Market-based autonomous and elastic application execution on clouds

Stefania Costache

N° d'ordre : x ANNÉE 2013

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

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Stefania Victoria Costache

préparée à l'unité de recherche 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

Market-based Autonomous Resource and Application Management in the Cloud

Thèse à soutenir à Rennes
le juillet 2013

devant le jury composé de :

Kate Keahey / **Rapporteur**
Scientist, Argonne National Laboratory

Christian Perez / **Rapporteur**
Directeur de Recherche, INRIA Grenoble Rhone-Alpes

Franck Cappello / **Examinateur**
Senior Scientist, Argonne National Laboratory

Rosa M. Badia / **Examinateur**
Scientific Researcher, Barcelona Supercomputing Center

Jean-Marc Pierson / **Examinateur**
Professeur, Université Paul Sabatier

Christine Morin / **Directeur de thèse**
Directrice de Recherche, INRIA Rennes - Bretagne Atlantique

Samuel Kortas / **Co-directeur de thèse**
Ingénieur de Recherche, EDF R&D

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

The journey of a thousand miles must begin with a single step.

Chinese proverb

Acknowledgments

This thesis is a result of a three-year journey, which would not have been possible without the involvement of multiple people.

First, I would like to thank to the members of my jury, who took the time to evaluate my work. I thank Christian Perez and Kate Keahey for accepting to review my work and for their thorough and to the point comments. I thank Rosa Badia and Franck Cappello for their interest in my work. Finally, I thank Jean-Marc Pierson for accepting to be the president of my jury.

To my advisors, I thank them for the guidance they provided me during these three years. Christine, first I thank you for welcoming me in the Myriads team. I thank you for the support and enthusiasm you showed me and for the freedom I had to take decisions and discover my own rhythm in which I can pursue my work. Samuel, I thank you for your flexibility, continuous enthusiasm and for all the support you provided during my stay at EDF. Nikos, I thank you for pointing out challenging directions in my work.

I would also like to thank to the people that are or were a part of the Myriads team since my arrival. I thank you for the lunches, breaks and "after-work" gatherings that we shared and for some of the nice moments that I had during these three years. Maryse, I thank you for making light in the administrative hurdle. Anca, I thank you for your endless enthusiasm and also for all the good and bad movies that we got to see ;). I also own you for teaching me how to ride a bike and for joining me in many of the nice bike trips that I got to do afterwards. Finally, I thank to the small group that joined the Longchamp Saturday lunches, Anca, Mihai, Sajith, which pulled me out of my cave while I was writing this manuscript.

To the SINETICS I2D group from EDF, Clamart, I thank them for making me feel "like at home" each time I got there. I thank David for welcoming me in the team. I also thank Ghyslaine for providing a nice place to come to from work during my months spent at EDF. I enjoyed the dinners that we shared and your culinary skills.

Thomas, I thank you for the careful guidance during my master internship and for start that you provided me with. Your advices and thorough "to the point" comments regarding how to write a research paper set me on the right foot when I started this thesis.

I thank the people that I got the chance to work with at Politehnica University of Bucharest and at University of Groningen, people that, through their determination, passion and enthusiasm for research, influenced my career path and inspired me to do a PhD.

Vidya, thank you for sharing a small part of this journey. Working with you was refreshing.

To my friends, I thank you for always finding the patience to listen and cheer me up. Sorin, Diana, Ancuta, I thank you for the rich discussions we had. Andreea, I thank you for being a wonderful host and offering me a warm and welcoming place to stay every time I came to Paris. Monica, I thank you for not letting me forget my hobbies and for encouraging me with a pragmatic and objective attitude. Florin, I thank you for your continuous support during the many years that went since we met. Your sometimes ironic and to the point attitude was refreshing and kept my feet on the ground.

To my parents, I thank you for all the endless support, encouragement and help that I got from you during all the years that past by. You always found a way to make me smile, even in the worse and most difficult moments. Finally, I thank you for the effort you made to come to Rennes almost every year, and especially at my defense.

Contents

List of Figures

Introduction

High Performance Computing (HPC) infrastructures are used to execute increasingly complex and dynamic applications. An organization owning an HPC infrastructure may have different application types with a variety of architectures, requiring different software configurations or specific frameworks.

Electricité de France (EDF) is a typical example of an HPC organization. As the world's largest producer of electricity, EDF has relied on HPC numerical simulation for more than 30 years. Simulation results are used to optimize the day-to-day production or choose the safest and most effective configurations for nuclear refuelling. On a longer term, these results can be used to explain complex physical phenomena, to better schedule the maintenance process or to assess the impact of new vendor technologies. These numerical simulations are performed in a wide variety of fields. Some examples are neutronics, fluid dynamics, mechanics or thermodynamics studies. A part of them consists of tightly coupled codes running with a fixed set of processes, while others are composed of thousands of independent tasks, running the same code with different parameters (bag of tasks). To predict the energy demand of users over the next time periods, large data sets might be analyzed. Such studies are performed with specific cluster computing frameworks (e.g., MapReduce) which use their own scheduling mechanisms. As these studies need to be performed in a secure context, EDF is restricted to use private clusters.

As programming models and user requirements evolve, developed applications have also more dynamic resource demands. As an example, let us take a computational fluid dynamics simulation, often used to analyze the flow of a fluid in certain configurations. When the right conditions are met, turbulence appears in the fluid and, to simulate it, an additional set of equations are added to the whole model. From that moment the computation is more intensive and the application could make use of more resources, e.g., CPU cores, to improve the computation speed. When the turbulence disappears, i.e., the fluid reaches a steady state, the computation amount decreases, making the acquired number of resources inefficient due to communication costs. Ideally, the application could release them. Other applications can also take advantage of the infrastructure's variable resource availability: bag-of-tasks applications [40] can expand on all the available resources when the infrastructure is under-utilized, and shrink when the user demand increases.

