4.1 Resource Management with Static Virtual Machines

The first category of work we consider in the literature is cloud resource management policies with fixed size virtual machines. Most research works do not consider the IaaS allowing customers to resize their virtual machines in running state. This fact is a limitation of IaaS providers: the customers are not provided with elastic virtual machines.

4.1.1 Traditional Resource Management

4.1.1.1 The Customer Side

Most customer's side resource management policies are on-demand resource provisioning and allocation [23]. Replication is currently the most widely used method for scaling the application according to the runtime load. Research work in this direction focuses on monitoring, allocating and distributing the customer's application one the provided virtual machines [25]. Algorithms are proposed to have better prediction of runtime workload [35, 47] and better allocation of virtual machines [24, 45, 39]. This direction has two main goals:

- **Performance**. Ensuring that the application has enough computing power for maintaining its performance is one of the most critical factors affecting the resource management policy on the customer side.

- **Minimize VM Usage**. This goal is to minimize cost for the customer. The less allocated and unused resources, the less the customer has to pay.

Automated service provisioning is where many research works concentrate on. It allows the customer to reach the two above goals [56]. The resource management policy on the customer side needs to take into account various aspects during this process. For example, the application manager may need to consider virtual machine boot time, because during boot time of a new virtual machine, another virtual machine may be released and be ready for another use. Therefore, an allocation for a new virtual machine is wasted in this case.

Authors at Florida International University and Delft University of Technology, with their work [51] in 2012, focused on **allocating** virtual machines and **deploying** application instances on those virtual machines in a cloud. This work concentrates on **jointly** considering both policies (allocating and deploying) at the same time, in the customer's resource management task and evaluating them using their system called **SkyMark**. The first policy is explained as the task for acquiring or releasing the resources based on the actual needs, only when, where, and for how long they are needed. The resource can be allocated according to the requirements of the workload. The later specifies which application instance will be running on which virtual machine in the pool of allocated resources. This policy needs to take into account delays during the allocation process, issued by the IaaS provider, including time to select, lease-and-boot, and release a VM instance.

The authors described eight virtual machine allocation policies, including both static and dynamic ones. Most of the dynamic policies are based on a generic *On-Demand* one: a new virtual machine is requested for each job that cannot be assigned to a previously-allocated resource. Additionally, four policies for application instance placements are described, mostly based on the traditional *First-Come, First-Served* policy. Throughout the evaluations, the authors claimed that these policies achieve better performance and cost to the customer.

One main contribution of this work is to have a clear separation of resource management tasks on the customer side: provision and allocation. The experiments, partly performed on EC2, evaluate the effectiveness of each pair of provision and allocation policy. The separation of resource management tasks shows that the customer should take into account his workload characteristics to achieve better per-

formance and cost efficiency. The authors claimed that informed allocation policies achieve better performance than uninformed counterparts. To summarize:

- **Server Consolidation**: relies on the cloud provider.
- **Horizontal Scaling**: both static and dynamic allocation of application instances depending on the experiment scenario.
- **Vertical Scaling**: only consider fixed size virtual machine.
- **Cooperation**: single-sided (customer) resource management. As a result, there is no cooperation proposed between the customer and the provider.

Another work [30], by authors at A*STAR Institute of High Performance Computing in 2011, proposes an extensible framework for On-Demand provisioning and adaptation in a IaaS infrastructure. The proposed framework is named **CREATE** (**C**loud **RE**source provisioning and **A**dap**T**ation fram**E**work). The authors proposed a set of algorithms for resource adaptation, dealing with informed provisioning decisions to adapt with various types of workload. The framework prototype is implemented as a web-service, allowing users to interact with different public cloud providers as well as local resource managers (or private clouds). The objective of the framework is to evaluate the proposed algorithms for managing provided resource sets (RSSs).

The framework is composed of various components for easy flexibility and extensibility. The main components include: RSS Monitoring and Management Service (RMM), RSSs Adaptation Service (RA) and Cloud Clustering Service (CC). At the lowest level, RMM is responsible for managing the configuration and gathering information regarding the resource and workload of a single RSS. RA is at a higher level of RMM: it manages multiple RSSs at the same time by communicating with the corresponding RMMs. According to the gathered information (resource and workload), RA executes the associated algorithms (6 provided in the paper) to adapt the provisioning decisions. These decisions are then performed by CC. CC is responsible for interacting with the cloud providers or the local resource managers, and act as an abstract layer on top of the cloud providers. Additionally, the algorithms described in [30] take into account various factors for making informed provisioning decisions, including location parameters, resource provisioning overhead, etc.

A noticeable contribution of this work is proposed in the CC: an abstraction layer to interact with various cloud providers. This layer allows the customer to work with multiple providers without worrying about the complexities and differences among them, and in some ways helps him to be resistant from vendor lock-in. Our implementation of cooperation resource management system has a similar structure:

intermediate components for interacting with cloud providers. Additionally, the authors should have also considered elastic virtual machines in their CC layer.

To conclude about CREATE with our mentioned characteristics:

- **Server Consolidation**: relies on the cloud provider.
- **Horizontal Scaling**: CREATE supports dynamic allocation of application instances based on the OnDemand policy.
- **Vertical Scaling**: CREATE does not use elastic virtual machines for vertically scaling up application instances.
- **Cooperation**: similar to SkyMark, CREATE is a single-sided resource management system. There is no cooperation between the customer and provider.

4.1.1.2 The Provider Side

In contrast with the customer side, research work on the provider side mostly focuses on (1) **size of resource slices**, i.e. provided virtual machine's size; or (2) **virtual machine placement**, i.e. allocation and migration of virtual machines among physical servers to improve infrastructure utilization ratio (note that this section only considers static-size virtual machines). Various algorithms are proposed to solve the virtual machine packing problem [20, 43], taking into account various factors like real resource usage, virtual machine loads, etc. Because of these factors, the process of making a migration plan may take a long time. There exists also prototypes of the multiple-level resource managers for the IaaS, which will be discussed later in section 4.1.2.

Resource Granularity: Small or Large Pieces of Resource

Many large IaaS (Google Compute Engine [5], Amazon EC2 [1], Windows Azure [12], HP Public Cloud [7], etc) offer various types of virtual machines (called instance types). Each type has a set of predefined characteristics: number of CPU cores, amount of memory, size of storage, network bandwidth, etc. The customer needs to select the best type to fit with his resource requirement before each virtual machine allocation. From the provider's point of view, his virtual machine packing algorithms have all predefined values (resource sizes) in all resource dimensions (CPU, memory, storage, network). As a result, offering resources with instance types poses less complexity in resource management tasks for the provider. The optimization algorithms for placing and migrating fixed-size virtual machines with instance types are therefore *less complicated* and *executes faster*.

Recent observations of IaaS trends by the authors at the Israel Institute of Technology in 2012 found that the model to provide fixed instance types will eventually change to flexible ones [19]. The authors called this model *Resource-as-a-Service*. **CloudSigma**[1] or **DimensionData**[2] are two popular provider examples of this trend. Instead of providing fixed-size instances, the customer can choose the desired resource size for his virtual machines. The flexibility in specifications includes how many CPU cores, how much memory, how big is the storage, etc. that the customer wants. The customer, after choosing the required virtual machine characteristics (which is not bound to any instance types), has an exact price per hour that he will need to pay for the desired virtual machine. This method brings more flexibility to the customer when choosing the desired resources than a traditional IaaS.

Note that, this flexibility allows the customer to specify their requirements at allocation time and not at runtime. This is one step in *the transition between fixed-size virtual machines and elastic virtual machines*. Most current IaaS providers use virtual machine as a scalability unit: they currently do not support elastic virtual machines (which can be resized at runtime), limiting the customer from dynamically resizing their virtual machines on the fly according to their needs [27]. IaaS providers should take one step further by offering fully elastic virtual machines to customers.

We analyze CloudSigma's and DimensionData's characteristics as follows.

- **Horizontal Scaling**: being the IaaS providers, CloudSigma and DimensionData do not prevent the customer from scaling his application.
- **Vertical Scaling**: Elastic virtual machine is not allowed in these IaaS. Customers can specify their virtual machine size at the beginning of resource allocation but not at runtime.
- **Cooperation**: These two providers have a set of IaaS APIs provided to the customer. However, there is still no cooperation similar to our proposal, especially upcalls.

Server Consolidation with Virtual Machine Migration

As previously discussed in section 2.3.1, the provider dynamically migrates his virtual machines into as few physical servers as possible, as long as the SLA is satisfied. Various algorithms are proposed for having the best virtual machine placement with different constraints. Actual CPU load and memory usage are among the most important ones in these algorithms.

[1] http://cloudsigma.com
[2] http://dimensiondata.com

Recent work by authors at University of Southern California in 2013 [37] considers server consolidation as a *Bin Packing Problem* (BPP). This is a complicated problem with multi-dimensional bin packing vectors (migration time, migration cost, failure, virtual machine sizes, resource types, placement constraints, etc.), and is a NP-Hard problem. Several heuristic techniques are proposed to reduce the time of execution to find the best possible result as migration plan.

FINAL [37] proposes a two-level consolidation system. This system consists of two distinct resource managers: the *global* and *local* managers. Due to the nature of a hosting centers that consist of a set of connected clusters, the *global* manager assigns the virtual machines to a cluster first. The main objective of this work is to minimize the total resource usage of cluster, defined as a sum of used resource, as a means to minimize the cost. The *global* manager in this work uses an FFD-based algorithm, namely VM2C (VM-to-cluster), taking into account both unrelated and correlated virtual machines, to select an appropriate cluster to assign the virtual machines. The *local* resource manager then deploys the assigned virtual machines on physical servers in the cluster. The authors proposed an algorithm (called VM2PM, or VM-to-PM) to achieve balanced resource allocation: the *local* manager deploys the assigned virtual machine into a physical server for which the remaining resource is the most similar with the being-considered virtual machine. This algorithm is basically the same as the traditional *Best Fit Decreasing* (BFD) solution. FINAL is described as the combination of VM2C and VM2PM.

This work clearly divides resource management tasks into two level: data-center level and cluster level. By dividing into two subproblems, the author achieved better time and cost for the virtual machine placement task. Like previous work, the author should also take into consideration elastic virtual machines. To summarize, FINAL [37] has the following characteristics:

- **Server Consolidation**: Taking into account various input constraints, consider server consolidation as the main goal. Additionally, FINAL is a two-level consolidation system.
- **Horizontal Scaling** is not considered in these work.
- **Vertical Scaling** is also not considered, as they focus on computing virtual machine's migration plans on the provider side with fixed size virtual machines.
- **Cooperation** is not taken into account (only server consolidation).

Another notable work related to server consolidation service for a IaaS provider is described in [32] by the authors in Grenoble Informatics Laboratory and École Mines de Nantes. Based on *Entropy* [36], this work extends the elasticity of the resource manager. Entropy itself is a cluster resource manager, handling all the

deployment and migration decisions in a cluster. It takes a set of input configurations (infrastructure, virtual machine sizes, physical server sizes, load and memory capacity of each, etc) and other constraints from the administrators to minimize the cloud's energy consumption and computes a migration plan that satisfies all provided constraints. This process can be executed periodically or by request. Because of its reliance on a constraint solver, Entropy usually takes time to solve the virtual machine packing and migrating problem, especially with a large number of virtual machines and physical nodes.

The authors in [32] deal with this problem and increase the elasticity of Entropy by organizing a two-level Entropy system, with an Entropy Server and a set of Entropy Workers. Each Entropy instance is virtualized, i.e. put in a virtual machine. This work includes partitioning a cluster or a cloud infrastructure into random groups (each is managed by an Entropy Worker), and gradually starting or stopping Entropy Workers to maintain elasticity of the consolidation system. The authors claim that this two-layer Entropy architecture brings a significant performance gain and less error rates over traditional static single/multiple Entropy systems.

One strong point of this work is the ability to deal with different numbers of virtual machines by dynamically adding or removing Entropy instances. This philosophy is similar to dynamic application sizing at the application level. The author should also consider using elastic virtual machines for Entropy instances to further improve elasticity and efficiency of the workers for consolidation service .

The following characteristics can be inferred from this work:

- **Server Consolidation** is considered as the main goal. Elastic Entropy ensures that the consolidation service is provided with enough computing capability to compute the migration plan with less errors.
- **Horizontal Scaling** is not considered in this work, because this system acts as a consolidation service on the provider side.
- **Vertical Scaling** is also not taken into account: only fixed size virtual machines is supported.
- **Cooperation** is not considered. Elastic Entropy is a single-sided resource management system.

4.1.2 Multi-Level Resource Management

In a real world scenario, both actors naturally implement their own resource management systems to reach their goals of minimizing operation cost (virtual ma-

chine and server costs). As previously discussed, multi-level resource management systems have better performance and less wasted resources. This section investigates existing research works in this direction.

4.1.2.1 Uncoordinated Policies

This situation most commonly happens in current IaaS clouds. In this scenario, both the customer and the provider have their own resource management systems, but they do not work cooperatively. The work in the literature in this direction does not expose the close relationship between these two actors.

Researchers at National Technical University of Athens and Bell Laboratories examine a slightly coordinated (although not fully cooperative) resource management policy in a cloud infrastructure in their work [41]. The authors describe a model to coordinate different resource management policies from both cloud actors' point of views, in order to provide an elastic cloud with optimum allocation for distributed services on virtualized resources. The optimization model takes into account various factors like trust, eco-efficiency and costs. The proposed approach allows the customer to specify his resource management constraints. These constraints include computing capacity, load thresholds for each host and for each subnet before an allocation of a new virtual machine, etc. The authors also describe a set of affinity rules for constraining virtual machine's collocation in the IaaS: a component or a service must be located in the same physical machine or same subnet. The authors claimed that this model allows an efficient allocation of services on virtualized resources.

One important difference compared to other works above is that this research work allows the customer to provide virtual machine collocation constraints to the provider, which is a form of knowledge sharing. However this type of knowledge sharing is done in one direction (from the customer to the provider). The author should have investigated knowledge sharing in the opposite direction. On the other hand, to have a better resource allocation, this work should also consider elastic virtual machines on the provider side when growing and shrinking the application. We summarize this work according to our criteria as follows:

- **Server Consolidation** is not explained. The authors do not mention about virtual machine migration or server consolidation on the provider side.
- **Horizontal Scaling** is done at the customer size with application's replicas.
- **Vertical Scaling** is not used, only fixed-size virtual machines.
- **Cooperation** is slightly more detailed than previous works with affinity rules.

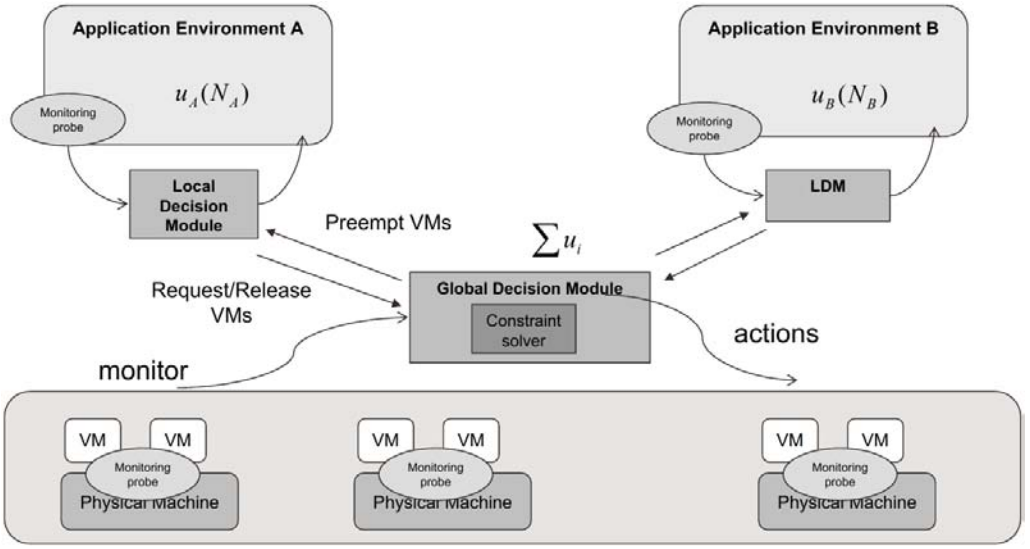

Figure 4.1: Relationship between Local Decision Module and Global Decision Module [44]

4.1.2.2 Cooperative Policies

A cooperative policy, like previously described, takes both actors into account while doing resource management tasks. This behavior insists on the knowledge sharing between these actors in order to have a mutual benefit. Note that the related work about cooperative resource management policies in this section is still with fixed-size virtual machines. Dynamic-size (or elastic) virtual machine will be discussed in section 4.2.

Researchers at University of Nantes described their research work closely-related to ours about knowledge sharing in two level resource management in [44]. The authors proposed an autonomic resource management system to tackle with the requirements of dynamic provisioning and placement of virtual machines, taking both application level SLA and resource cost into account; and to support various types of applications and workloads. Globally, the authors clearly separate two levels of resource management: Local Decision Module and the Global Decision Module (similar to our *AppManager* and *IaaSManager*, respectively, as described in 3.3.1). This architecture is illustrated on figure 4.1.

The *Local Decision Module* is responsible for managing a single *application environment* (AE). Each LDM is application-specific and is associated with each AE at runtime. According to the metrics gathered by the monitoring probes, the LDM evaluates the current system state and makes decision to allocate or release virtual machines, using predefined *utility* functions. These functions (1) transform system load and other metrics to indicators (so-called utility values) and (2) transform re-

source capability (based on the availably resource, provided by the GDM) to utility values. Comparing these values, the LDM makes a decision to add or to remove virtual machines to scale the AE accordingly. This behavior is the same as our defined *AppManager*, which works at the customer level, that we described in 3.3.1.

On the other hand, the *Global Decision Module* manages virtual machine allocation: VM Provisioning (deploying AE onto virtual machines) and VM Packing (placing virtual machines onto physical servers). The authors consider these mapping phases as two *Constraint Satisfaction Problems*. The author solves these problems by using a *Constraint Solver*, with a predefined set of constraints. For example, the first problem (VM provisioning) has a set of requirements for the solver: the acceptable virtual machine sizes (fixed ones, taken from a set of available instance types), the maximum number of virtual machines for a specific AE, etc. Similar to our work, the goal of the VM packing problem in this work is to minimize the number of active physical servers with minimum migration cost.