Besides the dynamicity of application resource demand, another problem is the exigence of the users, in terms of ease in the configuration and availability of their software environments or performance objectives, also known as Service Level Objectives (SLOs). The studies performed by EDF require various software libraries. Managing the software configuration for each of them on all the physical nodes is a difficult task for an administrator. Moreover, users might have different performance objectives for their applications. Some users want the application results by a specific deadline, (e.g., a user needs to send her manager the output of a simulation by 7am the next day) or as soon as possible (e.g., a developer wants to test a newly developed algorithm). Other users want to express simple

application execution policies like: "*I want to finish my application when a certain performance condition is met*", or "*I want to add a visualization module to my application so I need more compute power now*".

Even if virtualizing the infrastructure and transforming it in a private cloud is seen as an attractive solution to manage the infrastructure resources, it does not completely address the previously mentioned requirements. Making the infrastructure a private cloud has several advantages. One advantage comes from letting users customize by themselves the environments in which their applications run. A second advantage is that resources are time-shared, leading to an increased resource utilization. A third advantage is that applications can provision virtual resources, or virtual machines, dynamically during their runtime. On top of this virtualized infrastructure, recently developed software platforms provide users with support to develop and run their applications with no concerns regarding infrastructure's resource management complexities (also called Platform-as-a-Service). These solutions, however, provide *limited support* for users to express or use various resource management policies and they *don't* provide any support for application performance objectives.

Finally, as the infrastructure capacity is limited, the real challenge lies in how to share the resources between the user applications. This would not be a problem if the infrastructure capacity were enough for all user requests in the highest demand periods. Nevertheless, this is rarely the case, as expanding the resource pool is expensive. Thus, it is preferable to solve the contention periods when they appear, around deadlines (e.g., conference or project deliverable), rather than acquiring more resources to satisfy this increase in demand. Solving the contention periods is complex as the infrastructure administrator needs to manage hundreds of users,developers working on internal code, or scientific researchers using validated software. We cannot assume that all these users are cooperative or they want to use resources only when they most urgently require to do so. Current systems address this issue through limited resource regulation mechanisms. Usually, they use priorities, or weights, to define the importance of users and their applications. More advanced mechanisms allow administrators to define quotas for resources used by a user. However, such systems do not provide the social incentives for users to adapt their application resource demand in a way that benefits all the infrastructure occupants.

Problem Statement

In the face of increasing application complexity, dynamicity of resource demand and variety of user requirements, it is challenging to design resource management systems that provide efficient and fair resource allocations while offering flexible adaptation support to applications. By resource management system, or platform, we understand the software that manages the compute resources and multiplexes the access of applications to them. By flexible application adaptation we understand the ease to support multiple user objectives and diverse application resource demands. Efficient resource allocations are obtained by maximizing the aggregate usage of physical resources. Finally, fairness is achieved by limiting the resource usage of applications based on the real importance, or value that users have for them.

Requirements

Here we define a set of requirements that must be met in the design of a resource management platform.

Flexible SLO support

Users from a scientific organization have a variety of requirements in terms of what types of applications should be executed on the infrastructure and what performance guarantees or access rights should be ensured for them. To meet all these requirements with minimum user effort, one should design mechanisms which automatically adapt applications together with their resource demands to user-defined objectives. Such mechanisms should be flexible enough to be used for a variety of applications and user-defined objectives. The provided level of flexiblity depends on four characteristics: (i) how the resource control is distributed between applications and the resource management system; a resource management system that gives resource control to applications is more flexible as supports many resource demand adaptation policies; (ii) if applications can change the allocated resource amount during their execution; malleable and evolving applications, for example, can achieve better performance if allowed to change their resource allocations; (iii) how does the resource management system provides SLO support to its users; such support might include monitoring and scaling the application during its runtime; (iv) and what is the customization and isolation level of the environment in which users run their applications.

Maximum resource utilization

When providing concurrent access to resources there are two ways of sharing resources: (i) coarse-grained; (ii) and fine-grained. When providing coarse-grained resource allocation, either applications have exclusive access to the physical nodes, or they run in virtual machines with a configuration selected from a specific class, as in the case of Amazon EC2 [7]. This method of allocating resources leads to poor resource utilization, as while some applications don't use all their provisioned resources, other applications might wait in queues. With fine-grained resource allocation, applications can request arbitrary resource amounts and the infrastructure utilization is improved. Thus, mechanisms should be designed to provide fine-grained resource allocation.

Fair resource utilization

An organization usually disposes of an infrastructure that needs to be shared between users coming from different departments and having different values regarding when to run their applications. Regulating the access of these users to resources can be a complex task for an administrator, as she would need to decide some fair criteria for who gets access to resources. One criteria would be to assign priorities to users. Nevertheless, even if the administrator decides the user priorities, she cannot decide what application is the most urgent for the user to execute and when it should be executed. For example, in a computing center, the infrastructure may become overloaded around important conference deadlines, as scientific users might want to use resources to get results in time for their simulations. In this case, a user with a high priority might abuse her rights and run a less urgent simulation, taking resources from users who really need them. We cannot assume that all users will be cooperative and yield the access to resources when asked by other users. Thus, mechanisms should be designed to ensure a fair resource distribution by giving resources to users who truthfully value them the most. Such fair resource distribution can be ensured by forcing users to communicate the value they have for their applications.

Approach and Contributions

The goal of this dissertation is to *design, implement and evaluate a platform for application and resource management* that meets the previously stated requirements.

Our approach relies on the use of a proportional-share market [100] to allocate resources to applications and users. To buy resources, users are assigned rechargeable budgets by a banking service in the form of a virtual currency, from which they delegate a part to each of their running applications. Each application runs in an autonomous virtual environment that adapts individually its resource demand to the application's needs, with no knowledge regarding other participants. The only mechanism that binds these applications is the resource price, which varies with the total resource demand. Thus, the resource management is decentralized, allowing users to define their own resource demand adaptation policies. As it offers to each application the ability to control its own resource allocation, this resource management model can support a variety of applications, each with its own performance objective and possibly its own resource management algorithms. Moreover, the proportional-share market allows applications to run with fine-grained resource allocations, maximizing the infrastructure resource utilization. Finally, due to the budget constraints, users take "better" resource provisioning decisions. Thus, the market mechanism leads to a socially efficient user behavior as resources are fairly distributed between them: contention periods are solved efficiently by shifting a part of the demand from high utilization to low utilization periods.

To support this approach we designed a proportional-share resource allocation model to compute the amounts of CPU and memory to which each VM is entitled to, ensuring fine-grained resource allocation among applications. Then, we proposed two policies which receive a user-defined SLO and budget and adapt the application resource demand to the current resource availability and resource price: (i) a policy that adapts the resource demand (vertical scaling); (ii) a policy that adapts the number of VMs (horizontal scaling). These policies can be combined for better application performance.

We implemented the proposed resource allocation model and application adaptation policies in a software stack. We designed a software stack that is generic and extensible, allowing users to automate the application deployment and management process on virtualized infrastructures. Users have the flexibility to control how many resources are allocated to their applications and to define their own resource demand adaptation policies.

We have implemented and evaluated our approach in two contexts:

- A simulator, called Themis. We used Themis to evaluate the performance of the proportional-share market and the vertical scaling policy for different application SLOs. The obtained results have shown the resource management decentralization cost.

- A real-world prototype, called Merkat. We deployed Merkat on Grid'5000 and used it to test our previously mentioned policies. We used the vertical scaling policy to scale the resource demand of a static MPI application and the horizontal scaling policy to scale two task processing frameworks. We also used Merkat to run multiple static MPI applications that adapted their resource demand to a user-given deadline. The obtained results have shown the performance of Merkat when applications adapt selfishly.

Dissertation Organization

This dissertation is organized as follows: Chapter 1 describes the main application models and the organization of distributed systems, concepts which are important in understanding the applicability of our contributions. Chapter 2 covers the state of art. We present the important resource management approaches which are close to meeting our requirements. Chapter 3 gives a brief overview of our contribution. We present our approach and explain our design decisions. Chapter 4 provides a zoom on the resource management mechanisms implemented in our platform. We describe how applications can adapt their resource demand dynamically and how resources are managed on the infrastructure.

Chapter 5 presents the simulation results regarding the performance in large systems. Chapter 6 discusses the applicability of our approach in a real environment by validating it with several use cases of scientific applications. We conclude the manuscript in a chapter that summarizes our contribution and discusses possible future research directions.

Chapter 1

Distributed Computing for Scientific Applications

This chapter gives an insight on the requirements of scientific applications and the different infrastructure architectures. High Performance Computing (HPC) infrastructures are used to execute increasingly complex and dynamic scientific applications. Designing such applications is required by research fields like molecular dynamics, astronomy, aeronautics, physics, in which in the research process simulation is often preferred to real experimentation. To provide results in useful time intervals, such simulations need large amounts of computing power, which in turn leads to the design of infrastructures capable to provide it. A lot of research efforts were invested in the design of such infrastructures and, thus, different architectures exist, from supercomputers, clusters, and grids to clouds.

This chapter is organized as follows. Section 1.1 gives a taxonomy of scientific applications while Section 1.2 gives an overview of commonly encountered infrastructure types. Section 1.3 describes the current cloud technology, virtualization. Section 1.4 describes the cloud computing paradigm: it introduces the service and deployment cloud models and, to better understand the internals of our platform, it describes how virtual resources are managed in clouds.

1.1 Taxonomy of Scientific Applications

In this section we give an overview of the scientific application domain. Scientific applications are usually distributed programs, composed of many tasks, or processes. These tasks can be created at the beginning of the program or dynamically during its runtime. We describe the characteristics of these applications by classifying them based on their communication patterns, resource use and on how users interact with them.

1.1.1 Communication Requirements

Based on the communication amount between their processes, scientific applications can be classified as: tightly-coupled or loosely-coupled applications. Let us describe them next:

Tightly coupled applications are applications in which tasks communicate frequently.

Loosely coupled applications are applications for which the communication time does not represent an important part of their execution.

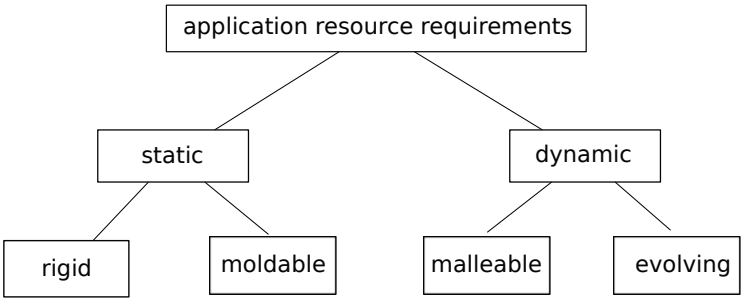


Figure 1.1: Application resource requirements.