These two decision modules works cooperatively with each other. The LDM makes requests to the GDM to allocate and deallocate virtual machines. The GDM then notifies the LDMs about virtual machines which have been allocated or deallocated. Similar to previous works, the authors should have also considered elastic virtual machines. Using them increases the number of cooperation tasks that can be performed.

This two-level resource management work has the following characteristics:

- **Server Consolidation** is explained as the packing of virtual machines with various resource constraints. The selected virtual machines are then migrated following an optimal migration plan.
- **Horizontal Scaling** is done at the customer size with application's replicas.
- **Vertical Scaling** is not used. This system considers fixed-size virtual machines only.
- **Cooperation** is applied but still basic.

Researchers at Chinese Academy of Sciences, Wayne State University and others proposed their work in 2013 named **PhoenixCloud** [55], a cooperative resource provisioning system fairly close with ours. This work's objective is to minimize hardware cost. The authors argue that the server cost is the largest share of data center cost, and most of current research work in the direction to reduce data center cost is categorized into two types: server consolidation with virtualization, and statistical multiplexing to predict future loads in order to be able to delay allocations. The main idea of this work is to share the same used resource among a number of

applications using statistical data to prioritize given applications in case of load peaks. The authors also claim that IaaS providers need to take into account the increasing heterogeneous workloads in terms of different classes of workload.

The idea of this work is to leverage different characteristics of heterogeneous workloads and use the knowledge about them to improve statistical multiplexing in the cloud. Each type of workload has its own attributes, such as runtime for each task in the work load, task's priority, etc. If the provider can exploit their complementarity, he can have a better utilization ratio and reduce hardware cost. The main contribution of this work is to take into account the workload characteristics (provided by the customer), and to reschedule (or replan) these workloads to decrease peak resource consumption. Typically, web server has more priority than queued data analysis jobs: each web request must be responded immediately. In a case of peak load, the submitted data analysis jobs can be delayed, so that there are more available resources for the web server to satisfy its workload. As a result, the authors the IaaS resource pool can be smaller when compared to a traditional, uncooperative IaaS. This helps to reduce the hardware cost for the IaaS provider.

The system is composed of two main entities: the *Common Service Framework* and the *Workload Manager*. The CSF is responsible for monitoring and managing resources in the cloud infrastructure, similar to our *IaaSManager*. The workload profilers handle incoming requests, schedule the jobs and manage application instances. Its function is the same as our *AppManager*. From this point of view, PhoenixCloud is a **cooperative, two-level** resource management, similar to our approach.

One important aspect of PhoenixCloud is the need of delaying submitted jobs. This behavior would need the agreement of the customer. Cooperation can only be leveraged when two actors see the benefits. We assume that in order to allow delaying tasks, the customer has lower price for their resource allocation.

We conclude the characteristics of **PhoenixCloud** before switching to the Elastic Virtual Machine category as follows.

- **Server Consolidation**. PhoenixCloud does not consider the migrations of virtual machines to reduce power consumption. It focuses on rescheduling workloads of the customers.
- **Horizontal Scaling** is based on the replication of the customer's applications.
- **Vertical Scaling** is not applied for PhoenixCloud: it uses fixed-size virtual machines for the customer's application replicas.

– **Cooperation** is explained as the sharing of workload knowledge between the customer and the provider.

4.2 Elastic Virtual Machines

The problems of fixed-size virtual machines have been discussed in section 3.1. This section describes and compares our work with other approaches regarding the usage of elastic virtual machines in cloud infrastructures. As previously discussed in section 3.2, elastic virtual machines bring benefits not only to the customer (fine-grained resource, less performance overhead and more responsiveness) but also to the provider (less number of virtual machine instances, migrations and less fragmentation). This is usually referred as vertical scaling (or *scaling up*, in contrast with *scaling out* - the action to add more replicas in replicated systems to deal with load peaks).

Research work [28] (by researchers at Potsdam University in 2011) is one of the pioneer works in the application of elastic virtual machines in the cloud. The authors propose to use elastic virtual machines as a means of scaling the customer's application. By using elastic virtual machines, resource management in a cloud is more fine-grained than adding or removing application instances in fixed-size virtual machines (horizontal scaling). In this work, the author claims that scaling up has more benefits than scaling out: having a wider range of supported applications (can be applied to any tier), better stability, mitigates SLA violations, no overhead of booting virtual machine, lower cost and complexity, and most importantly, higher throughput with less performance overhead. Experiments with elastic virtual machines in a typical multi-tier application, including a web server tier and a database tier, confirm this benefit.

Although the resource management policy is not detailed in the original paper, we classify this work as an uncoordinated two-level resource management with elastic virtual machines, because there can be server consolidation at the IaaS level. On the customer side, the resource management policy is implemented as a basic *On-Demand* one. According to the system load and response time, the *AppManager* requests to add or reduce resources to the virtual machine. Upon receiving these requests, the *IaaSManager* vertically resizes the virtual machine, by adding or removing the associated vCPUs.

While showing benefits of elastic virtual machines, this work solely focuses on vertical scaling. Scaling up is limited because servers do not have unlimited re-

source. The authors mentioned this problem in the paper. They also confirmed that this should be better to support both scaling out and scaling up in order to have both fine grain resource allocation and unlimited amount of resources. Finally, the resource management policies in this work are uncoordinated: the *IaaSManager* simply resizes the virtual machines.

- **Server Consolidation**. This work does not migrate virtual machines to ensure server consolidation.
- **Horizontal Scaling** is not considered.
- **Vertical Scaling** is the main focus of this work.
- **Cooperation** is not applied for better performance and power save.

Another recent work [54] in 2013 by researchers at TU-Dresden uses a different approach for autonomic resource allocation in a cloud. The authors provide **VScaler**, a framework using reinforcement learning (RL) to adapt the resource allocation policies with its observation from the environment (i.e. the cloud infrastructure). VScaler scales the application both horizontally and vertically (using tier replication and elastic virtual machine, respectively). The authors consider the adaptive policy for resource management as a Markov Decision Process with states, actions, transaction probabilities and rewards. Additionally, the authors use parallel learning with assumption to speed up the learning process.

The main difference of VScaler with our work is the single-sided resource management policy. VScaler focuses only on scaling the application while it *does not have any knowledge about the application* being managed. It considers the application as a blackbox and captures its workload and other metrics at runtime, by using a built-in proxy. Based on the gathered information, VScaler makes scaling decision, either horizontally (replicating application instances) or vertically (adding resources for an existing virtual machine of this application). From this point of view, this system is working on the *provider side* only. As argued previously in the previous chapter, the author should implement cooperation from both side optimizes resource usage and application performance.

- **Server Consolidation**. The author does not mention about server consolidation, only focuses on scaling customer's application.
- **Horizontal Scaling** is considered by using tier replication.
- **Vertical Scaling** is used with elastic virtual machines.
- **Cooperation** is not also not mentioned because VScaler works on the provider side only.

A more matured work is described in [31] by researchers at IBM Research India in 2012. The authors propose **SmartScale**, taking into account both horizontal and vertical scaling in its resource management policy, i.e. by changing the number of virtual machines and changing the resources assigned to the allocated virtual machines. The authors claim that SmartScale reduces the total cost of running virtual machines and outperforms both traditional vertical and horizontal scaling policies. The combination of these policies brings a challenging problem that SmartScale must solve at runtime: determining the optimal scaling path and the trade-off of the corresponding reconfiguration. The authors solve this problem with a two-phase algorithm: computing the optimal virtual machine sizes and the corresponding application's throughput; and determine the minimum number of instances such that the total throughput is reached. Note that SmartScale's scale algorithm also takes into account the information about the application provided by the customer.

From our point of view, SmartScale should also take into account the information (tier knowledge) provided by the customer to have a better placement. This work focuses on application scaling, taking into account reconfiguration cost, and does not consider server consolidation with virtual machine migration. With these additional actions, SmartScale can also support splitting and merging virtual machine, further improving resource management at both sides.

Before summarizing the related works, we conclude the characteristics of SmartScale:

- **Server Consolidation**. SmartScale does not optimize virtual machine's placement.
- **Horizontal Scaling** is used with tier replicas.
- **Vertical Scaling** is also used with elastic virtual machine.
- **Cooperation** is slight, because SmartScale takes the information about the application as an input parameter for its optimal scaling algorithm

4.3 Synthesis

This chapter summaries the study of related works regarding resource management with both fixed-size and elastic virtual machines in the cloud infrastructure (table 4.1). From this table, we realized that there are very few research works considering **multiple level resource management**. Many research works focus on resource management at only one level. On the provider side, they typically concentrate on the placement and migration plan of virtual machines at runtime. On the customer side, virtual machine provision and allocation/deallocation algorithm

	Consolidation	Scaling		Cooperation
		Horizontal	Vertical	
SkyMark [51]	No	Static, Dynamic	No	No
CREATE [30]	No	Yes, OnDemand	No	No
CloudSigma DimensionData	Not mentioned	Not limited	Only sized at creation	No
FINAL [37]	Two-Level, VM2C and VM2PM	No	No	No
Elastic Entropy [32]	Two-Level Consolidation	No	No	No
[50]	Migration	Yes	No	No
[41]	Not mentioned	No	No	Affinity rules
[44]	Migration (constraints)	Replicas	No	Basic
PhoenixCloud [55]	No	Replicas	No	Workload Knowledge
[28]	No	No	Yes	No
VScaler [54]	No	Replicas	Yes	No
SmartScale [31]	No	Replicas	Yes	Application Information for Scaling Algorithm

Table 4.1: Comparison of the Policies

are the main topic of research. On-Demand and its derivatives are the most widely used resource management policies at this level.

Few recent researches exploited the **elasticity of virtual machine** in the cloud. This is because most large IaaS providers, if not all, do not permit the customer to resize their allocated virtual machines on the fly. These works show the benefit of dynamic-size virtual machines over traditional fixed-size ones, mainly in terms of speed and management cost. No existing work takes into account the full cooperation of the two cloud actors in combination with elastic virtual machine.

From the general orientation that we described in section 3.3 and this analysis, our work takes the best factors of two areas, namely cooperative two-level resource management policy with elastic virtual machine, to achieve the best possible performance to the customer and power economy to the provider. The detailed description of our work will be described in the next chapter.

Chapter 5

Contribution: Cooperation Design

Contents

5.1 Cooperation Protocol 61
5.1.1 Tier Subscription 61
5.1.2 Changing Tier Resource 61
5.1.3 Splitting or Merging Tier Instances 63
5.2 Quota Management 65
5.3 Cooperation Calls 70
5.3.1 Upcall 71
5.3.1.1 Proposals of Virtual Machine Allocation 71
5.3.1.2 Notifications of Virtual Machine Allocation . . . 72
5.3.1.3 Elasticity Proposals 72
5.3.2 Downcall 74
5.3.2.1 Requests 74
5.3.2.2 Confirmations 75
5.4 Synthesis 75

Our system extensively uses the notion of **cooperation calls** between the two managers (*AppManagers* and *IaaSManager*). This chapter is dedicated to describing the specification of major cooperation calls. Furthermore, we also detail our algorithms to process quota change requests in a cooperative IaaS. We divide the cooperation calls into two main types: **Upcall** and **Downcall**, according to the direction of the call. A downcall is made from the customer to the provider. An up call is in the opposite direction, from the provider to the customer.

5.1 Cooperation Protocol

The cooperation protocol is defined as a sequence of cooperative calls at runtime to achieve a particular goal. During the design of the protocol we identified the following main operations:

- Subscription for application tier;
- Modification of the amount of resource (also called quota) for an application tier, triggered by the *AppManager*; and
- Splitting or merging virtual machines associated to an application tier, triggered by the *IaaSManager*.

This section describes the actions performed by each actor in each operation. An example of these operations is illustrated in figure 5.1.

5.1.1 Tier Subscription

The first operation of the cooperation resource management policy is to share the knowledge of the application tiers. The customer initiates the cooperation with a tier subscription. By subscribing all tiers and providing tier name in subsequent calls, the *AppManager* provides the group notion of its application instances to the *IaaSManager*. Based on the provided tier notion, the latter can perform certain tasks at runtime for further tier and virtual machine placement optimization.

Figure 5.1 shows an example of a tier subscription at time *a*. The *IaaSManager* then confirms this subscription with a confirmation upcall "OK" at time *b*.

5.1.2 Changing Tier Resource

At runtime, based on the actual application needs, the *AppManager* can request to modify resources allocated to a specific application tier, either adding resource, or removing resource. Like previously described in section 3.3, our cooperative resource management policy uses elastic virtual machines at runtime. As a result, there are several possible solutions to a single quota modification request. Based on the actual physical server usage and virtual machine placement, the *IaaSManager* can

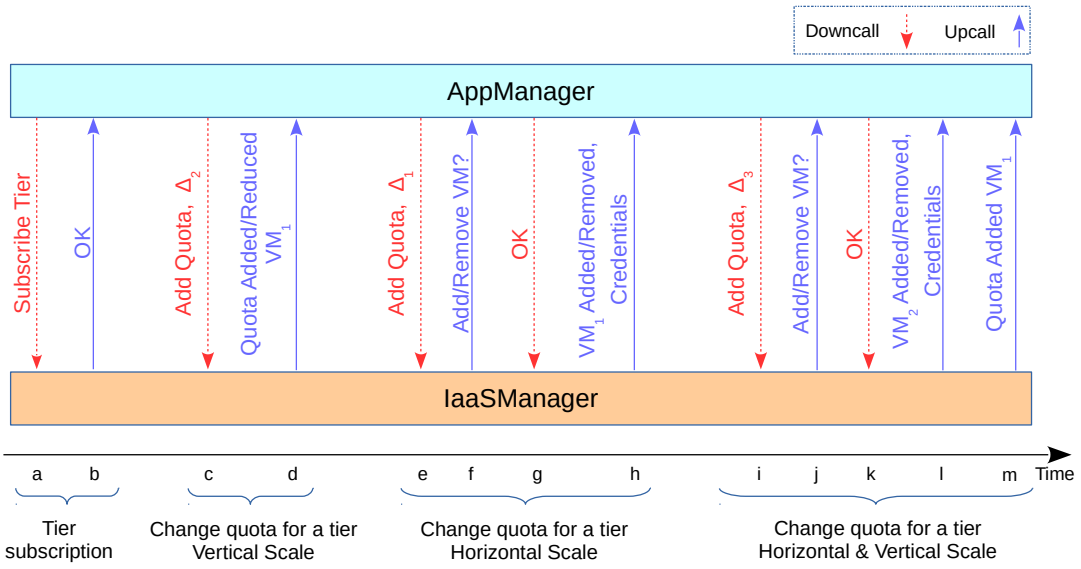

Figure 5.1: Example of Actions for Subscription and Add Resources

perform: (1) scaling the tier horizontally (adding/removing virtual machines); (2) scaling the tier vertically (adding/reducing resource to running virtual machines); or (3) a combination of both (1) and (2). Algorithms for managing application tiers according to a resource modification request will be detailed in the next section (5.2). To illustrate the above sequences of cooperation calls, we use figure 5.1 to show 3 different examples of "add quota" operations at runtime for an application tier.

If the modification request can be solved with a vertical scale, the *IaaSManager* simply resizes one or several running virtual machines and notifies the *AppManager* with an upcall about these scaling actions (figure 5.1, time *d*). Upon receiving these notifications, in case of a multi-tiered application, the *AppManager* reconfigures its balancer to take into account the new weight of its instances on the resized virtual machines.

On the other hand, if that modification request for changing tier resource can be solved with a horizontal scale, the following 3 cooperation calls are performed: a proposal upcall from the *IaaSManager* to allocate/deallocate a virtual machine (time *f*), a confirmation downcall from the *AppManager* (time *g*) and finally an upcall from the *IaaSManager* to notify the *AppManager* about this allocation/deallocation (time *h*). Upon receiving the last notification, if this is an allocation, the *AppManager* has access credentials to deploy its application instance on the newly allocated virtual machine. If this is a multi-tier application, the *AppManager* needs to add this new instance into its load balancer.

Finally, in case of a combination of both horizontal and vertical scaling, the sequence of cooperation calls is the accumulation of the two above sequences: a proposal upcall from the *IaaSManager* to allocate/deallocate a virtual machine (time j), a confirmation downcall from the *AppManager* (time k), a notification upcall about a new virtual machine allocation/deallocation (time l) and another notification upcall about resized virtual machines (time m).

5.1.3 Splitting or Merging Tier Instances

At runtime, if the *IaaSManager* found an opportunity to optimize its physical resource usage or application performance, it can propose to split a big virtual machine of an application tier to two smaller ones (in order to fill resource holes), or propose to merge small virtual machines into a bigger one (in order to reduce virtualization and balancer overhead). Note that, these elastic splits or fusions can be rejected by the *AppManager* depending on the customer's goal. For example, application A may need two application instances of the same tier to act as a solution for fault tolerance, thus merging these two instances is not feasible. Another application B may need big application instances to process data from large database tables, requiring a lot of memory. As a consequence, splitting an instance of application B is not preferred because smaller virtual machines will need to a lot of swap to disk (which is generally slow) to be able to handle such large memory requirement.