1.1.2 Resource Requirements

Scientific applications might have different resource requirements in terms of CPU cores, or memory, which may or may not evolve in time. Figure 1.1 illustrates a classification of applications based on their resource requirements. Based on how their resource requirements can evolve during their execution, applications can be differentiated in *static* and *dynamic*. Static applications require a set of resources, e.g., CPU cores, that remain constant during their execution. Dynamic applications may change their resource demand during their execution. To further classify applications, Feitelson [66] introduced a taxonomy based on four application types: rigid, moldable, malleable and evolving. Rigid and moldable applications are static while malleable and evolving applications are dynamic. Let us explain these types next:

Rigid applications are applications for which the user can only specify a fixed number or resources at submission time (e.g., 4 CPU cores).

Moldable applications are applications for which the user can specify different resource configurations at submission time (e.g., 4, 8, or 16 CPU cores). A resource manager selects the most convenient one when the application is selected for execution. Examples of rigid or moldable applications are tightly coupled applications, as they are usually composed of a fixed number of processes, e.g., simulation codes like Gromacs [82] or Code_Saturne [48].

Malleable applications are applications which can adapt to changes in their resource allocation. The change in resource allocation can be done by an external entity, i.e., the infrastructure scheduler, based on the resource availability on the infrastructure. If resources need to be reallocated to other infrastructure occupants, the application can shrink and when resources become available again the application can expand. These applications can easily manipulate their resource demand as they mostly rely on a master-worker architecture: a master process submits tasks to workers and waits for them to be processed. To shrink the application, the master can suspend or stop executing some of their tasks and then shut down the workers. If the task execution time is small, stopping and resubmitting the task does not have a large impact on the application execution time. To expand the application it is sufficient to create more workers. Examples or malleable applications are *Bag of Tasks* [40], composed of many tasks that need to process different data sets. A special class of Bag of Tasks is represented by *Parameter Sweep* [31] applications: the same task is run with different parameter values taken from a range. Another example consists in *MapReduce* applications, developed using the MapReduce programming model [56], i.e., a model relying on the map and reduce paradigm from functional programming.

Evolving applications are applications that may need to change their resource demand during their execution, due to changes in their internal state. Such changes, initiated by the application, might be driven by the changing complexity of the computation or by different application phases having different resource demands. An example of evolving application is the computational fluid dynamics simulation with turbulence models [48], discussed in the Introduction chapter.

Malleable and evolving applications Some applications can be both malleable and evolving. Examples of such applications are workflows [107]. An workflow is composed of multiple tasks, or even distributed applications, which might have dependencies between them, requiring a certain execution order. Thus, a workflow has multiple stages, each of them requiring different resource amounts depending on the number of tasks that need to be executed at that specific stage. The total number of tasks and their execution order can be known from the beginning of the execution or it can change in time as each workflow stage might have a different execution branch depending of the output of the previous stage.

1.1.3 Interactivity

Based on the level of interaction between the user and her application, we split applications in two classes:

Interactive applications require frequent interaction from the user, e.g., visualization software.

Passive applications only receive an input from the user at their submission, perform a computation and return the result at the end with no other interaction, e.g., simulation software used in a variety of fields, like molecular dynamics, astronomy, physics. Passive applications are also called batch applications.

1.2 Distributed Infrastructures

Usually, scientific applications receive large input data, which they process and return the output to the user. Splitting the data among multiple processes and executing them on different processors can speed up the computation time, even if it's up to a specific limit, as given by the Amdahl's law [13]. To obtain this computing power, parallel and distributed computing systems were designed. We discuss in this section three main types of infrastructures used by the scientific community: supercomputers, clusters and grids.

1.2.1 Supercomputers

Supercomputers were designed to provide scientific applications with the needed computation power. These infrastructures are usually used to run tightly coupled applications that require hundred of thousands, or even more processors to give results within reasonable delays. Supercomputers are massively parallel processors (MPPs), composed of large numbers of processors grouped in compute nodes. Each of them has its own memory hierarchy (DRAM, or cache) and high bandwidth network cards to exchange messages with other nodes or performs I/O operations. Each node runs an optimized operating system. Recent trends are to build hybrid architectures composed of CPUs and GPUs (Graphical Processing Units). Although a GPU's frequency is lower than the one of a CPU, GPUs are designed to run a much larger number of threads and can be efficiently used to handle

a wide number of elementary matrix-vector operations in parallel. If the communication needed to move data from CPU to GPU memory back and forth is not prohibitive, the combination of CPUs and GPUs can reach a decent acceleration at a lower energy cost than CPU-only architectures.

To illustrate the size and speed of a supercomputer, let us take the fastest supercomputer in the world at the time of this writing, according to Top500 [1]. This infrastructure was designed by the Cray company and is composed of 18,688 nodes, interconnected with a proprietary Cray Gemeni torus network. Each node has one AMD Opteron 16-core CPU paired with a Nvidia Tesla K20X GPU. This supercomputer, located at DOE/SC/Oak Ridge National Laboratory achieves a performance peak over 17 PFLOPS (Floating-Point Operations per Second) on the Linpack benchmark.

Supercomputers are usually administered by resource management systems, also called *batch schedulers*. A popular solution is Slurm [193]. To run their applications, users submit requests to the system, which are then queued and run when enough resources for the request become available. We discuss further about these systems in Chapter 2.

1.2.2 Clusters

A cluster is composed of a set of individual computers built with commodity hardware, loosely connected through fast local area networks. Usually, clusters are homogeneous infrastructures, as all the machines composing them have the same configuration. The inter-node network is usually built on Gigabit Ethernet, Infiniband or Myrinet technology. Clusters represent a cheaper alternative to supercomputers and are widely used by various universities and companies. Their weak point is the network topology which usually has a lower bandwidth and a higher latency than the proprietary ones used by supercomputers. Clusters provide comparable computational power to supercomputers, occupying a large percentage in Top500 (i.e., over 80%).

Similar to supercomputers, clusters are managed by resource management systems or by cluster operating systems. Popular resource management systems are Moab [64], Torque [160] and Slurm [193]. Opposed to resource management systems, cluster operating systems, like Kerriged [120] and Mosix [20], provide users with a "single-system image", allowing them to use the cluster in the same manner as a personal computer. Users can run their applications on the cluster nodes with no information on where the application runs. An organization can have different clusters, and even different supercomputers, belonging to different departments. To provide more computing power to cycle-hungry applications, clusters are federated and managed by meta-schedulers. Meta-schedulers [171] provide an aggregated view of the federated clusters, allowing applications to be submitted to the optimal cluster for their execution.

1.2.3 Grids

Grids [70] were designed with the goal to aggregate and share resources from multiple geographical areas and belonging to different organizations in a controlled manner. Resource sharing relies on the concept of *virtual organization*, i.e., dynamic groups of individuals and institutions identified by common sharing rules. Such resources could be computational power, like in the case of computing grids, or data storage, like in the case of data grids. Basically, by connecting to the "grid" the user was supposed to have access to resources in way similar to electric grids. Some well-known examples of compute grids are the Extreme Science and Engineering Discovery Environment (XSEDE) [188] or the European Grid Infrastructure (EGI) [88]. Some grids were designed with the goal to provide users with an

experimentation platform, like PlanetLab [22] or Grid'5000 [23]. Grids were mostly used to run loosely coupled applications, composed of thousands of independent tasks. Some projects like BOINC [11] were quite successful in taking advantage of many desktop computers that were remaining idle, i.e., also called a desktop grid. Grids were also used for running workflows or tightly coupled applications, although in this case it is preferable to run the application within a site's bounds.

Grids are usually managed by grid middleware, designed to provide users with a transparent execution interface and to seamlessly run their applications on other organization resources. These systems implement their own resource discovery and selection policies. Examples of grid middleware are Globus [3], DIET [4], gLite [101] and XtreemOS [51].

To give an insight into the organization of a grid, we describe the Grid'5000 distributed platform. Grid'5000 is a distributed platform composed of 9 sites in France and one site in Luxembourg, incorporating more than 20 clusters and 7000 CPU cores. This platform can be used to run a large variety of experiments, from scientific simulations to deployment and development of middleware solutions. Grid'5000 allows users to deploy customized environments on the physical nodes, with administrative rights to install the appropriate software for their applications. Different tools to acquire resources and automate the experiments are provided to users. Grid'5000 resources are managed by the OAR resource manager [129], which allows users to make reservations of different types, including best-effort and advance reservation, from one site to multiple sites. Other tools include network management, e.g., KaVLAN [91], and efficient parallel execution of user commands, e.g., Taktuk [41].

1.3 Virtualization

The use of virtualization in distributed computing introduces multiple advantageous aspects. This section gives a brief introduction in virtualization and defines some common terms.

Definition 1.1 *Virtualization is the process of creating a resource or a whole machine, which is virtual instead of real [156]; the real resource is hidden behind a virtual interface.*

A similar concept to virtualization is *emulation*, which is the process to replicate the behavior of a resource. As opposed to virtualization, in the case of emulation, the real resource might not even exist. To better explain virtualization, let us first introduce some basic notions. Computers are composed of multiple layers of hardware and software that communicate with each other. The main hardware components are usually a central processing unit (CPU), display, keyboard, mouse, storage, memory and network card. The software layer that controls this hardware is the operating system. From the operating system's point of view the hardware components compose a *physical machine*. Because of the required level of control, the operating system needs exclusive access to the physical machine.

Definition 1.2 *A virtual machine (VM) is composed of the set of virtual resources.*

A virtual machine is defined by its resource requirements, in terms of memory and number of CPU cores, and, similar to a physical machine, it can have I/O and network devices, and a hard disk, also called a *virtual machine image*, that is backed up by a real file. This file has the layout of a real hard disk and contains all the operating system files.

Definition 1.3 *The hypervisor, or the virtual machine monitor (VMM), is the software layer that hides the physical resources from the operating system.*

Figure 1.2 illustrates better the concept of virtualization. Through virtualization, multiple operating systems are allowed to run on the same hardware, as each one of them assume that it has full control over the resources. We call these operating systems *guests* and the physical machine on which they are running a *host*. We discuss in the remaining of this section the state of the art in virtualization techniques and their capabilities.

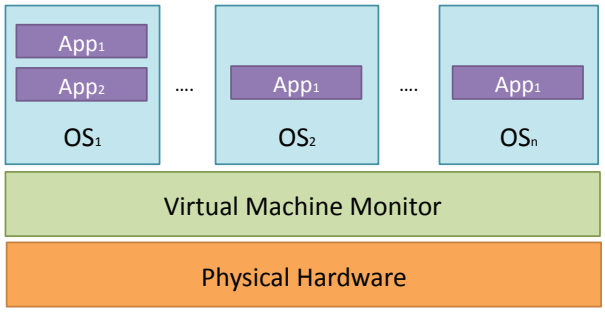


Figure 1.2: Physical machine virtualization.

1.3.1 Virtualization Techniques

Virtualizing the entire physical machine is a concept introduced in mid 60's, to support multiple operating systems on the IBM's mainframes, i.e., large multi-processor computer systems, and give concurrent access to users. At that time, however, the concept did not seem appealing enough to be adopted on a large scale, probably due to the high performance overhead and the appearance of desktop computers. Nevertheless, the concept was brought to light again in 1997 by the Disco [24] project, with the same goal: to run multiple operating systems on a multi-processor machine. This time it gained momentum as it started to be used for isolation purposes in multi-tennant datacenters, owned by an increasing number of companies. Thus, it became commercialized by vendors like VMware [184]. Currently, multiple state of the art virtualization solutions are developed.