An elastic split of a big virtual machine is explained as the following sequence:

- An allocation of a new virtual machine
- A vertical scale down of the original virtual machine

In contrast, an elastic merge between two virtual machines can be explained as a sequence of two actions:

- A deallocation of the first virtual machine
- A vertical scale up of the second virtual machine

Based on the above split and merge description, we then define cooperation calls for splitting and merging virtual machines, along with an example from figure 5.2. For a cooperative split, the following 5 cooperation calls are performed: a split proposal upcall from the *IaaSManager* (time A), a confirmation downcall from the

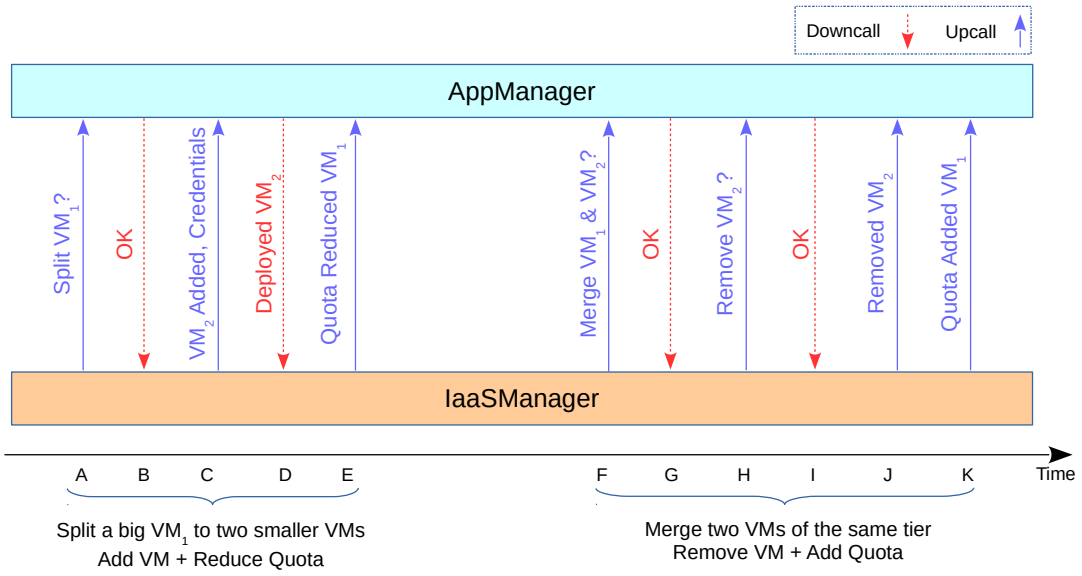

Figure 5.2: Example of Actions for Merging and Splitting Virtual Machines

AppManager (time B), a notification upcall from the *IaaSManager* to notify about a new virtual machine allocation with its access credentials (time C), a confirmation downcall from the *AppManager* that its tier instance has finished deployment (time D), and finally a notification upcall from the *IaaSManager* that the original virtual machine has been scaled down (time E). In case of a multi-tier application, like previously, after performing these calls, the *AppManager* has to reconfigure its load balancer.

On the other hand, a cooperative fusion has the following sequence: a fusion proposal upcall from the *IaaSManager* with the names of two virtual machines to be merged (time F), a confirmation downcall from the *AppManager* (time G), a proposal upcall from the *IaaSManager* to remove either one of the two virtual machines (time H), a confirmation downcall from the *AppManager* after it has finished undeployment on this virtual machine (time I), a notification upcall from the *IaaSManager* about this virtual machine has been deallocated (time J), and finally a notification upcall from the *IaaSManager* about the first virtual machine has been scaled up (time K). The load balancer still needs to be reconfigured in case of a multi-tier application like previously.

With the above procedures, we have described the major operations and sequences in our cooperation protocol. The next section describes our algorithms to effectively manage resources (or quota) allocated to an application tier.

5.2 Quota Management

Due to the group notion, the task of managing virtual machines which contain the whole application tier is shifted from the customer to the provider. While describing our cooperative IaaS approach in section 3.3, we mentioned that the *AppManager* monitors its tier loads and sends requests to change the quota (the size) of the whole tier. On the other side, the *IaaSManager* is responsible for the organization of the virtual machines to fit the required computing power, including the placement and size of each virtual machine. Upon receiving a downcall to change quota from the *AppManager*, according to the actual status of the virtual machine allocation on servers, the *IaaSManager* can have multiple choices to serve this request. This section introduces the algorithm being used in the *IaaSManager* in order to handle such requests.

With the following definitions:

- m: number of machines in the server pool
- $\psi = \{M_j, \quad 0 \leq j < m\}$: the set of running servers
- φ_j: remaining resource on M_j
- n: number of allocated virtual machines for a *tier*
- $\chi = \{V_k, \quad 0 \leq k < n\}$: set of running virtual machines for the current tier
- α_k: amount of allocated resource for V_k

Based on the amount of resources being modified for an existing tier Δ_q (negative value of Δ_q means to reduce quota, and otherwise), we identified *four* possible solutions to deal with an increase quota request:

- **Vertical scale of an existing virtual machine**: the *IaaSManager* can add a specific amount of resource Δ_q from an existing virtual machine V_k: $\alpha_k = \alpha_k + \Delta_q$, such that the server containing V_k is free enough (in terms of available resources) for this vertical scale:

$$\exists k \mid 0 \leq k < n, 0 < \Delta_q \leq \varphi_j, V_k \in M_j \tag{5.1}$$

  The *IaaSManager* can traverse through its internally-managed list of virtual machines to find a possible virtual machine for this action. If not found, it tries the next action (see below).

- **Distribute the required quota change among existing virtual machines**. The *IaaSManager* tries to split the required quota change $\Delta_q > 0$ into $p \leq n$ smaller sub-quota changes:

$$\Delta_q = \sum_{i=0}^{p-1} \delta_i \tag{5.2}$$

such that these sub-quota changes can be applied into a set S consisting of p virtual machines of the same tier:

$$S = \{V_{s_0}, V_{s_2}..., V_{s_{p-1}}\}, S \subset \chi, 0 \leq s_i < n, \forall i \in [0, p-1] \tag{5.3}$$

If it is possible to find such case, the *IaaSManager* then scales them vertically:

$$\alpha_{s_i} = \alpha_{s_i} + \delta_i, \forall i \in [0, p-1] \tag{5.4}$$

to avoid the need of a virtual machine allocation. However, similar to above, the free-resource constraints must be satisfied:

$$\delta_i \leq \varphi_j, V_{s_i} \in M_j, \forall i \in [0, p-1] \tag{5.5}$$

- **Allocation of a new virtual machine**: if $\Delta_q > 0$, and two actions above cannot be performed because of the free-resource constraints (5.1, 5.5), the *IaaSManager* creates a virtual machine V_n with $\alpha_n = \Delta_q$ and asks the *AppManager* to deploy an application instance on it.

- **Combination of the above actions** is the last action in case each single one above cannot be done. For example, the combination of vertical scale of an existing virtual machine V_k: $\alpha_k = \alpha_k + \delta_1$ and an allocation of another virtual machine V_n with $\alpha_n = \delta_2$, such that $\delta_1 + \delta_2 = \Delta_q$. We prioritize vertical scaling of one or several virtual machines to avoid adding virtual machines .

In contrast, a reduce quota request ($\Delta_q < 0$) is easier to handle than above:

- **Deallocations of running virtual machines**: this action has the highest priority. The *IaaSManager* firstly tries to find a virtual machine V_k with $\alpha_k \leq \Delta_q$, and if found, proposes a virtual machine removal to the *AppManager*. It repeats this action until there is not any virtual machines which are small enough to remove.

- **Vertical scale of existing virtual machines**: The *IaaSManager* then tries to reduce resources from the remaining virtual machines allocated to the tier.

Following the above analysis of possible actions to process a quota change request, we implemented the quota modification algorithms in Algorithm 1, 2 and 3.

Algorithm 1: Handling quota modification request of the *IaaSManager*

Input : Tier t
Amount of resources to be changed Δ_q
Set of occupied servers $\psi = \{M_j\}$
Output: Successfully resized tier

```
/*Find the set of running virtual machine for tier χ(t) = {V_k}*/
```
$\chi = \text{findVm}(t)$;
if $\Delta_q > 0$ **then**
  addQuota(t, Δ_q, χ, ψ)
else
  reduceQuota(t, Δ_q, χ, ψ)

Algorithm 2: Reduce quota from a tier

Input : Tier t
Amount of resources to be removed $\Delta_q < 0$
Set of running virtual machines $\chi(t) = \{V_k\}$
Set of occupied servers $\psi = \{M_j\}$
Output: Successfully resized tier

```
/*Sort ascending the list of VM χ by allocated size*/
```
sortVm(χ);
foreach $V_k \in \chi$ **do**
  if $size(V_k) \leq \mid \Delta_q \mid$ **then**
    `/*Less or equal the requested amount, remove this VM*/`
    $\Delta_q = \Delta_q - size(V_k)$;
    notify(t, `ET_UPCALL_REMOVE_VM`, V_k);
    free(t, V_k);
  else
    `/*Larger than the requested amount, a reduction is enough*/`
    notify(t, `ET_UPCALL_QUOTA_CHANGED`, $size(V_k) - \mid \Delta_q \mid$);
    setSize($V_k, size(V_k) - \mid \Delta_q \mid$);
    return;
  `/*Anything left?*/`
  if $\Delta_q = 0$ **then**
    break;

Algorithm 3: Adding quota to a tier

Input : Tier t
Amount of resources to be added $\Delta_q > 0$
Set of running virtual machines $\chi = \{V_k\}$
Set of occupied servers $\psi = \{M_j\}$
Output: Successfully resized tier

```
/*Sort ascending the list of PM χ by free size*/
sortPmFree(ψ);
/*Try to scale up one VM in a PM with the least free resource*/
foreach V_k ∈ χ do
    M_j = findPm(ψ, V_k);
    if freeRes(M_j) ≥ Δ_q then
        setSize(V_k, size(V_k) + Δ_q);
        notify(t, ET_UPCALL_QUOTA_CHANGED, Δ_q);
        return;

/*Try to distribute quota among the allocated VMs*/
free = calcTotalPmFreeSize(t, ψ);
if free ≥ Δ_q then
    q_added = 0;
    while q_added < Δ_q do
        (V_k, δ_q) = calcSubQuota(t, χ, Δ_q, q_added);
        if δ_q > 0 then
            setSize(V_k, size(V_k) + δ_q);
            q_added = q_added + δ_q;
        else
            break;
    notify(t, ET_UPCALL_QUOTA_CHANGED, q_added);
    if q_added = Δ_q then
        return;
    else
        Δ_q = Δ_q − q_added;

/*Scaling up is not enough, need to scale out*/
r = min(Δ_q, MaxVmSize);
n = allocate(t, r);
notify(t, ET_UPCALL_ADD_VM, n);
if Δ_q > r then /* Add recursively the remaining quota */
    addQuota(t, Δ_q − r, χ, ψ);
```

With the described algorithms, we try to minimize the number of virtual machine allocations and deallocations by attempting to distribute the quota changes to the running virtual machines. We aim to exploit the elasticity of virtual machines in a cooperative IaaS, because allocating and deallocating virtual machines are costly in terms of time and performance. We also use a combination of both vertical scaling and horizontal scaling when vertical scaling only is not enough to handle quota modification requests.

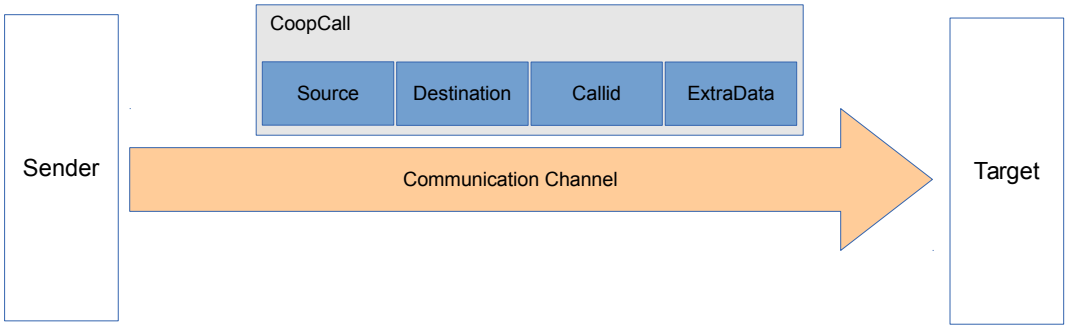

Figure 5.3: Detail of a Cooperation Call

5.3 Cooperation Calls

Previous sections described the protocol and algorithms for resource management in a cooperative IaaS. In this section we detail the specification of cooperation calls being used in our cooperative IaaS.

A typical cooperation call is embedded inside an *event* (events will be detailed in section 6.1). A cooperation call includes four main attributes: the call id, the sender, the target, and extra information (figure 5.3). All of these fields will be serialized and sent through the communication channel. At the destination, data will be deserialized, parsed and finally processed.

Firstly, the *call id* field identifies the purpose of this call. For example, a call with *call id* `ET_DOWNCALL_REGISTER_TIER` is to register a tier from the customer to the IaaS: this call provides the architecture of the application to the IaaS. Another example of *call id* is `ET_UPCALL_MERGE_VM`, used by the provider to propose a fusion of two virtual machines into a bigger one to the customer. The second important data of a call is the *sender* and the *target*. Each managed application component in our cooperative resource manager is wrapped inside a specific component called *Wrapper*, which will be detailed in the next chapter. The sender and target field inside the call are the names of the source wrapper and the destination wrapper, respectively. Finally, the *extra information* field stores the additional data being attached to the call. Some examples of this information include the name of the tier being registered, the amount of quota needed to be added or removed, the name or IP address of the newly allocated virtual machine, etc. This field is optional, but most calls use this field to provide supplemental data to the target.

Cooperation calls can be also categorized from another point of view as **service calls**. This point of view has 3 kinds of calls: requests, responses and notifications. A *request* call is made when a manager (*AppManager* or *IaaSManager*) invokes a

service from the other (ET_DOWNCALL_ADD_VM to create a new virtual machine for example). The *response* will be wrapped in another cooperative call (e.g. ET_OK or ET_ERROR). A *notification* call is made when an actor announces an action or proposal to the other (e.g. ET_UPCALL_MERGE_VM). Note that this is another point of view for cooperation call: an Upcall or a Downcall can be either *request*, *response* or *notification*.

5.3.1 Upcall

An **Upcall** is defined as an event passed through the communication channel: the sender is the *IaaSManager*, and the target is the *AppManager*. The upcall is mostly used for the *IaaSManager*'s proposals. Like discussed in section 5.1, we divide upcalls into three main subcategories: virtual machine allocation proposals, virtual machine allocation notifications, and proposals for splitting or merging virtual machines. This section roughly describes the usage and procedures of the major upcalls used in our system.

5.3.1.1 Proposals of Virtual Machine Allocation

The proposal calls for virtual machine allocation are sent to the *AppManager* after the *IaaSManager* receives a quota change request. Based on the actual allocation and size of the running virtual machines belonging to the tier, the *IaaSManager* can decide either to change quota of one or several existing virtual machines or to allocate/deallocate some more. This sub-category describes the allocation/deallocation proposal.

ET_UPCALL_ADD_VM is used when the *IaaSManager* proposes new allocation of a virtual machine to the *AppManager*, according to the quota change request from the latter. After the confirmation (either accept or reject the allocation with ET_OK or ET_ERROR) the *AppManager* is then responsible for deploying its application instances onto the newly allocated virtual machine. The *ExtraData* field contains the necessary information for the *AppManager* to access the virtual machine, including its name, resource specification, IP address and access credentials. Having these informations, the customer has full responsibility to manage his own application on this virtual machine.

ET_UPCALL_REMOVE_VM is similar to the previous one, but with the opposite purpose: to propose a removal of virtual machine from the allocated virtual machine

list (i.e. to deallocate an existing virtual machine). This proposal is to notify the *AppManager* that its running instance on this virtual machine should be removed from the tier. The *AppManager*, after confirming its decision, detaches the running instance from the load balancer, waits for its finalization, stops it, clears the deployment and notifies the *IaaSManager* that the tier removal has been finished. This notification allows the later to safely terminate a virtual machine without losing any requests (if tier replica is smoothly terminated). In this case, the cooperation improves minimizes request loss during an instance removal with the confirmation and notifications from both sides.

These two cooperation calls were previously illustrated on figure 5.1, time *f* and *j*.

5.3.1.2 Notifications of Virtual Machine Allocation

This sub-category contains the notification calls from the *IaaSManager*, to inform the *AppManager* about the allocation/deallocation status of a virtual machine, after this progress is finished.

`ET_UPCALL_VM_ADDED` is sent to the customer's *AppManager* whenever a new virtual machine is added. This cooperation call is in part of the allocation proposal series, starting with `ET_UPCALL_ADD_VM` in the previous section. Upon receiving this call, the customer has all the necessary information to access the newly allocated virtual machine to deploy his application instance, and finally add the tier to the balancer when the deployment is finished. This call was illustrated on figure 5.2, time *C*.

`ET_UPCALL_VM_REMOVED` is generated after a virtual machine has just been deallocated (i.e. terminated). This call is to notify the *AppManager* that the tier removal process has been completed and was illustrated on figure 5.2, time *J*.

5.3.1.3 Elasticity Proposals

This sub-category of cooperation call is special when compared with a traditional IaaS. Like we previously compared in chapter 4, the ability to split or to merge virtual machine is unique to our system. Other existing works do not take into account cooperative resource management with elastic virtual machine, thus are unable to propose split or merge actions to the customer. A *merge* is proposed to the customer when two (or more) virtual machines of the same application tier are migrated,

collocated and sharing the same physical machine. This merge brings benefit to the customer in terms of performance. On the other hand, a *split* is proposed to the customer when the *IaaSManager* consolidates his cloud infrastructure to reduce power consumption by turning off a physical server. The *split* and *merge* behaviors can be considered as both vertical and horizontal scaling at the same time.

`ET_UPCALL_SPLIT_VM` is proposed to the customer when the *IaaSManager* calculates a better virtual machine placement for server usage optimization. Basically, a virtual machine split is a combination of a vertical scale of an existing virtual machine with a horizontal scale of a new one. As a result, the procedure of this cooperation call is similar to `ET_UPCALL_ADD_VM` (section 5.3.1.1): a new virtual machine will be added, the *AppManager* deploys one more application instance and links it to the load balancer after finishing deployment. An additional task that must be performed is the reconfiguration of the load balancer to take into account the new size of the scaled-down virtual machine.

After the scale-down of the original virtual machine and the creation of a smaller one, the *IaaSManager* has more opportunities to reorganize its allocated virtual machines by migrating them across the server pool to achieve a better hardware utilization ratio. Because split is important to cooperative resource management, we summarize the procedure of the split behavior as follows.