To better understand these solutions, we first need to describe the main problem in virtualizing the physical machine. This difficulty actually comes from virtualizing a part of the CPU instruction set of the x86 architecture, composed of priviledged and sensitive instructions. This architecture is used by the majority of modern processors.

To explain what is a privileged instruction, we first need to discuss about the privilege levels that the x86 architecture provides. As shown in Figure 1.3, this architecture uses 4 privilege levels to limit what user programs can do. To prevent users to have full control over the hardware (for example a program might use the CPU forever), their applications run on the highest level while the operating system runs at the lowest level. Some instructions like I/O operations, operations on system registers or memory management operations can only be executed on the lowest privilege level, otherwise they cause a general protection exception. When the applications need to execute these instructions they give the control to the operating system. These instructions are called *privileged* and all the other instructions are called *unprivileged*.

Besides privileged instructions, the x86 instruction set also contains *sensitive*, but *unprivileged* instructions, for example instructions that modify the system registers. These

instructions do not generate an exception, but, if executed at a privilege level different than the right one, they might return wrong results.

As the hypervisor needs to have full control over the physical resources, it needs to also run at the lowest privilege level, by pushing the operating system on a higher privilege level, and handle any privileged or sensitive instructions, which might affect its state.

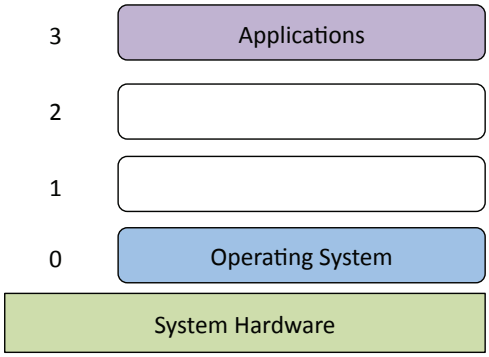


Figure 1.3: Execution levels on a x86 architecture.

Based on how state of the art virtualization solutions manage these instructions, they can be classified in : (i) full virtualization; (ii) paravirtualization; (iii) hybrid virtualization; (iv) operating system virtualization.

Full virtualization Full virtualization technologies provide complete hardware emulation such that operating systems from the virtual machines run unmodified, like on real hardware. Basically, with this technique the operating system "believes" that the hardware on which it runs is real and, thus, multiple operating systems (e.g., Linux, Windows) can reside on the same physical resources. This type of virtualization is advantageous when, for example, legacy applications requiring older operating systems might need to run on a newer operating system.

Full virtualization solutions manage privileged and sensitive x86 instructions as follows. The privileged instructions, as they do not execute at the lowest privilege level, cause an exception. The exception is caught by the hypervisor, which executes some code that emulates the proper behavior of the instruction. The sensitive instructions are replaced "on-the-fly" with instructions that can be executed in the VM through a technique called binary translation. The most popular solutions are VirtualBox [176] and VMWare Workstation [187]. Binary translation, however, brings a big performance penalty on the applications running in the VMs.

To make virtualization easier, recent processor architectures like Intel VT-x and AMD-V [174] include hardware extensions that enable processor virtualization without binary translation, also called *hardware accelerated virtualization*. Basically, these extensions introduce a new mode, called "root", which is used to run the hypervisor and is equivalent to a privilege level of -1. With this new mode, when privileged and sensitive instructions trap, they are caught by the hypervisor which executes them in the root mode. This avoids the overheads brought by emulating privileged instructions and using binary translation. Nevertheless, this transition between the two privilege levels is costly as it requires a certain non-negligible number of CPU cycles. KVM [95], recent versions of Xen (Xen HVM) [21], VMWare ESXi [33] or Microsoft Hyper-V [175] are examples of hypervisors that take advantage of hardware virtualization.

Paravirtualization Paravirtualization is a technique introduced by Xen [21]. It emerged as an alternative to full virtualization before the apparition of hardware-enabled virtualization acceleration extensions. In this case, the guest operating system is modified to replace all the instructions that are difficult to virtualize with hypercalls provided by the VMM. Paravirtualization leads to better performance than full virtualization for two reasons: (i) non-virtualizable instructions are changed to hypercalls, avoiding the overhead of using binary translation or instruction emulation; (ii) the guests are provided with a better interface to some devices, like disk or network card, allowing them to access these devices directly instead of using emulated ones. Modifying the guest operating system, however, has its own drawback, as such modifications might be difficult to apply and maintain.

Hybrid virtualization Hybrid virtualization is an optimization of the hardware accelerated full virtualization technique. It consists in providing guest operating systems with paravirtualized drivers. In this case, the hypervisor uses the hardware extensions to treat sensitive and privileged instructions while providing hypercalls for devices instead of emulating them. Hybrid virtualization was introduced in Xen HVM and in KVM through the virtio package [143].

Operating system-level virtualization This virtualization technique isolates applications in separate *containers* that share the same operating system, in a similar manner to chroot [36]. In this case, the hypervisor is the host operating system kernel. As the access to the hardware resources is done by the same host operating system, they provide better performance than previous virtualization technologies. Nevertheless, containers do not provide the same isolation guarantees as the other virtualization technologies; as the operating system kernel is shared, it is easier to compromise it and also the other containers. Well known operating system level virtualization solutions are OpenVZ [132], LXC [108], or Linux VServers [177].

1.3.2 Virtualization Capabilities

Virtualization technologies provide several attractive capabilities: (i) resource control knobs; (ii) suspend/restore mechanisms; (iii) and migration.

Resource control knobs Current hypervisors provide *fine-grained* resource control capabilities. Basically, the amount of allocated resource for each VM can be changed at user command during the VM lifetime. Let us explain how some state of the art hypervisors, Xen, KVM and VMWare ESXi server, control the allocation of different resources:

CPU allocation To control the CPU allocation, state of the art hypervisors rely on two means: proportional-share weight scheduling and CPU capping. For multi-processor systems, VMs can be allocated more than one CPU core. In this case, virtual CPU cores, also called VCPUs, can be "pinned" to the physical cores at user command. Xen provides CPU allocation and virtual CPU core pinning through its credit-based scheduler. The Xen scheduler applies proportional share to distribute the CPU time between the operating system guests and performs load balancing of guest VCPUs across all available CPUs. A guest can be assigned a weight and a cap value. The weight gives the relative CPU time share the guest receives according to the resource contention level. This share is proportional to the guest's weight and inversely proportional to the sum of all the running guest weights. The cap value sets the maximum amount of CPU time to which the guest is limited even when the host has idle cycles

left. VMWare ESXi also provides CPU time-share tuning, based on limits, proportional shares and reservations. As KVM relies exclusively on the Linux scheduler functionality, it lacks CPU time-share tuning support.

Memory allocation The memory allocation is handled through a process called "ballooning". A kernel "balloon" driver is a process that can expand or shrink its memory usage. One is started in each guest operating system. The hypervisor sends commands to this driver to expand or shrink its balloon. When a guest's memory needs to be shrunk, the hypervisor issues a command to the balloon driver to expand it's memory usage, i.e., to inflate the "balloon". When the balloon expands, the guest operating system is forced to swap pages or possibly kill other applications to free up memory. The memory pages claimed by the balloon are then handed to the hypervisor, which can allocate them to other guests. When the guest's memory needs to be expanded, the hypervisor issues a command to the balloon driver to shrink its memory usage, or deflate the "balloon". When the balloon shrinks, the guest operating system receives back the memory pages. Note that the memory cannot be expanded indefinitely. Each VM starts with a maximum amount of memory that is allocated at its boot time, amount which cannot be modified during the VM lifetime.

Other resources For more advanced functionalities, *cgroups* [115] can be used. *cgroups* is a feature provided by the Linux kernel to limit and account the resource usage of hierarchies of groups of processes. cgroups can be used to limit the resource usage of KVM guests or containers. Important resources that can be limited are: the amount of physical memory, the amount of swapped memory, the total amount of allocated memory (basically the physical and the swap), the relative CPU share and the relative disk I/O bandwidth. The term relative means that groups are assigned weights and proportional fair sharing is applied. Other resources like network bandwidth can also be controlled. For example, network bandwidth can be controlled through high-level network packet filtering rules.

Suspend/resume Hypervisors also provide the capability to suspend and resume a VM. This mechanism relies on the fact that the state of a VM, composed of its memory content and metadata concerning I/O accesses and CPU registers, can be saved on local storage. The save process starts by pausing the VM, then it saves its current memory state to the local disk, and finally stops the VM. The time required to save the VM depends on the amount of allocated memory. The restore operation loads the VM's previous memory state from the file and starts the VM from this saved point.

Migration VM migration allows transferring the memory state of a VM between two nodes. Migration is useful to achieve a diversity of scheduling goals transparently for the applications running on the physical nodes: applications can be migrated to perform maintenance on some nodes, or when the nodes are faulty, or even to shut down nodes and minimize the cluster's energy consumption.

Basically there are two types of migration: cold migration and live migration. Cold migration is done by suspending the VM, transferring its state on the destination node and restarting it. Live migration allows the application to continue running while the state is transferred to the destination. Live migration has a smaller performance degradation than cold migration as the application only perceives an "unnoticeable" down time.

Although there are some research efforts in migrating the VM image [111], current live migration techniques focus only on the memory state, and they assume that the VM image

is stored on a location accessible to both nodes involved in the process. Thus, a common setting for live migration is a cluster environment in which all the nodes share the same subnet and have access to a shared file system, like NFS [147] for example.

To better understand the impact of live migration on the application's performance, we give a rough overview of the stages involved in this process. As we do not focus on improving or changing this mechanism, we do not give a full state of the art of these techniques.

The migration process, similar to earlier techniques on process migration [118], is composed of four stages:

Preparation The migration process is started and a VM is created on the destination node. Some techniques also involve transferring the CPU state of the VM from the source node to the destination node and starting the VM.

Pre-copy The memory state of the VM is transferred to the destination node while the VM continues to run on the source node. If the application writes in memory during this time, memory pages are dirtied and need to be sent to the destination node. This transfer happens in rounds: at the beginning of each round, pages written in the previous round are sent over the network to the destination node. The transfer of dirty pages continues either for a predefined number of iterations or until their number becomes small enough to allow a total transfer of memory with a minimal down time for the application. This technique, called *pre-copy migration* was initially proposed by Xen and also applied to some extent by other hypervisors like VMWare or KVM (which does not apply any condition to stop the transfer of dirty pages and thus the migration might not converge in case of applications that dirty the memory at a fast rate).

Stop-and-copy At this point the VM is stopped and its remaining memory, together with the VM execution state are transferred to the destination. This is the period in which the application experiences a down time.

Post-copy The VM resumes on the destination node and other operations to finish the migration process might occur.

This process proves to be inefficient for applications performing memory writes, even if is efficient for applications performing many memory reads. This inefficiency comes from the fact that memory pages get dirtied fast and they need to be transferred many times over the network. To solve this issue, other live migration techniques [84] propose to skip the initial memory state transfer and the dirty page transfer from the pre-copy stage. In this case, the VM is started on the destination node and any required memory pages are transferred "on-demand" over the network from the source node; this mechanism is also known as *post-copy migration*.