- The *IaaSManager* proposes a cooperative split with `ET_UPCALL_SPLIT_VM`;
- The *AppManager* decides to accept or reject, either with `ET_OK` or `ET_ERROR`;
- If accepted, the *IaaSManager* allocates a new virtual machine, deploys a stock operating system on it;
- The *IaaSManager* starts the newly allocated virtual machine, waits until it becomes ready, then notifies the *AppManager* with `ET_UPCALL_VM_ADDED`, including all access credentials;
- The *AppManager* deploys his application on the virtual machine using the received the credentials, then add this instance to the load balancer;
- The *AppManager* notifies about the success of the deployment with `ET_DOWNCALL_DEPLOYED`
- Finally, the *IaaSManager* resizes the original virtual machine.

`ET_UPCALL_MERGE_VM` is opposite to the previous call when the *IaaSManager* tries to replace collocated virtual machines with a larger one. This operation is a

sequence of a horizontal scale and then a vertical scale. This is done by requesting to deallocate virtual machines and then resizing the remaining one. This procedure is similar to a `ET_UPCALL_REMOVE_VM` (section 5.3.1.1), but it also requires an additional reconfiguration from the load balancer to take into account the new weight of the scaled-up instance. After a fusion, the application is expected to have a better performance with less tier instances. Like above, we summary the procedure of a cooperative merge as follows.

- The *IaaSManager* proposes a cooperative merge with `ET_UPCALL_MERGE_VM`. A list of virtual machine names being merged are passed using the *ExtraData* field;
- The *AppManager* decides to accept or reject this merge, either with `ET_OK` or `ET_ERROR`;
- The *AppManager* chooses a candidate from the virtual machine list to keep running after this merge, then removes the rest from the load balancer;
- The *AppManager* undeploys these application instances and notifies the *IaaSManager* after it finishes undeployment with `ET_DOWNCALL_REMOVE_VM`;
- The *IaaSManager* frees the virtual machines containing the undeployed application instances, and scales up the remaining one.

5.3.2 Downcall

Opposite to the upcalls, the downcalls are passed from the *AppManager* to the *IaaSManager*. These calls allow the customer to make requests to change the quota, application's tier registration, confirmation for the deployments of application instances, responses of split or merge proposals from the *IaaSManager*, etc. We divide the upcalls into two main sub-categories: requests and confirmations.

5.3.2.1 Requests

The request downcalls are made to properly demand services from the IaaS. The two main calls of this sub-category include tier registration and quota change.

During its initialization, the *AppManager* registers its tiers to the *IaaSManager* using `ET_DOWNCALL_REGISTER_TIER` (note that one *AppManager* can handle more

than one tier). Tier name is put in the *ExtraData* field. After tier registration, all subsequent cooperative downcalls need to specify the tier being used. By supplying these kinds of information, the *AppManager* allows the *IaaSManager* to gain more knowledge about the virtual machines being used in the tier, therefore having better proposals for tier optimization. When receiving the tier registration downcall, the *IaaSManager* saves the tier into an internally-managed tier list, in order to reuse the tier information later; and responds with another upcall, either `ET_OK` or `ET_ERROR`, accordingly.

Based on the actual workload of the application tiers, the *AppManager* resizes its allocated resources to satisfy the workload requirement. After determining the required amount, the *AppManager* makes a cooperative call with call id `ET_DOWNCALL_CHANGE_QUOTA`. The type (CPU, Memory) and amount (number of vCPUs, fraction of CPU core, MB of memory, etc) of quota change are stored in the *ExtraData* field of the call. As discussed in the previous section, based on this amount of quota change, the *IaaSManager* can horizontally and/or vertically scale the application, with the corresponding upcalls (`ET_UPCALL_ADD_VM`, `ET_UPCALL_REMOVE_VM`, or `ET_UPCALL_QUOTA_CHANGED`).

5.3.2.2 Confirmations

The second type of downcalls is confirmation. These calls allow the *AppManager* to respond to the proposals from the *IaaSManager*, vary from the approvals of merge, split, tier deployment, allocation and deallocation of virtual machines. Most of these calls contain only tier name and is dependent on the call context. Two confirmation downcalls being widely used are `ET_OK` and `ET_ERROR`, corresponding to accepted and rejected responses, respectively.

5.4 Synthesis

This chapter described our protocol specification, procedures and algorithms of cooperative IaaS. We also categorized and detailed several important calls for a cooperative resource management system. The next chapter describes our autonomic resource management framework jTune, which is the foundation for our cooperative resource manager jCoop. jCoop can be considered as an implementation of the proposed cooperation specification.

Chapter 6

Contribution: Cooperative Resource Management System

Contents

6.1 jTune Framework 77
6.1.1 System Representation 77
6.1.2 Application Deployment 80
6.1.3 Control Loop 81
6.1.4 Communication between Components 82
6.1.5 Runtime Management 82
6.1.6 Usages 84
6.2 jCoop: Cooperative Resource Manager 87
6.2.1 XenManager 87
6.2.2 AmpManager 91
6.3 Synthesis 93

In the previous chapters, we concluded that existing research works do not consider the combination of two-level resource management and elastic virtual machines, in order to improve both performance for the customer and power save for the provider. From this point of view, we proposed a general direction in section 3.3, which exploits this combination to achieve better optimization for both actors. We also described the protocol, procedures, and specifications of cooperation calls in chapter 6.

Following this direction, we implemented our autonomic management framework named jTune. Based on jTune, we then implement our cooperative resource man-

agement system named jCoop. jTune is used as the same base for two separate resource managers in jCoop, each working on a different level: at the customer level or at the provider level. jCoop was used with the Xen hypervisor and the multi-tier RUBiS application. Note that, using a component model and adaptability in design, it is easy to encapsulate other hypervisors and applications.

This chapter details the model, architecture and implementation of jTune in section 6.1. jCoop is illustrated as an extension of jTune in section 6.2.

6.1 jTune Framework

jTune, a derivative version of TUNe [22], is a framework that aims to simplify the management of legacy applications and the links between them. Originated from the ideas of the autonomic resource management system JADE [21], jTune has the same philosophy: to encapsulate the managed application components into *Wrappers* and to interact with the managed applications through these *Wrappers*. This section describes the general architecture, principles and usage of jTune framework.

In a typical autonomic management system (AMS), the managed application's architecture is presented in the *System Representation* (SR). Application components are bound together in the SR. The AMS uses the architecture in the SR and its resource configuration to deploy the managed components. Using a *control loop*, the AMS then actively monitors and administrates the managed application at runtime.

6.1.1 System Representation

System representation (SR) is a subset of jTune's components, used for presenting and encapsulating the managed application's architecture, its components and the links between them. Before describing the SR, we have the following machine definitions:

- *Manager Machine* is where jTune executes. It is responsible for performing management tasks of the whole infrastructure.
- *Remote Machines* are where the managed application is deployed and executed.

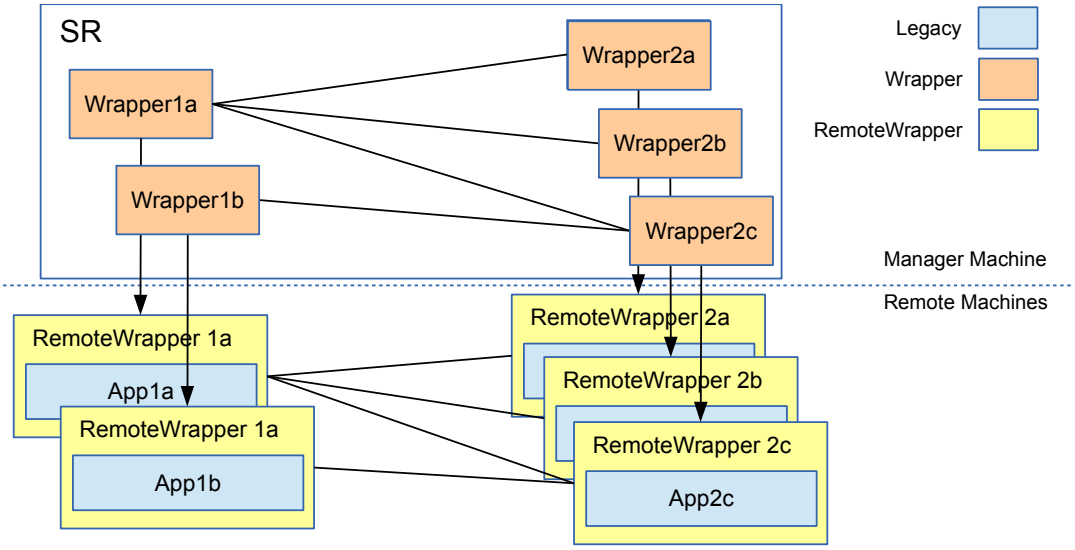

Figure 6.1: jTune's SR

Note that, the manager machine or the remote machines can be either physical servers or virtual machines, depending on the infrastructure.

System representation and deployment in jTune is illustrated in figure 6.1. We describe the main components in jTune's SR as follows.

Legacy. A *Legacy* is defined as an application component which will be deployed and managed at runtime. In other words, autonomic administration of *Legacies* is the main objective of the jTune framework. A *Legacy* is generally prebuilt into binary packages to be ready for deployment. jTune is responsible for transmitting these packages to remote machines and executes them at the destination at runtime. Examples for *Legacies* of a typical multi-tier J2EE application are: Apache Web Server, Tomcat Application Server, MySQL Database Server.

Wrapper. A *Wrapper* is defined as a component at the manager machine which implements the management interface for a specific *Legacy*. The *Wrappers*' interface provide low level management tasks for the provided application, such as start, stop, configure (e.g. setting listening port, directory for deployment), etc. When requesting a deployment for one application instance, the administrator needs to specify which *Wrapper* will manage this instance. Listing 6.1 shows an example of two important methods, `start()` and `stop()`, for managing Apache.

Listing 6.1: Wrapper for Apache

```
public void start() {
  super.start();
  // set LD_LIBRARY_PATH environment variable
```

```
    sendEvent(new Event(EventType.ET_SET_LIB_DIR, "", serverRoot +
        "/lib"));
    executeCommand(serverRoot + "/bin/apachectl start")
    String apachePids = "";
    while (apachePids.length() == 0) {
      sleep(Settings.getInt("monitor_interval"));
      apachePids = executeCommand("pidof httpd");
    }
  }

  public void stop() {
    sendEvent(new Event(EventType.ET_SET_LIB_DIR, serverRoot + "/
        lib"));
    executeCommand(serverRoot + "/bin/apachectl -k stop");
  }
```

RemoteWrapper. The *RemoteWrapper* is the only remote component in jTune. It is deployed along with the legacy to the remote machine, using the same protocol (SSH, OarSH). After being deployed to the remote machine, the *RemoteWrapper* is launched. It then starts waiting for requests from the manager node. Finally, it connects back to jTune at the manager node and notifies its ready status.

The corresponding *Wrapper* at the central manager machine communicates with the deployed *RemoteWrapper*. Management actions (e.g. starting or stopping legacy, setting port for a server application) are sent from the *Wrapper* to the *RemoteWrapper* and executed at the remote machine. Note that all commands executed at the remote machines will be sent using a separate communication channel to reduce SSH's performance impact. SSH or OarSH is only used for the initial deployment and launch of the *RemoteWrappers*.

Link. A *Link* allows to bind two *Wrappers* together. By binding the *Wrappers* at the SR construction time, the management policy can traverse in the managed application's architecture to look for a specific *Wrapper* at runtime. Binding is performed simply by using with the provided SR's method, for example:

Listing 6.2: Binding Apache instances to MySQL instances

```
// bind all Apache Servers to all MySQL Servers
ArrayList<MySqlWrapper> mySqls = sr.getWrappersByClass(
    MySqlWrapper.class);
ArrayList<ApacheWrapper> apaches = sr.getWrappersByClass(
    ApacheWrapper.class);

for (ApacheWrapper apache : apaches) {
  for (MySqlWrapper mySql : mySqls) {
```

```
      sr.addWrapperBinding(apache, mySql);
    }
  }
```

6.1.2 Application Deployment

After the SR is constructed, jTune's *DeploymentManager* uses this architecture knowledge to deploy the application to the remote machines. It exploits SSH as the initial deployment and remote control protocol. SSH is a widely used protocol for remote management in cloud infrastructures. Taking advantages of the JSch[1] library, jTune does not need any external SSH client to connect to its managed machines (both physical and virtual ones). Additionally, using JSch, jTune can optionally use the given access credentials (username, password) to manage the remote machines, without the need of pre-authorization using generated key-pairs.

When an application component is requested for deployment, a machine is automatically chosen from a resource pool. This task is handled by *ResourceManager*. This component manages a set of available machines[2] for deployment with a configuration file. These configurations include server host name (or IP address), protocol to access the server (SSH, OarSH[3]), access credentials and deployment directory. A typical configuration file for physical machines is shown as follows.

Listing 6.3: Configuration for Resource Pool

```
#name  protocol  javapath        dir    isPwd  user   pw/keypair
node1  ssh       /usr/bin/java   /tmp/  1      root   rootpw
node2  ssh       /usr/bin/java   /tmp/  1      root   rootpw
node3  ssh       /usr/bin/java   /tmp/  1      root   rootpw
node4  ssh       /usr/bin/java   /tmp/  1      root   rootpw
```

Depending on the allocation request, the *ResourceManager* decides to select a physical server or a virtual machine as the deployment target. The legacy's compressed binary is transmitted with the specified protocol to the selected target, unpacked to the destination directory and is ready to be managed.

[1] Java Secure Channel, a pure Java implementation of SSH2. http://www.jcraft.com/jsch/
[2] The deployment target can be a virtual machine or a physical server.
[3] A variant of SSH for use in Grid5000 – the French National Grid for Scientific Purposes

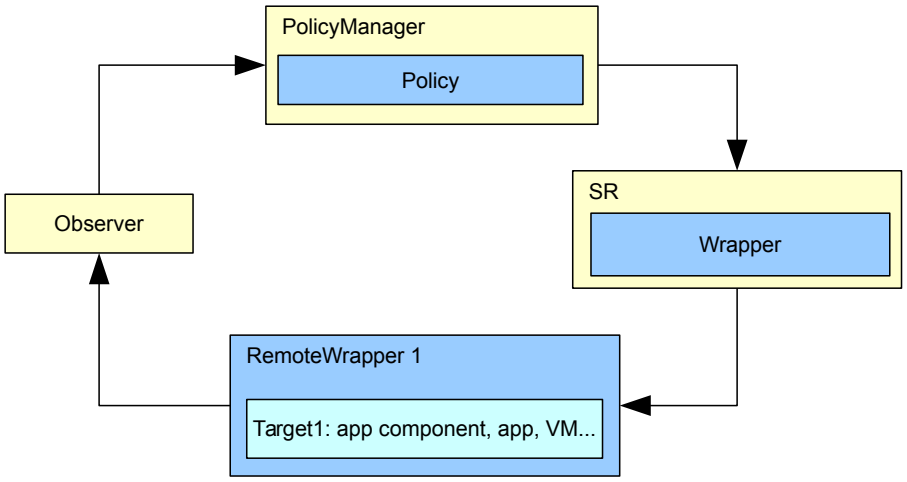

Figure 6.2: jTune's Main Components in the Control Loop

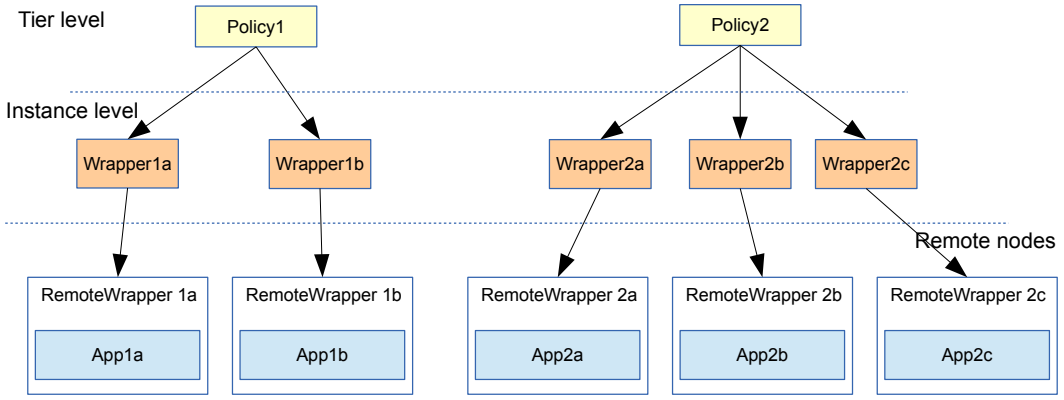

Figure 6.3: Relation of *Policies*, *Wrappers* and *RemoteWrappers*

6.1.3 Control Loop

jTune's control loop monitors and administrates the whole infrastructure at runtime. From the point of view as an autonomic resource manager, jTune implements all necessary components of autonomic computing [40]: an observer, a decision core and an effector. Figure 6.2 illustrates the simplified components of jTune acting as the control loop.

jTune's *Observer* listens to all information transmitted from all remote machines. Typical gathered informations include: CPU load and memory usage of the remote machine, process status of the managed application process, etc. Based on the collected data, the *Policy*, acting as the *decision core*, makes decisions on how to manage the application (e.g. deploy, reconfiguration, repair, etc.). The decision is then sent to the *effector* to be performed. In jTune, the *Wrappers* (at the manager machine) and *RemoteWrappers* (at the remote machines) act as the *effector*.

Since jTune's objective is to manage the whole application, it has several *Policies*. As mentioned above, a *Wrapper* only executes primitive tasks (start, stop, configure, etc.). The associated *Policy* works at a higher level: it gathers informations about the application components, and bases on this data, makes decision to manage it at runtime. Generally, a *Policy* collaborates with a number of *Wrappers* (figure 6.3), each maintaining a single application instance. As a result, the *Policy* administrates the whole application tier in a multi-tier web application. For instance, an *ApachePolicy* works with a set of *ApacheWrappers* to manage the whole Apache tier. Note that all *Policies* and *Wrappers* are executed in the main manager node, all tasks assigned to each *Wrapper* are transmitted to the remote machines and are performed by the *RemoteWrapper*.

6.1.4 Communication between Components

In jTune, components communicate with each other using *Events*, both locally (e.g. *Observer-Policy* and *Policy-Wrapper*) and remotely (e.g. *Wrapper-RemoteWrapper*, *RemoteWrapper-Observer*). Figure 6.4 summarizes the event flows and detailed jTune's component architecture. We extensively use event-driven programming model in jTune. Events are asynchronous, allow autonomic management tasks to be non-blocked and ensure that administration tasks (deployment, configuration, launch, repair, etc.) can be performed in real time. The *EventManager* maintains an internal list of registered *Events*. Each component in jTune needs to subscribe its managed events to the *EventManager* during their initialization. Whenever receiving an *Event*, the *EventManager* dispatches it to the corresponding component according to this list.

6.1.5 Runtime Management

Once the application is deployed and its *RemoteWrapper* is ready, the corresponding *Policy* starts managing the application's life cycle. The *Policy* is written by the administrator to implement his desired resource management policy. We determined the following major tasks in the application's life cycle that each *Policy* must implement (figure 6.5):

- **Configuring the application parameters.** Before starting the application, the *Policy* needs to prepare the application to make it fit with the current configuration, e.g. the application port, the connection to other tiers, load balancers, database initialization, etc. During the configuration phase, the

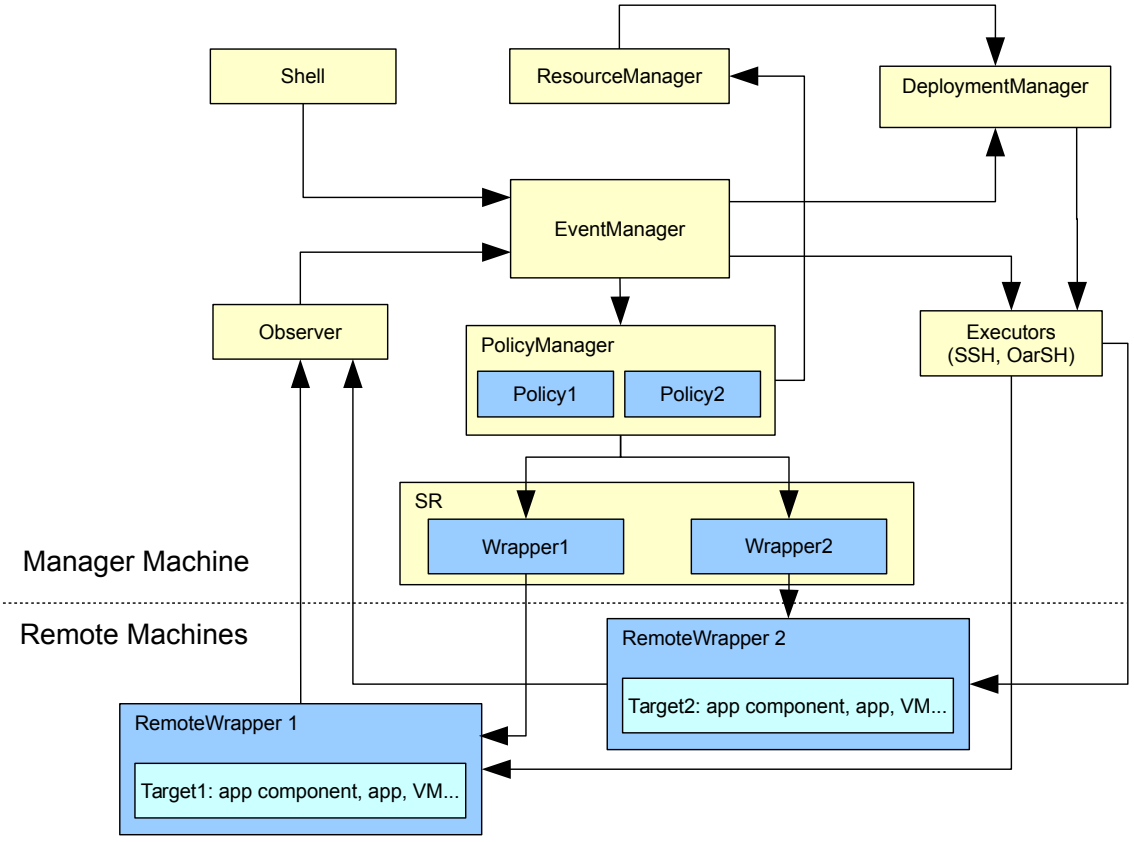

Figure 6.4: Event Flows in jTune's Architecture

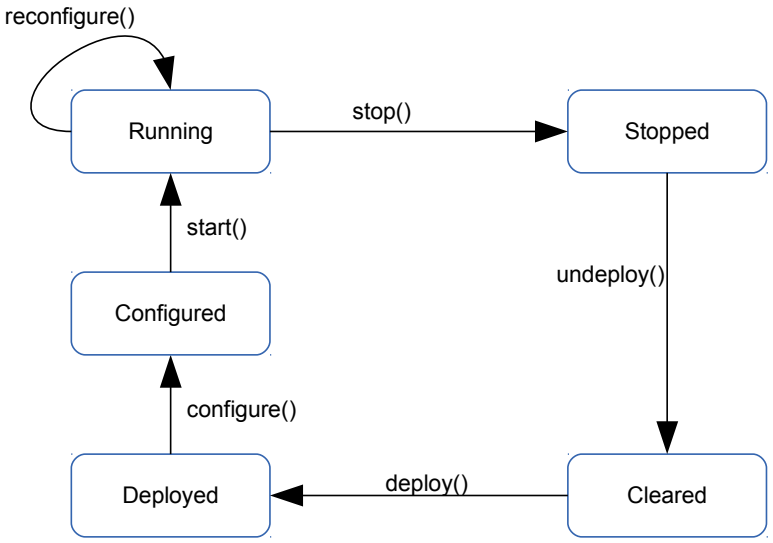

Figure 6.5: Application Lifecycle in an Autonomic Manager

Policy uses the predefined settings (specified by the administrator) and requests the *Wrappers* to set them accordingly. These parameters are typically written to the application's configuration file. As a result, the *Wrappers* are tier-specific.

- **Launching the application process.** The *Policies* then starts the application with the help from its associated *Wrapper* and *RemoteWrapper*. The command to launch the application (wrapped inside an *Event* `ET_EXEC`) is passed from the *Wrapper* to the *RemoteWrapper*. At remote destination, the *RemoteWrapper* uses system calls to execute the binary. Additionally, it automatically starts the embedded monitoring probe. This probe periodically gathers various information about the deployed system and application, such as system load, port status, existence of a process, etc. This information is reported back to the central jTune's *Observer* for further decisions.

- **Reconfiguring the application.** Various applications need to be configured in real time without being stopped. At the runtime, based on the gathered metrics, the responsible *Policy* reconfigures the application instances with its associated *Wrappers* to satisfy the required goals (set by the administrator). The reconfiguration task includes, but not limit to: reacting with failures, dynamically resizing tier in a multi-tier application, changing the listening TCP port, etc.

- **Terminating the application.** At the end of the application life cycle, the *Policy* component requests the *Wrapper* to stop the application, clears the destination directory, then stops the associated *RemoteWrapper* at the target node. From this time, the target machine is freed and is available for later use, such as another application deployment.

6.1.6 Usages

After an introduction of the jTune's components as an AMS, we show in this section the basic usage of jTune from the administrator's point of view. jTune is a Java application, built and packaged into a JAR[4] file. Additionally, section 6.1 mentioned that *RemoteWrapper* is deployed along with the application to the target destination. Therefore, the manager node and all managed machines (virtual machines or physical servers) must have a JRE[5] pre-installed. The utilization of jTune is divided into 3 steps: configuration; specification of the wrappers and policies; build and launch.

[4]Java ARchive
[5]Java Runtime Environment

In the **Configuration** step, the administrator modifies the jTune's configuration file to fit his needs. These configurations are managed by the *Settings* component. The important values include:

- *node_pool_file*: name of the file containing a list of available physical nodes for deployment.
- *observer_service* and *observer_port*: the interface name and port of the *Observer* component on the manager node. This interface is the central point for aggregating events from the deployed *RemoteWrappers*.
- *remote_service* and *remote_port*: the end point of *RemoteWrapper* on the managed nodes. This interface allows the transmission of commands (in the form of *Events*) from central *Wrappers*.
- *nfs_server* and *nfs_path*: indicates the NFS[6] server and mount path that contain the virtual machine images and settings. We support NFS as a means to support virtual machine live migration in our experiments with Xen later.
- *log*-related settings: jTune supports a wide range of logs from event log to remote or local log. This helps the administrator to have an in-depth knowledge about the management system at runtime.

Additionally, a list of nodes with access credentials also needs to be specified similar to listing 6.3, so that the *NodeDeployer* can locate the resource pool at runtime. At the end of this step, the administrator needs to package the application into a single compressed binary. jTune supports various file formats (e.g. zip, tar.gz, tar.bz). The application package will be automatically uncompressed after being transferred to the target machine.

The administrator then implements the **Wrapper** and **Policy** for his application. Note that the *RemoteWrapper* can be reused without further modifications. We made a *GenericWrapper* component for the administrators to base their own *Wrappers* on. This component eases the basic tasks for administrating the wrapped application: executing a command, modifying a configuration file, etc. Similarly, the customized *Policy* for each resource management policy is based on a *GenericPolicy* component. For example, a *Policy* typically starts / stops the application, asks the *RemoteWrappers* to periodically monitor CPU load, handles idle or peak loads using replication of application instances.

Finally, the administrator **launches** the whole system. This task is done from the manager node. From the command line, the administrator can specify the extra

[6]Network File System

parameters to pass to jTune for automating the management process. During the deployment and execution of the managed application, the administrator can interact with jTune and override management policy using the provided console using the *Shell* component.

Since jTune is only a generic autonomic resource management framework, it cannot be used in a specific IaaS environment. The next section described our work jCoop as an extension of jTune for managing resources in a cooperative IaaS.

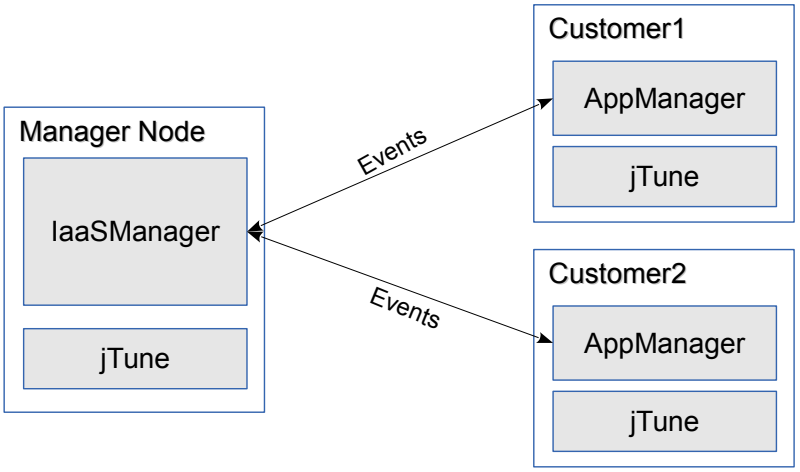

Figure 6.6: IaaSManager and AppManager in jCoop

6.2 jCoop: Cooperative Resource Manager

We built jCoop as the implementation of the cooperation specification on top of the autonomic management system jTune. jCoop consists of two main components: the *IaaSManager* and the *AppManager*. One *IaaSManager* can serve multiple *AppManagers* (each for a separate customer) at the same time (figure 6.6). These managers inherit jTune's *Observer* and *Wrappers* to communicate with each other using *Events*. The *IaaSManager* works at the provider's cloud manager node, while each customer has at least one *AppManager* on their allocated virtual machines.

In order to evaluate our cooperative resource management policy in the next chapter, we chose a specific platform and a target application. On the provider side, we consider **Xen** as the virtualization platform. Our resource management policies in the *IaaSManager* consider both CPU and memory management. On the customer side, we consider managing a typical **multi-tier application** using Apache / MySQL / PHP (*AMP*) software stack. This is one of the most popular platforms for web application development and deployment. We implemented our *XenManager* and *AmpManager* for these targeted platform and application, respectively.

6.2.1 XenManager

Our *XenManager* is implemented as a simplified IaaS using Xen as the hypervisor to further demonstrate the benefits of a cooperative resource management system in the next chapter. The *XenManager* has basic functionality of a IaaS: allocation, deallocation, launch and termination of virtual machines. Our *XenManager* also

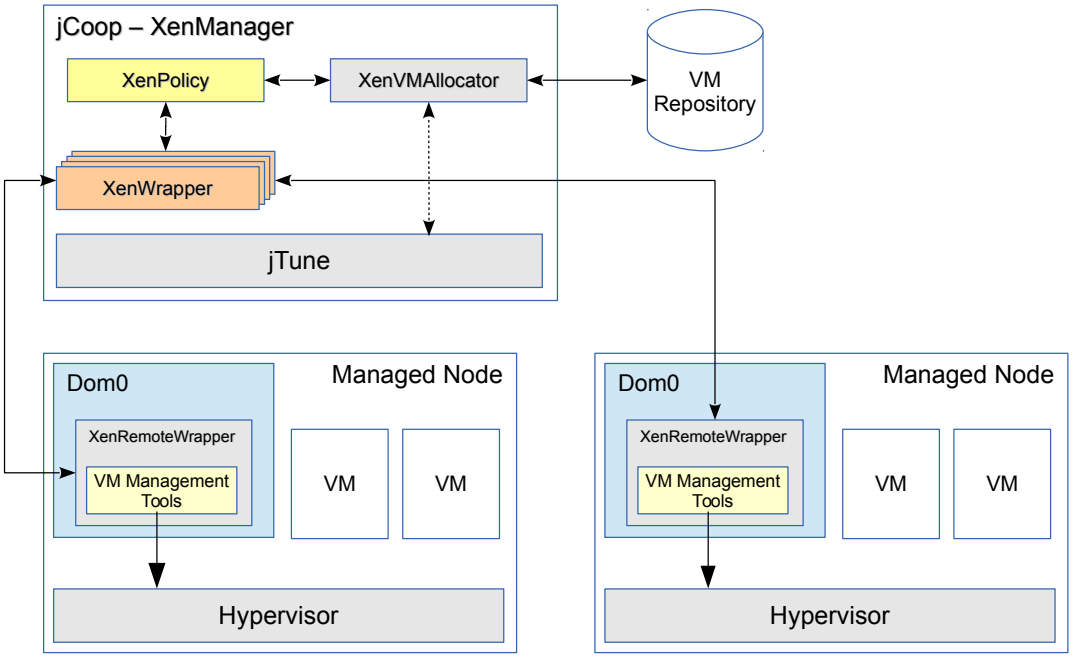

Figure 6.7: XenManager's Components in jCoop

supports the discovery of virtual machine images and their corresponding configurations using NFS (see section 6.1.6 for NFS parameters). The *XenManager* searches for virtual machines in this specified repository at startup. After being added to the resource pool, these virtual machines are ready for the customers. Note that the customer can prepare his own virtual machines and deploys into this repository, so that they can be reused later without the need of deploying and configuring the application.

Similar to most conventional IaaS, our *XenManager* uses virtual machine live migration for saving physical resources. Depending on the experiment, the migration plan is computed using the real CPU and memory usage or using the booked CPU and memory resource of the allocated virtual machines. Our migration algorithm is based on *Best Fit Decreasing* with heuristics to improve calculation time. We also limit the frequency of virtual machine migrations to avoid bandwidth bottlenecks, causing performance degradation for the applications in the same network. After migration, the freed servers are suspended and resumed using provided tools from the host operating systems (particularly `pm-suspend`) and Wake-On-LAN feature.

Elasticity

The *XenManager*'s support for **elastic CPU** is implemented using the default *credit scheduler* in Xen hypervisor. To distribute CPU processing power among virtual machines, Xen considers each virtual machine similar to a process in operating

systems: it switches CPU among its virtual machines after each timeslice. This process is called VM scheduling. Xen's default credit scheduler allows each managed virtual machine to have a guaranteed a portion of CPU's processing power with different parameters: weight, cap, vcpu, and timeslices.

- *Weight* indicates the priority for CPU computing power of a virtual machine, relative with the others.
- *Cap* is used to define the upper threshold of the CPU load for a virtual machine, given in percentage.
- *vCPU* is the number of virtual CPUs allocated to a virtual machine.
- *Timeslice* specifies the duration of each virtual machine's execution before the scheduler interrupts it and switches to another virtual machine. This value can be tweaked to change the responsiveness of virtual machines.

Our *XenManager* exploits the ability to use these parameters, mainly *cap* and *vCPU*, to dynamically resize its elastic virtual machines at runtime.

Additionally, the *XenManager* handles **elastic memory** by using Xen's *memory ballooning* mechanism. Xen added supports dynamic virtual machine size with its *ballooning driver* (described in section 3.2) since version 3.3. By inflating and deflating the memory balloon at runtime, *XenManager* can resize the effective memory size for its virtual machines. Note that the amount of memory at runtime for each virtual machine cannot exceed the maximum amount of allowed memory (which was specified at its boot time).

Xen's virtual machine management tools allow changing these values with the provided Xen management user interface (`xl`) from the host. The mentioned CPU values can be changed with `xl`'s `vcpu-set` and `sched-credit` tool. On the other side, resizing memory for a running virtual machine can be done with `mem-set` tool. Our *XenManager* utilizes these provided tools to support elasticity for its managed virtual machines (i.e. the ability to *vertically* scale them), both in terms of CPU processing power and effective memory size.

Components

Our *XenManager* consists of three major components: *XenVMAllocator*, *XenWrapper* and *XenPolicy* (figure 6.7). These components communicate with each other using internal calls or *Events*. Note that all of these components are executed inside the provider's management node, all decisions and commands to manage the

virtual machines are transmitted to the remote nodes using *Events* and then executed by *XenRemoteWrappers* (a slightly modified version of *RemoteWrappers*).

- *XenVMAllocator* is responsible for discovering virtual machines in the repository. It then analyzes the configuration file, simulates DHCP requests to obtain the automatically assigned IP addresses, and adds these virtual machines to the resource pool for later use.
- *XenWrapper*, based on jTune's *GenericWrapper*, is responsible for managing the virtual machines on remote servers by using `xl` tool on each host. By using this tool and its parameters, *XenWrapper* indirectly manages the hypervisor on each physical server.
- *XenPolicy* is the core our *XenManager*. It listens to requests from customers (with their *AppManagers*), serves these requests, optimizes resource usage in the given resource pool (using both horizontal and vertical scaling). Additionally, *XenPolicy* implements the cooperation calls with the specifications described in chapter 6.

At startup, *XenPolicy* registers many events with the *EventManager* to work with both non-cooperative and cooperative *AppManagers*. The most important ones include:

- `ET_ADD_VM` is used for allocating a virtual machine. *XenManager* looks for an available virtual machine in the pool, starts it and passes the access credentials to the customer. If no virtual machine in the pool is suitable for this request, *XenManager* creates a new one with the specified resource requirement. This event is not used in the cooperative scenario (with the *AppManager* being aware of the cooperative resource management).
- `ET_REMOVE_VM` is used for deallocating an existing virtual machine. The *XenManager* stops the embedded *Wrapper* inside this virtual machine (responsible for monitoring this virtual machine), undeploys it, and stops this virtual machine using the corresponding *XenWrapper*. In a non-cooperative case, this *Event* is transmitted from the *AppManager* to the *IaaSManager* to directly request a virtual machine removal. On the other hand, in a cooperative scenario, this event is used internally by the *XenManager* to remove a virtual machine when there is a possibility for a cooperative merge.
- `ET_DOWNCALL_REGISTER_TIER` is used for retrieving tier registrations from the cooperative *AppManagers*. Our *XenManager* uses an internally managed list to store these tiers and to look up later.

- `ET_DOWNCALL_CHANGE_QUOTA` is used for changing the quota for a specific tier. *XenManager* implemented the algorithms 1, 3 and 2 in chapter 6 to change quota of the registered tiers at runtime.

6.2.2 AmpManager

The previous section described our *XenManager* as an implemention of the *IaaS-Manager* on the provider's side. This section highlights the main components of the *AmpManager* built on top of jTune on the customer's side, using a typical multi-tier application with the Apache / MySQL / PHP (*AMP*) architecture (figure 6.8). Similar to the J2EE architecture (figure 3.1), *AMP* has a web server tier (Apache) and a database tier (MySQL). Unlike J2EE, *AMP* does not have a dedicated tier for application servers. In this situation, PHP acts as the application server in *AMP*, because all server side scripts are interpreted and executed by a PHP processor module embedded in Apache.

Being a multi-tier architecture, *AMP* needs to distribute the incoming requests to its tier instances, similar to the HTTP load balancer, Mod_JK and MySQL proxy in the J2EE example (figure 3.1). We consider HaProxy[7], one of the most popular TCP load balancers, as the load balancer for the Apache tier and the MySQL tier in the customer's application. Using these load balancers, the customer can *horizontally* scale his application tier (Apache or MySQL) according to the runtime load easily by adding or removing tier instances.

Components

Our *AmpManager* consists of various *Wrappers* and *Policies*, each combination handles a specific tier. Note that the HaProxies acting as load balancers also need to be managed. As a result, the *AmpManager* has the following *Wrapper*/*Policy* combinations:

- *HttpHaProxyWrapper* and *HttpHaProxyPolicy* are responsible for managing the HTTP load balancer. Basic tasks include starting and stopping the balancer itself. Furthermore, they also allow adding and removing tier instances into the balancer. This layer needs to take into account the weight of each Apache's virtual machine sizes. The changes to HaProxy can be applied on-the-fly by modifying its configuration file and notifies the running instance.

[7] http://haproxy.1wt.eu/

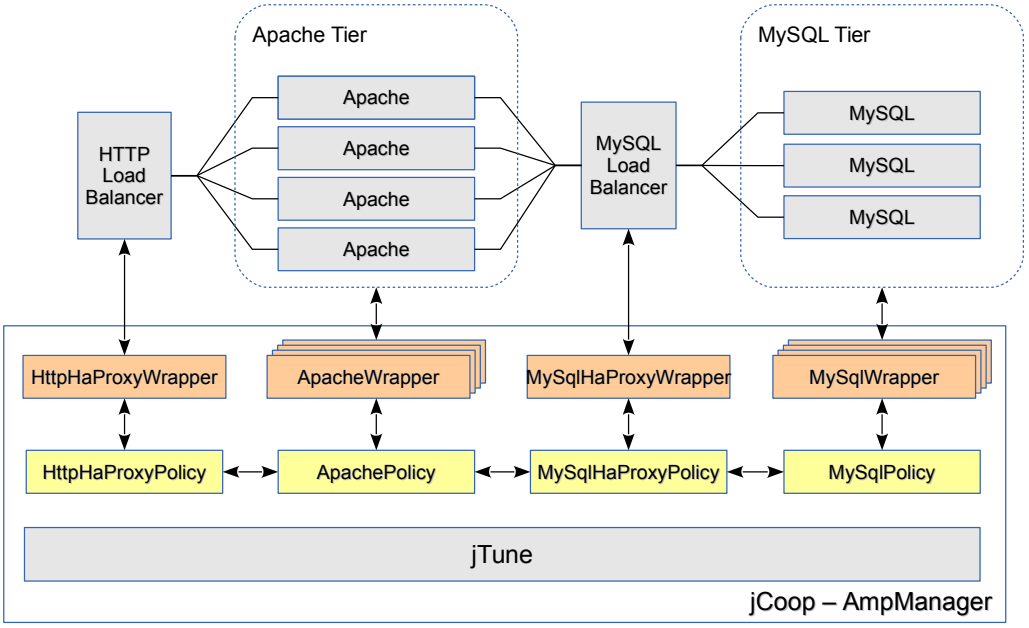

Figure 6.8: AmpManager's Components in jCoop

- *ApacheWrapper* and *ApachePolicy* are responsible for the management of Apache tier. The *ApacheWrapper* provides low-level tasks for administrating Apache instances, such as starting, stopping, setting listening port, and configuring the customer's PHP application. At a higher level, *ApachePolicy* gathers Apache loads, memory usage and horizontally resizes this tier at runtime.

- *MySqlHaProxyWrapper* and *MySqlHaProxyPolicy* manage the MySQL load balancer, similar to the HaProxy acting as the HTTP load balancer above.

- *MySqlWrapper* and *MySqlPolicy* manage the MySQL tier. Similar to the Apache tier, they horizontally scale this tier according to the actual CPU load and memory usage. The configuration task of *MySqlWrapper* has database-specific actions, such as creating user accounts and initializing databases.

The *AmpManager* registers itself with the *EventManager* to process various types of *Events*. Since our later experiments consist of both cooperative and non-cooperative scenarios, the *AmpManager* handles tier resizing with both vertical and horizontal scale. In a non-cooperative scenario, the *AmpManager* monitors each tier and resize it with `ET_ADD_VM` and `ET_REMOVE_VM`, according to the actual load at runtime.

On the other hand, in a cooperative scenario, as an implementation of the cooperative specification (chapter 6), the *AmpManager* registers several upcall events

with the *EventManager*. These events allow the *AmpManager* to communicate and to response with requests and notifications from the *XenManager*. These events include:

- `ET_UPCALL_SPLIT_VM`
- `ET_UPCALL_MERGE_VM`
- `ET_UPCALL_ADD_VM`
- `ET_UPCALL_REMOVE_VM`

Upon receiving these calls, the *AmpManager* decides to accept or to reject, based on the current status of each tier. Additionally, the *AmpManager* requests quota changes as needed with `ET_DOWNCALL_CHANGE_QUOTA`. We also limit the frequency and amount of each quota change to avoid the *XenManager* being overloaded with too many quota changes at runtime.

6.3 Synthesis

In this chapter, we have illustrated the architecture and implementation of jTune, from an autonomic management system's point of view. We presented jTune's System Representation, Control Loop and Runtime Management. We also described the roles and tasks of each major components in jTune, particularly the *Wrapper*, *Policy*, *ResourceManager*, *DeploymentManager*, *PolicyManager* and *EventManager*. The most important parameters in jTune's configuration are also detailed. We then insisted on the close relationship between *Policy*, *Wrapper* and *RemoteWrappers* (on three different levels), three main components for legacy application's management in jTune. These components are application-specific and need to be implemented by the administrator. To ease this task, we made the *GenericWrapper* and *GenericPolicy* for the administrators to base their works on.

This generic administration management framework is then used as a base for jCoop, our implementation of the cooperation call's specification (previously detailed in chapter 6). We chose Xen as the target platform, and Apache / MySQL / PHP software stack as the target application. Two main components in jCoop are *XenManager* and *AmpManager*, representing the *IaaSManager* on the provider side and *AppManager* on the customer side, respectively. We detailed the subcomponents inside *XenManager*, *AmpManager*, the links among them, and also the methods to achieve elasticity and cooperation between these components.

The next chapter evaluates the benefit of the proposed cooperative resource management policy, using the implemented *XenManager* and *AmpManager*.

Chapter 7

Evaluations

Contents

7.1 Experimental Setup 96

7.1.1 Hardware Testbed 96

7.1.2 Customer Application: RUBiS 96

7.1.3 Metrics 98

7.1.4 Workload Profile 98

7.2 Evaluations 99

7.2.1 Scalability and Elasticity 101

7.2.2 Performance Overhead 102

7.2.3 Virtual Machine Occupation 104

7.2.4 Physical Server Utilization 105

7.3 Synthesis 106

The objective of this chapter is to show the effectiveness in minimizing resources usage and maximizing performance of the cooperation strategy. To reach this goal, the experiments are divided in 3 major parts: (1) confirmation of the ability to split and merge elastic virtual machines; (2) evaluation of the benefit to reduce performance overhead; and (3) evaluation of the benefit to improve resource usage.

7.1 Experimental Setup

7.1.1 Hardware Testbed

Our experiments were performed in our private, simplified IaaS with two clusters (figure 7.1). The first cluster (*SlowCluster*) consists of 5 identical nodes Dell Optiplex 755. Each node in this cluster is equipped with an Intel Core 2 Duo 2.66GHz (2 cores) and 4GB RAM. The second cluster (*FastCluster*) consists of two HP Compaq Elite 8300, each equipped with an Intel Core i7-3770 3.4GHz (quad core with HyperThreading) and 8GB RAM. All nodes are installed with Debian Squeeze on top of Xen 4.1.5 and connected with 1Gbps switch. We configure each dom0 (the host operating system) to have $^1/_3$ of a physical machine (50% of a CPU core and 1GB of memory). Our node usage is distributed as follows:

- Our *XenManager* is installed on a dedicated node in *FastCluster*. This *IaaS-Manager* instance is executed in a non-virtualized system to have the maximum performance at runtime. As previously described in the previous chapter, all components of *XenManager* are executed in this node, the central point of our private cloud's management.
- On the second unvirtualized node in *FastCluster*, we install a NFS server for the cloud's virtual machine repository, so that virtual machine live migration can be performed between the nodes. Additionally, a DNS forwarder and a DHCP server (both are integrated in *dnsmasq*[1]) are also collocated on this node. Therefore, this node provides basic network services for our private cloud (NFS, DNS and DHCP).
- All 5 nodes of *SlowCluster* are used as the resource pool. These resources are virtualized and provided on-demand to the customer: virtual machines will be allocated, deployed and executed on these nodes.

7.1.2 Customer Application: RUBiS

Our target application is a multi-tier application named RUBiS (Rice University Bidding System [14]), an implementation of eBay-like auction system. RUBiS is composed of several tiers: a web tier, an application tier and a database tier.

[1] http://thekelleys.org.uk/dnsmasq/doc.html

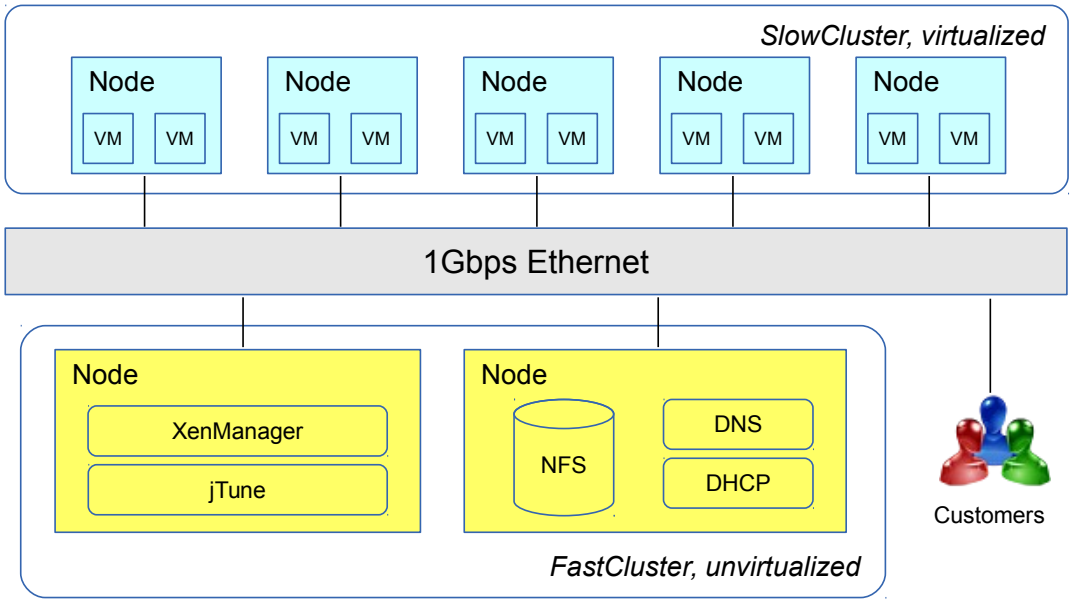

Figure 7.1: Our Private IaaS Cloud

There exists several implementations of RUBiS: Java Servlet, PHP and Enterprise JavaBean. We chose the PHP version to fit with our managed *AMP* software stack (described in the previous chapter). RUBiS provides multiple types of interactions from the external clients, including listing product, managing user accounts, buying, selling and bidding items. Most of these activities are dynamic web pages. Thus, most requests need to be processed by all tiers: static content are served directly by Apache, while dynamic pages are processed and rendered by PHP modules with the data queried from the MySQL tier.

Additionally, RUBiS allows the simulation of web clients to benchmark its implementation. This benchmark simulates many web clients at the same time, each represents a specific user. During the benchmark, RUBiS benchmark generates three different phases: up ramp, main session and down ramp. In the *up ramp* phase, RUBiS benchmark gradually increases the number of client. At the end of up ramp, the *main session* executes for a predefined amount of time with the maximum number of clients. Finally, the *down ramp* phase steadily reduces the number of simulated clients.

In our experiments, we consider managing only MySQL tier because it is the first tier to be saturated when the number of clients increases.

7.1.3 Metrics

We define several metrics in our evaluations to measure the effectiveness of a cooperative resource management system. These metrics include:

- **Response Time** is the average response time of RUBiS application to the incoming requests. Response time represents the Quality of Service that the customer must ensure. The customer also uses this metric as a parameter to scale his application: high response time means the application is being saturated and overloaded.
- **Physical Machine Utilization** (ψ) is the accumulated number of powered-on physical servers for every second (i.e. the sum, for all servers, of the accumulated time that each server was powered-on). ψ represents the energy usage of the whole cloud infrastructure: the lower ψ is, the less time physical machines are powered on. ψ is measured in seconds.
- **Virtual Machine Occupation**: Given a cap (the capacity of CPU resource) value $0 < c_{k,i} \leq 2$ (our *SlowCluster* has 2 cores on each machine) in each duration $t_{k,i}$ (in seconds) allocated to a virtual machine VM_k, we define *the occupation* of it as ω_k in our experiments as follows.

$$\omega_k = \sum_{i=1}^{n} c_{k,i} \times t_{k,i} \tag{7.1}$$

From this definition, we have a virtual machine occupation of each application App_j is

$$\Omega_j = \sum_{k=1}^{n} \omega_k \tag{7.2}$$

Using the occupation value as a metric, we can evaluate the effectiveness of each resource management policy toward minimizing the *booked* virtual machine resource for the customers.

7.1.4 Workload Profile

We simulate a scenario in which three different customers share the same IaaS. The synthesized workload for each application is generated by a RUBiS benchmark tool. As a result, we have three different workloads for three RUBiS applications like illustrated in figure 7.2. The up ramp, session and down ramp of each workload generator are 4 minutes, 18 minutes and 14 minutes, respectively.

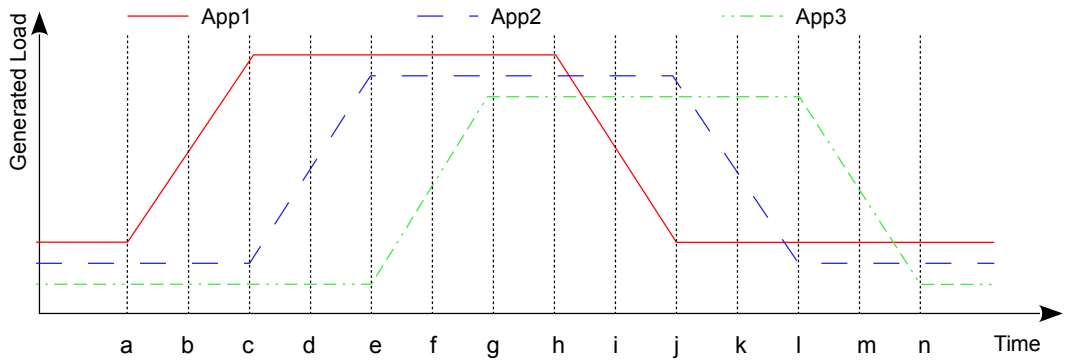

Figure 7.2: Generated Workload

Resource Management System	jCoop with *XenManager* and *AmpManager*
Virtualization Platform	Xen
Customer's Architecture	Multi-tier Apache / MySQL / PHP
Customer's Application	RUBiS
Workload Profile	RUBiS Benchmark (figure 7.2)
Comparison Metrics	Response Time, PM Utilization ψ and VM Occupation Ω

Table 7.1: Summary of our Evaluation's Conditions

7.2 Evaluations

This section describes and discusses our experiments to confirm the benefit of our cooperative resource management, particularly in providing elasticity for virtual machines, reducing performance overhead and improving both virtualized and physical resource utilization ratio. Various aspects of these experiments are summarized in table 7.1.

We define the following scenarios:

- **Static placement**. In this situation, one big virtual machine of each RUBiS application occupies a whole physical server. This amount of resource is enough for these virtual machines to deal with our experiment's workload profile. With only one virtual machine per server, this placement is expected to have the *lowest response time* for the customer's application (the lower-bound of response time for the application). As a result, Static can be considered as an "ideal" virtual machine placement for maximizing application's perfor-

mance. However, this placement clearly wastes resources, because it does not allocate or relocate resources at runtime.

- **Server Consolidation Only** (SCO) is a less static scenario, in which the customer does not have on-demand resource manager (i.e. without the *AmpManager*), but the provider implements his *XenManager* with server consolidation. In other words, a fixed number of instances for each tier (two in our experiments) is provisioned and allocated throughout the application life cycle, even when they are idle. Note that, the consolidation service in this scenario *bases on the actual CPU load* to perform live migrations. This placement is expected to have *highest application response time* without an On-demand *AmpManager*, because this scenario always occupies most virtual machines (no tier sizing by the *AmpManager* is performed). SCO is also expected to have best hardware utilization with active virtual machine migration using CPU load. This strategy is implemented by research works discussed in section 4.1.1.2.

- **Both Level, Independent** (BLI) is the two-level, non-cooperative scenario. In this situation, the *XenManager* and *AmpManager* work without any coordination: *XenManager* migrates virtual machines to ensure server consolidation, while *AmpManager* minimizes the number of application instances. No knowledge is shared and no cooperative call is made. This scenario is similar to resource management *in the conventional IaaS* and is discussed by research works in section 4.1.2.1. *XenManager* uses *booked CPU resource* to make migration decisions, because *AmpManager* on the customer level has already effectively minimized the number of application instances based on the CPU load: idle instances are removed.

- **Both Level, Cooperative** (BLC) is the scenario in which jCoop takes into account resource management of the two levels at the same time: the IaaS level and the application level. We configured jCoop to have $^1/_6$ of a physical machine (memory and CPU) as each quota addition step. *XenManager* in BLC also uses *booked CPU resource* to migrate virtual machines in order to ensure server consolidation, similar to BLI.

To evaluate the benefit of our cooperative resource management policy, we run jCoop various times with the generated workload (figure 7.2), then we compare the defined metrics (section 7.1.3) using the above scenarios.

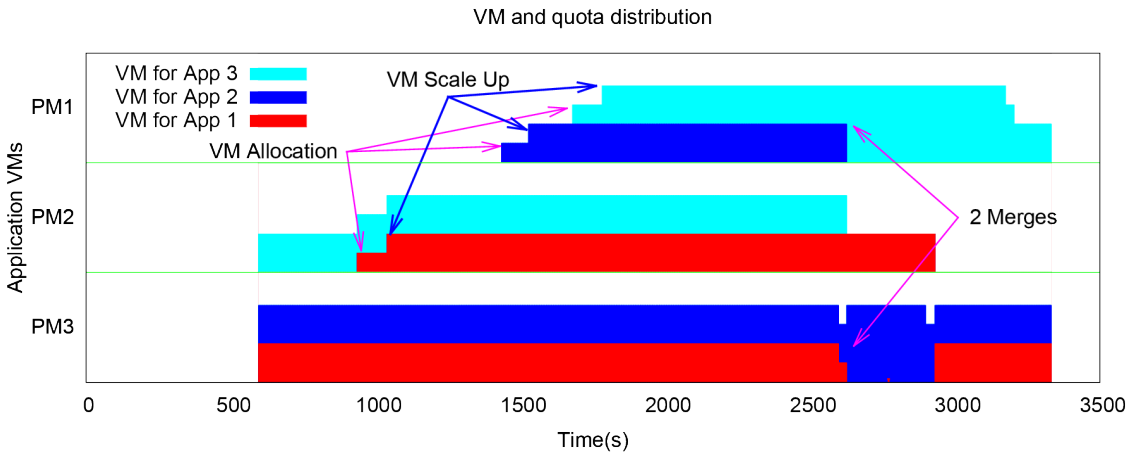

Figure 7.3: BLC: Virtual Machine Placement and Quota Distribution

7.2.1 Scalability and Elasticity

Firstly we confirm scalability and elasticity of virtual machines with our cooperative IaaS, i.e. the ability to scale (both horizontally and vertically), merge and split virtual machines. Cooperation calls are sent at runtime to notify each other ongoing situations and actions. Cooperation feature is activated for all *AmpManager* instances and the central *XenManager* in this experiment.

The virtual machines and quota allocation of all applications in this experiment are shown in figure 7.3. Each RUBiS *AmpManager* uses down calls to request for quota increase during the up ramp phase of its workload (950th, 1050th, 1450th, 1550th, etc second). Depending on the virtual machine placement and available resources on each PM in the time of those down calls, the *XenManager* either **vertically scales** an existing tier virtual machine (1050th, 1550th and 1750th second) or **horizontally scales** (allocates a new virtual machine) the associated tier for this *AmpManager* (950th, 1450th and 1650th second). In the later case, the *XenManager* notifies this allocation to the customer's *AmpManager*, so that a new application tier instance is deployed. In both cases, the customer's load balancer (*HttpHaProxies*) needs to be correctly configured to take into account each tier instance's weight. Similarly, the possibilities to decrease size for a tier's virtual machine or to remove a virtual machine are handled in the down ramp phase of the workload. The customer's *AmpManager* asks for quota decrease with its down calls. The *XenManager* then decides to **reduce size** (vertically scale) of a tier's virtual machine (2600th, 2900th and 3150th second) or to remove it (at 2650th and 2950th second).

Notice that at runtime, with the tier knowledge provided by the *AmpManagers*, the *XenManager* proposes to **merge small virtual machines into bigger ones**,

in attempt to reduce overhead. For example, a cooperative merge happens at 2650th second: the *XenManager* merges two virtual machines for application 2 (in PM1 and PM3) into one big virtual machine (in PM3). At the same time, another cooperative merge is executed between two virtual machines of application 3 (in PM1 and PM2) into a big virtual machine in PM1. Application 2 and 3 benefit with each having a single instance running in a big virtual machine (similar to Static). This performance benefit will be discussed in section 7.2.2. Additionally, after reducing quota for the virtual machine of application 2 at 2950th second, the IaaS migrates the virtual machine of application 1 from PM2 to PM3. It then turns PM2 off, and the provider benefits in energy saving. This resource optimization will be further analyzed in section 7.2.4.

7.2.2 Performance Overhead

After confirming that our cooperative IaaS handles scalability of elasticity for the customer's virtual machine, we evaluate the benefit to reduce performance overhead by the cooperative resource management system. Like described at the beginning of this section (7.2), Static placement can be considered as "Ideal" placement of virtual machine in terms of performance (with only one big virtual machine per server). As a result, performance overhead for each scenario is evaluated as the difference of application response between the concerning scenario and Static. We claimed in section 3.1 that (1) performance overhead is generated by the hypervisor, the virtualized I/O layer and the balancer; and (2) overhead can be lowered by reducing the number of virtual machines of the same application tier (MySQL in our experiments) collocating on the same physical server.

To evaluate jCoop's performance benefit, we compare our cooperative IaaS with Static (lower bound of response time), SCO (upper bound of response time) and BLI (being used in conventional IaaS). Figure 7.4 compares the average response time of the third RUBiS application with the mentioned scenarios. This figure confirms the response time's upper and lower bound of Static and SCO, respectively. The response time of SCO during full load period is approximately 15%-20% higher than in Static, because of the overheads.

When the workload increases (1400th to 1750th second), jCoop in BLC deals with load peak with a virtual machine allocation at 1500th second (PM1 in figure 7.3). This action generates a peak response time (BLC curve, 1500th second in figure 7.4). After this allocation, jCoop then vertically scales this new virtual machine (PM1, 1750th second in figure 7.3), lowering the response time while the workload continues to rise.

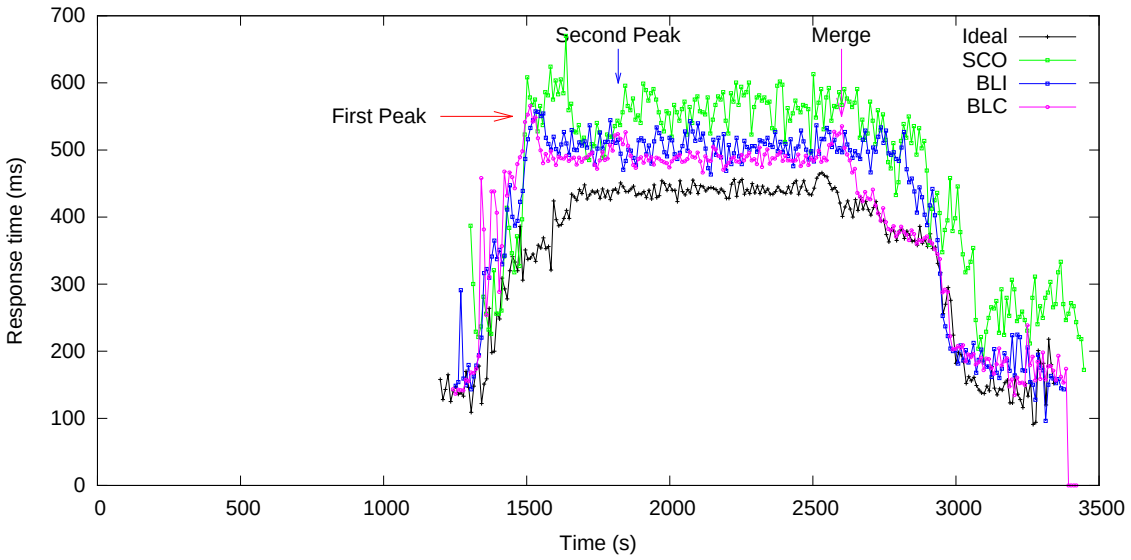

Figure 7.4: Response Time of Application 3

When compared with a static tier configuration (SCO), jCoop in BLC has more stable response time with elastic virtual machines: additional required resources can be added on-demand and instantly (1750th second in figure 7.3). Additionally, BLC does not suffer from virtual machine migration's overhead when dealing with load peak, unlike SCO (which has a virtual machine migration at 1750th second to deal with increasing load, therefore has increased response time – SCO curve, 1750th second in figure 7.4).

Comparing with a non-cooperative resource management system in a conventional IaaS (BLI), jTune in BLC has similar response time in the *up ramp* and *main session* phase (1200th to 2500th second in figure 7.4). However, the benefit of cooperation strategy appears in the *down ramp* phase: two small virtual machines of the application 3 is merged (2650th second in figure 7.3). After this merge, the whole PM1 is occupied by only one big virtual machine for application 3. This situation is similar to **Static**: only one virtual machine for each RUBiS application, each physical server hosts only one virtual machine (figure 7.5, from 2650th to 2950th second). As a result, response time of application 3, from 2650th onward, stays very close to Static's one ("ideal" performance). BLI does not have this merge, and therefore, has higher response time, up to 10-15%. This phase clearly shows the cooperative IaaS benefit in terms of performance optimization for the customer's application.

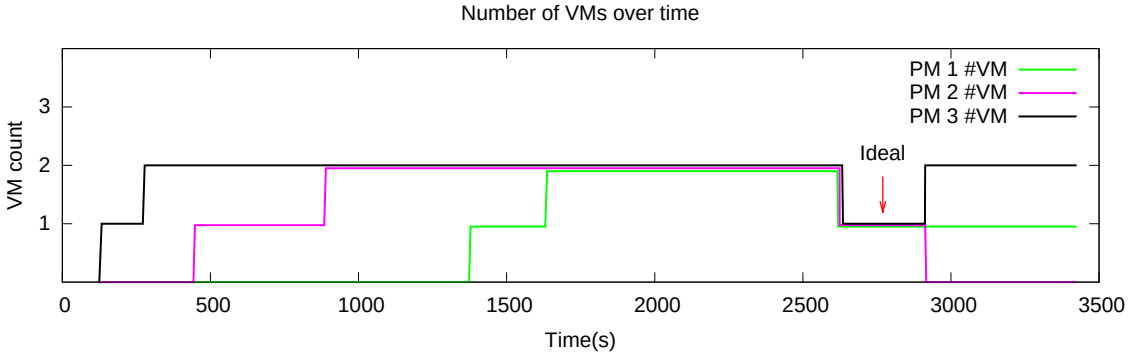

Figure 7.5: BLC: Virtual Machine Allocation on Physical Servers

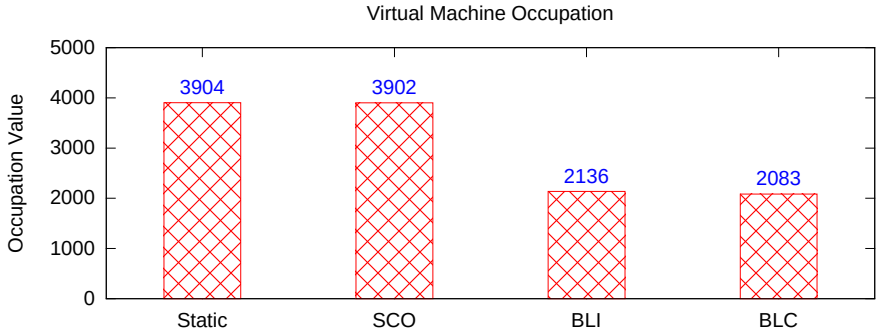

Figure 7.6: Comparison of Virtual Machine Occupation

7.2.3 Virtual Machine Occupation

In this section we measure the virtual machine occupation Ω among the defined scenario. Section 7.1.3 defines Ω as the metric to represent the effectiveness of resource allocation policy: the lower Ω, the less waste for *booked* virtual machine's resource. In other words, low Ω confirms the fine grain resources provided by the IaaS. The customer saves cost if the resource management policy provides low Ω in the experiment.

Figure 7.6 summarizes the calculated Ω for the defined scenarios. As can be seen from this figure, both Static and SCO have highest virtual machine occupation (around 4000): the customer's virtual machines are preallocated and not scaled at runtime. Static and SCO can be considered as the *upper bound for virtualized resource waste* in a IaaS.

BLI and jCoop in BLC have much better occupation rate, because the allocated virtual machines are either well used (actively loaded) or they will be removed by the *AmpManager* to improve utilization rate and to reduce cost for the customer. In our experiment, jCoop in BLC has better utilization rate than BLI, with $\Omega = 2083$

and $\Omega = 2136$, respectively. These values reflect the fact that jCoop's *XenManager* exploits elastic virtual machines and their ability to vertically scale.

Although BLC's total virtual machine size at runtime is quite similar to BLI in all phases, BLC's improvement over BLI in terms of saving virtualized resources is shown when the newly allocated virtual machine has smaller size than BLI's one (950th, 1450th and 1650th second in figure 7.3). In other words, BLC's virtual machine size has an **intermediate step** between unallocated and fully allocated ($^1/_3$ of a physical machine, respectively). This intermediate size reduces cost to the customer when compared with full size, while it still provides to the customer's enough resources to handle the incoming load. In fact, the number of intermediate steps can be increased in jCoop to have finer grain resource allocation. However, to avoid saturation of *XenManager* with too many quota allocation requests, we set minimum size for each quota change as $^1/_6$ of our node's total resource in our experiments with jCoop. These intermediate steps allow BLC to have potentially lower virtualized resource waste than BLI in real world's workloads.

In conclusion, our jCoop in BLC has the least virtualized resource waste (i.e. lowest Ω) with the cooperation and elastic virtual machines. Static and SCO are the two most wasted resource management policy for the customer's virtual machines.

7.2.4 Physical Server Utilization

In this section, we analyze the defined scenarios' capabilities to minimize physical resource usage. We use the total PM utilization time to measure the hardware resource usage. The higher PM utilization time, the more energy the provider needs to spend. The comparison is shown in figure 7.7.

Static is the worst case for physical server utilization: it occupies all servers at runtime and there is no migration. In contrast, SCO's benefit is confirmed: it minimizes the amount of physical machines being used for the provider (7419s) by packing as many virtual machines into as few physical servers as possible. However SCO is not widely used in a IaaS, because it does not have dynamic application sizing, saving cost for the customer and being able to handle load peaks.

Our jCoop in BLC saved average 5% of utilization time for all physical machines when compared with BLI (7611s and 8059s, respectively). This comparison shows jCoop's powersaving benefit with a resource management policy being widely used in conventional IaaS (BLI). Although we cannot reach the lower bound of server

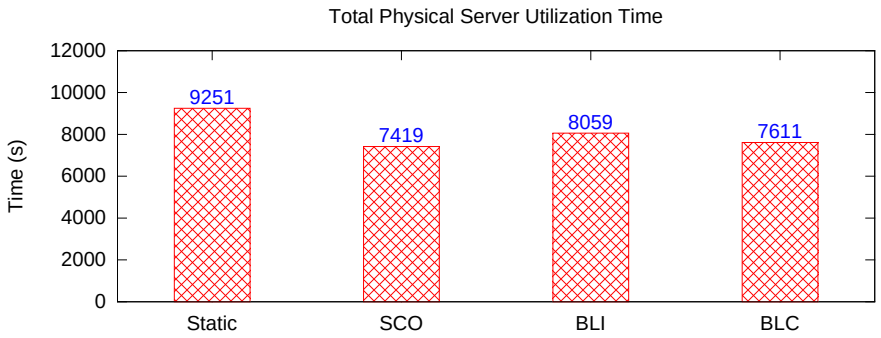

Figure 7.7: Comparison of Physical Server Utilization

utilization like SCO (7419s) but the trade-off of 2.5% server usage (BLC) is better than 10%-20% lower application performance (SCO and BLC, figure 7.4).

7.3 Synthesis

In this chapter, we justified the benefit of jCoop with the BLC scenario. When comparing our cooperative resource management policy with others, namely Static, SCO, and BLI, we showed that jCoop has certain advantages.

First, we showed that jCoop handles elastic virtual machines effectively. It allows *XenManager* to resize and propose cooperative merges to the *AmpManagers*. Second, jCoop has close-to-ideal performance (with response time similar to Static) after these merges are performed. This helps the customer to have better performance with less performance overhead. Third, jCoop improves virtualized resource usage by allowing the customer to have intermediate virtual machine sizes. This support also reduces cost for the customer, as he is provided with finer grain resource blocks. Finally, we achieved better physical resource utilization: physical server is less used. jCoop outperforms BLI (resource management policy being used in conventional IaaS) with all of these metrics.

Chapter 8

Conclusion and Perspectives

Contents

8.1 Conclusion 107

8.2 Perspectives 108

8.2.1 Short Term Perspectives 109

8.2.2 Long Term Perspectives 110

8.1 Conclusion

Recent years have observed the rise of cloud computing model: companies externalize their hardware resources to dedicated centers. This evolution comes from the increasing complexity of managing infrastructures. Essentially, cloud computing meets the needs of the customer with the provider's services. This dedication improves each actor's performance and more importantly, it saves costs for both actors.

Resource management is an important task that such model must ensure, in order to reduce costs: the customer only pays for the amount of resources being really used, and the provider optimizes his physical resource usage while sharing his resource pool between the customers. The complexity of the provider's infrastructure raises the problem of resource management with both virtualized and physical resources (virtual machines and physical servers). Managing these resources at the same time is an error prone task. To solve this problem, autonomic administration systems are implemented on both sides. These systems automatically monitor the application and the infrastructure and react with events (e.g. load fluctuation) at

runtime. In traditional cloud infrastructures, these management systems are usually not coordinated. Additionally, the lack of elastic virtual machines prevents them from having optimal resource utilization, causing performance overhead and resource holes. A naive solution is to provide elastic virtual machines to the customer. However, elastic virtual machine alone, without the cooperation from both actors, does not fully optimize resource usage and reduce performance overhead.

From this point of view, we proposed a direction to use the combination of cooperative resource management with elastic virtual machine. This cooperative IaaS brings the two resource managers (*IaaSManager* and *AppManager*) closer. Information about the customer's application (e.g. tier instances) is shared between the two management layers. With the provided application information, the provider can better optimize his infrastructure and reduce performance overhead. In a cooperative IaaS, the decision to allocate or deallocate virtual machines is shifted from the customer to the provider. The customer is only responsible for evaluating the total resource needed as a whole for his application. Virtual machine distribution and placement tasks are handled by the provider. Our proposed cooperative IaaS can be considered from different perspectives: an extension of PaaS, a hybrid IaaS-PaaS model, or distributed virtual machines.

We confirmed the benefit of our cooperative resource management policy using a set of experiments. We compared our work with different scenario (upper bound, lower bound, and actual resource management policy in conventional IaaS) in terms of both performance and resource utilization. The evaluations showed that our cooperative IaaS outperforms traditional two-level, non-cooperative resource management in current IaaS (BLI) with: (1) lower performance overhead (better response time for the customer's application), (2) better virtual machine usage (reducing cost for the customer with finer grain resource blocks), and (3) better physical resource usage (reducing energy and cost for the provider).

8.2 Perspectives

Although the evaluation shows merits of the cooperative IaaS, our work still has a wide range of improvements. We identified several potential directions that can extend the effectiveness and usability of a cooperative IaaS. These directions can be classified into main categories: short term works to improve jCoop and long term works to have wider adoption of cooperative resource management in the cloud.

8.2.1 Short Term Perspectives

Algorithms

jCoop is our first prototype to evaluate the benefit of cooperative resource management in the cloud. One of the most important component in jCoop is *XenManager*. It uses a derivation of *Best-Fit-Decreasing* algorithm to distribute and migrate virtual machines among physical server in the provider's resource pool. While showing good results, this algorithm can be improved to have better virtual machine placement, taking into account knowledge about other tiers and migration costs (jCoop does not yet consider migration cost while making decision to consolidate physical servers).

Additionally, our proposed algorithms do not take into account the overhead of splitting: the customer has two virtual machines instead of one. Performance overhead would be increased in this situation. Therefore, the algorithms should take into account providing extra quota for the customer to counter balance this overhead.

Fault Tolerance

As an early prototype, jCoop does not take into account unexpected failures that may happen at runtime. These failures include application crashes, hardware problems, etc. Because of possible sfailures, our cooperation protocol may not work properly. For example, the proposal to merge virtual machines (`ET_UPCALL_MERGE_VM`) may not reach the destination (customer's *AppManager*) when the network link between the *IaaSManager* and the *AppManager* is down. As a result, the proposal is not accepted, tier instances are not merged, and the application performance is not improved. We need to enhance jCoop's reliability to handle failures at runtime.

Real Scenarios

In chapter 7, we evaluated our work using synthesis benchmark with RUBiS client and Apache / MySQL / PHP stack in our private IaaS. We believe that our work can show benefit in real conditions: real multi-tier applications with real workloads in real cloud infrastructures. We are currently in progress of evaluating the proposed cooperative resource management system with real workload traces provided by Eolas[1] datacenter. Another source of workload trace is *The Grid Workloads Archive*[38]. Using these data, we can conduct trace-based simulations of the

[1] http://www.businessdecision-eolas.com

customer's applications in a cooperative IaaS, evaluate the performance gain and resource optimization.

Additionally, we think that the experimented resource step of $^1/_6$ of a physical machine in BLC is rather coarse. We believe with finer grain resource steps (e.g. $^1/_{10}$ or $^1/_{20}$ size of a physical machine) will bring more benefits to both sides at runtime: the customer's resource allocation is closer to what is really needed, and the provider has less total physical utilization time. We aim to continue our experiments with these settings in order to confirm the benefits.

8.2.2 Long Term Perspectives

In a longer term, we aim to improve the easy adoption of cooperative resource management in the cloud.

Resource Dimensions

First, the cooperative resource management policy should be able to take into account more types of elastic resources. In our algorithms, we currently consider only two major resource dimensions for elastic virtual machines: CPU and memory. The two missing dimensions are storage and network bandwidth. Introducing more dimensions to manage increases the complexity and performance of our virtual machine packing and quota changing algorithms, but also provides more flexibility to the customer when managing his applications at runtime.

Application Domains

In this thesis, we considered only the case of multi-tier applications in the cloud. In real world clouds, the customer can work with various application domains. Each application domain has different effectiveness when used with a cooperative IaaS. For example, in a parallel batch processing application (with MapReduce [29] jobs), the virtual machines do not need to be merged or split because they maximize their utilization of allocated resources most of the time (e.g. their vCPUs are at maximum load).

Cooperative PaaS

Finally, we wonder if the cooperation between the customers and providers can be exploited in a PaaS environment. Most PaaS is built on top of IaaS (public or hybrid IaaS). In the case of using public IaaS, there exists three actors: PaaS customer (who

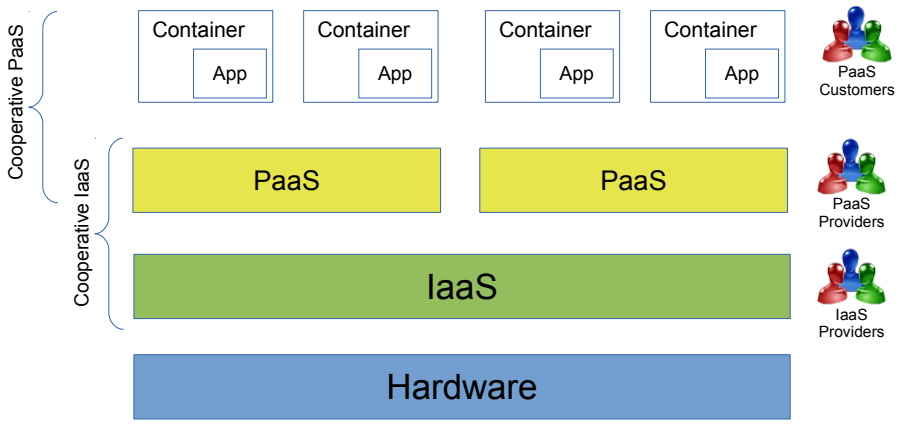

Figure 8.1: Cooperative PaaS on top of Cooperative IaaS

develops his application on the PaaS), PaaS provider, and IaaS provider (figure 8.1). From this point of view, cooperative PaaS can be considered to have two cooperative resource management policies: PaaS customer – PaaS provider, and PaaS provider – IaaS provider. The later is similar to what was described in this thesis: the PaaS provider uses the IaaS, therefore is a IaaS customer.

In this context, the cooperation between the PaaS customer and the PaaS provider is an interesting direction that can be further explored. For instance, in a traditional J2EE scenario, the PaaS provider can propose to collocate servlets from various PaaS customer's applications into the same container, to reduce the number of virtual machines serving these containers and reduce performance overheads. A deeper investigation of this cooperation may bring benefit to both the PaaS customer (in terms of performance) and the PaaS provider (in terms of virtualized resource usage).

Bibliography

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/. Accessed December/2013.

[2] AWS Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/. Accessed December/2013.

[3] Google App Engine Cloud Platform. https://cloud.google.com/products/. Accessed December/2013.

[4] Google Apps for Business. http://www.google.com/apps/. Accessed December/2013.

[5] Google Compute Engine. https://cloud.google.com/products/compute-engine. Accessed December/2013.

[6] Google Documents. https://docs.google.com. Accessed December/2013.

[7] HP Public Cloud. http://www.hpcloud.com/. Accessed December/2013.

[8] IBM SmartCloud Application Services. http://www.ibm.com/cloud-computing/us/en/paas.html. Accessed December/2013.

[9] Microsoft Office 365. http://office.microsoft.com/. Accessed December/2013.

[10] OpenVZ. http://openvz.org/. Accessed December/2013.

[11] VMware. http://vmware.com/. Accessed December/2013.

[12] Windows Azure: Microsoft's Cloud Platform. http://www.windowsazure.com/en-us/.

[13] Adams, K. and Agesen, O. (2006). A comparison of software and hardware techniques for x86 virtualization. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 2–13. ACM.

[14] Amza, C., Chanda, A., Cox, A., Elnikety, S., Gil, R., Rajamani, K., Zwaenepoel, W., Cecchet, E., and Marguerite, J. (2002). Specification and implementation of dynamic Web site benchmarks. In *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 3–13.

[15] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A View of Cloud Computing. *Commun. ACM*, 53(4):50–58.

[16] ASHRAE (2012). *Datacom Equipment Power Trends and Cooling Applications, 2nd Edition*. ASHRAE, 2nd edition.

[17] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). 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. ACM.

[18] Belady, L. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101.

[19] Ben-Yehuda, O. A., Ben-Yehuda, M., Schuster, A., and Tsafrir, D. (2012). The resource-as-a-service (RaaS) cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 12–12. USENIX Association.

[20] Bobroff, N., Kochut, A., and Beaty, K. (2007). Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE.

[21] Bouchenak, S., Boyer, F., Hagimont, D., Krakowiak, S., De Palma, N., Quema, V., and Stefani, J.-B. (2005). Architecture-Based Autonomous Repair Management: Application to J2EE Clusters. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 369–370.

[22] Broto, L., Hagimont, D., Stolf, P., Depalma, N., and Temate, S. (2008). Autonomic management policy specification in Tune. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1658–1663, New York, NY, USA. ACM.

[23] Buyya, R., Yeo, C. S., and Venugopal, S. (2008). Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. In *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, pages 5–13.

[24] Chaisiri, S., Lee, B.-S., and Niyato, D. (2009). Optimal virtual machine placement across multiple cloud providers. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 103–110. IEEE.

[25] Chieu, T., Mohindra, A., Karve, A., and Segal, A. (2009). Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. In *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, pages 281–286.

[26] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Proceedings of the*

2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2. USENIX Association.

[27] Dawoud, W., Takouna, I., and Meinel, C. (2011). Elastic vm for cloud resources provisioning optimization. In *Advances in Computing and Communications*, pages 431–445. Springer.

[28] Dawoud, W., Takouna, I., and Meinel, C. (2012). Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning. In Krishna, P., Babu, M., and Ariwa, E., editors, *Global Trends in Computing and Communication Systems*, volume 269 of *Communications in Computer and Information Science*, pages 11–25. Springer Berlin Heidelberg.

[29] Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113.

[30] Duong, T. N. B., Li, X., and Goh, R. (2011). A Framework for Dynamic Resource Provisioning and Adaptation in IaaS Clouds. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 312–319.

[31] Dutta, S., Gera, S., Verma, A., and Viswanathan, B. (2012). SmartScale: Automatic Application Scaling in Enterprise Clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 221–228.

[32] El Rheddane, A., De Palma, N., Boyer, F., Dumont, F., Menaud, J.-M., and Tchana, A. (2013). Dynamic Scalability of a Consolidation Service. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 748–754, Washington, DC, USA. IEEE Computer Society.

[33] Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee.

[34] Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica, I. (2009). Above the clouds: A Berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28.

[35] Ganapathi, A., Chen, Y., Fox, A., Katz, R., and Patterson, D. (2010). Statistics-driven workload modeling for the Cloud. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 87–92.

[36] Hermenier, F., Lorca, X., Menaud, J.-M., Muller, G., and Lawall, J. (2009). Entropy: A Consolidation Manager for Clusters. In *Proceedings of the 2009*

ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '09, pages 41–50, New York, NY, USA. ACM.

[37] Hwang, I. and Pedram, M. (2013). Hierarchical Virtual Machine Consolidation in a Cloud Computing System. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 196–203, Washington, DC, USA. IEEE Computer Society.

[38] Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., and Epema, D. H. (2008). The Grid Workloads Archive. *Future Generation Computer Systems*, 24(7):672 – 686.

[39] Jayasinghe, D., Pu, C., Eilam, T., Steinder, M., Whally, I., and Snible, E. (2011). Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 72–79. IEEE.

[40] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

[41] Konstanteli, K., Cucinotta, T., Psychas, K., and Varvarigou, T. (2012). Admission Control for Elastic Cloud Services. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 41–48.

[42] Mell, P. and Grance, T. (2011). The NIST definition of cloud computing (draft). *NIST special publication*, 800(145):7.

[43] Nakada, H., Hirofuchi, T., Ogawa, H., and Itoh, S. (2009). Toward virtual machine packing optimization based on genetic algorithm. In *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, pages 651–654. Springer.

[44] Nguyen Van, H., Dang Tran, F., and Menaud, J.-M. (2009). Autonomic Virtual Resource Management for Service Hosting Platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 1–8, Washington, DC, USA. IEEE Computer Society.

[45] Piao, J. T. and Yan, J. (2010). A network-aware virtual machine placement and migration approach in cloud computing. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pages 87–92. IEEE.

[46] Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421.

[47] Quiroz, A., Kim, H., Parashar, M., Gnanasambandam, N., and Sharma, N. (2009). Towards autonomic workload provisioning for enterprise grids and clouds.

In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 50–57. IEEE.

[48] Rasmussen, N. (2011). Determining Total Cost of Ownership for Data Center and Network Room Infrastructure.

[49] STANDARDIZATION, C. (2011). The problem with cloud-computing standardization.

[50] Vaquero, L. M., Rodero-Merino, L., and Buyya, R. (2011). Dynamically Scaling Applications in the Cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52.

[51] Villegas, D., Antoniou, A., Sadjadi, S., and Iosup, A. (2012). An Analysis of Provisioning and Allocation Policies for Infrastructure-as-a-Service Clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 612–619.

[52] Winkler, V. (2011). *Securing the Cloud: Cloud Computer Security Techniques and Tactics*. Elsevier Science.

[53] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. (2013). Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE.

[54] Yazdanov, L. and Fetzer, C. (2013). VScaler: Autonomic Virtual Machine Scaling. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 212–219, Washington, DC, USA. IEEE Computer Society.

[55] Zhan, J., Wang, L., Li, X., Shi, W., Weng, C., Zhang, W., and Zang, X. (2013). Cost-Aware Cooperative Resource Provisioning for Heterogeneous Workloads in Data Centers. *Computers, IEEE Transactions on*, 62(11):2155–2168.

[56] Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18.

List of Figures

2.1 Components of each Cloud Service Model 8
2.2 Cloud Service Models in Vertical Stack 9
2.3 Classification based on Deployment Model 10
2.4 Bare Metal Hypervisor *(Type 1)* 15
2.5 Hosted Hypervisor *(Type 2)* 16
2.6 Full Virtualization 18
2.7 Paravirtualization 18
2.8 Operating System Level Virtualization 19

3.1 Typical J2EE Architecture 26
3.2 Example of Flexibility from Small Virtual Machines 28
3.3 Dealing with Load Increase 29
3.4 Finer-Grain Resource of Elastic Virtual Machine 33
3.5 Multiple Instances of the same Application running on the same PM after a Migration 34
3.6 Traditional IaaS 36
3.7 Cooperative IaaS: Knowledge of each Cloud Actor 37
3.8 Communication Channel in a Cooperative IaaS 39
3.9 Comparison of a cooperative IaaS with traditional IaaS and PaaS . . 41

4.1 Relationship between Local Decision Module and Global Decision Module [44] 53

5.1 Example of Actions for Subscription and Add Resources 62
5.2 Example of Actions for Merging and Splitting Virtual Machines . . . 64
5.3 Detail of a Cooperation Call 70

6.1 jTune's SR 78
6.2 jTune's Main Components in the Control Loop 81
6.3 Relation of *Policies*, *Wrappers* and *RemoteWrappers* 81
6.4 Event Flows in jTune's Architecture 83
6.5 Application Lifecycle in an Autonomic Manager 83
6.6 IaaSManager and AppManager in jCoop 87
6.7 XenManager's Components in jCoop 88
6.8 AmpManager's Components in jCoop 92

7.1 Our Private IaaS Cloud 97
7.2 Generated Workload 99
7.3 BLC: Virtual Machine Placement and Quota Distribution 101
7.4 Response Time of Application 3 103
7.5 BLC: Virtual Machine Allocation on Physical Servers 104
7.6 Comparison of Virtual Machine Occupation 104
7.7 Comparison of Physical Server Utilization 106

8.1 Cooperative PaaS on top of Cooperative IaaS 111