1.3.3 Advantages

We identify three main advantages of use for virtualization techniques: i) customization and isolation of user environments; ii) transparent fault tolerance for applications; iii) flexible resource control.

Customization and isolation Virtualization is used in the context of HPC computing to encapsulate applications in controlled software environments, also called virtual clusters,

or virtual workspaces [93]. By encapsulating applications, many application types can be executed on the same physical infrastructure, as each application can have its own software stack, from operating system to libraries, independent of the physical platform they are running on. Also, applications can be executed on sites that do not necessary have the software configuration required by the application [60]. Because virtual environments are isolated one from another, users can be allowed to execute privileged operations without getting special privileges from system administrators and the malicious user behavior can be restrained.

Transparent fault tolerance Virtualization can be used to provide proactive and reactive fault tolerance. When failures of current nodes are imminent [61], an application running in a virtual environment can be transparently migrated to different hosts. In this case, using virtualization overcomes the need of installing operating system kernel modules or linking the application to special libraries. Moreover, techniques were proposed to suspend the virtual machines in which the application is running [60], by executing a "vm save" command on each host in synchronized fashion. For synchronization, an NTP server was used. Such techniques were attractive as they are generic and transparent for the application.

Flexible resource control There are two capabilities provided by virtualization which can be used in resource management: (i) fine-grained resource allocation; (ii) live migration. Virtualization allows to control the resource allocation of applications in a fine grain manner during the application life-time; physical nodes can be time-shared between multiple virtual machines by controlling the access to node's resources like CPU, memory and network bandwidth. The live migration capability allows applications to be seamlessly migrated between physical nodes or clusters when the resource allocation does not correspond to the host available capacity or due to administrative decisions (e.g., nodes need to be shut down for maintenance).

Using virtualization for time-sharing of cluster resources was studied by Stillwell et.al [162]. Stillwell et.al. proposed a set of scheduling heuristics to allocate fractional CPU amounts to static applications, usually submitted to batch schedulers, given their memory constraints. Thus, applications that, in the case of batch schedulers, would wait in queues as not enough resources are available on nodes to satisfy their request, could start with a fraction of their requested resources. To enforce fairness, i.e., applications could wait a long time because of their memory requirements while other applications would use too much CPU time, the heuristics use suspend/resume policies. Applications are assigned a priority dependent of their wait time, i.e., time in which they do not get to use the CPU, and applications with the lowest priority are suspended. The results, obtained through simulations, are encouraging.

Other solutions applied virtualization-based time-sharing in datacenters hosting web servers, as these applications have a dynamic resource demand. In this context, the main goal is to minimize the number of active nodes, and reduce the electricity cost. State-of-the art [80, 186] and commercial solutions [78] were proposed for VM load balancing (we do not give an exhaustive list here, as they are outside the purpose of this thesis).

Besides tuning the resource allocation in a fine-grained manner, virtualization can be used to build distributed virtual infrastructures, like clusters, spread over multiple physical clusters [60], possibly placed at different geographical locations [121]. Priority and load balancing policies can be designed for these infrastructures, by changing on-the-fly their capacity both in terms of virtual machines and resource allocated to each virtual

machine [96, 140, 145, 75]. In the grid context, such dynamic virtual infrastructures can be build for each virtual organization [121] based on the number of applications submitted by users belonging to it. With the emergence of cloud computing, private clusters can be transparently expanded by buying VMs from public clouds [110]. Moreover, the application execution on volatile resources, e.g., desktop grids, can be improved in the same manner [57].

1.4 Cloud Computing

The notion of *cloud computing* became well-known in the recent years. This notion was used to describe various service delivery and provisioning models. To clarify its meaning, we use for it the same definition as the one provided by the National Institute of Standards and Technology (NIST) [114]:

Definition 1.4 *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.*

In this manuscript, we also rely on the notion of "cloud provider" defined as follows :

Definition 1.5 *A cloud provider is the entity that offers the cloud service to interested cloud users over the network.*

To provide a cloud service, the cloud provider may manage the computing infrastructure, storage, network and cloud software components, or outsource them to other providers.

Similarly to HPC infrastructures, cloud computing provides network access to a shared resource pool. As clusters were aggregated in the context of grids, clouds can be aggregated too. For example, BonFire [86] and OpenCirrus [18] plan to provide federated cloud testbeds for users to test and develop their applications. Cloud providers can take advantage of the "cloud federation" by acquiring resources from other cloud providers to manage peaks in user requests, and better use their own resources by offering them to other cloud providers. Software stacks are designed to ensure virtual machine provisioning and application execution on multiple clouds [29, 26, 58]. A strong focus is put on providing APIs to ensure interoperability between clouds and Service Level Agreement (SLA) management, both on cloud provider and on federation level. N. Grozev et. al. [77] provide a detailed overview of these recent efforts.

Nevertheless, unlike HPC infrastructures, cloud computing promotes a "pay as-you-go on-demand" resource access model. First, the on-demand resource access allows users to quickly expand or shrink their pool of resources during their application execution. This capability opens up many paths in improving the execution of dynamic applications and allows scientific users to manage demand bursts in their own HPC centers in a fast and easy manner. Then, with the pay-as-you-go model the user is given the possibility and the incentive to make a trade-off between the speed and the cost of their computation. Finally, cloud computing commonly relies on resource virtualization: by using virtual resources, users gain control over the software environment in which their application runs in a secure and easy-to-manage fashion for the infrastructure administrator. This is advantageous both for users and infrastructure administrators as scientific applications have specific software requirements, including specific operating system versions, compilers and libraries that are

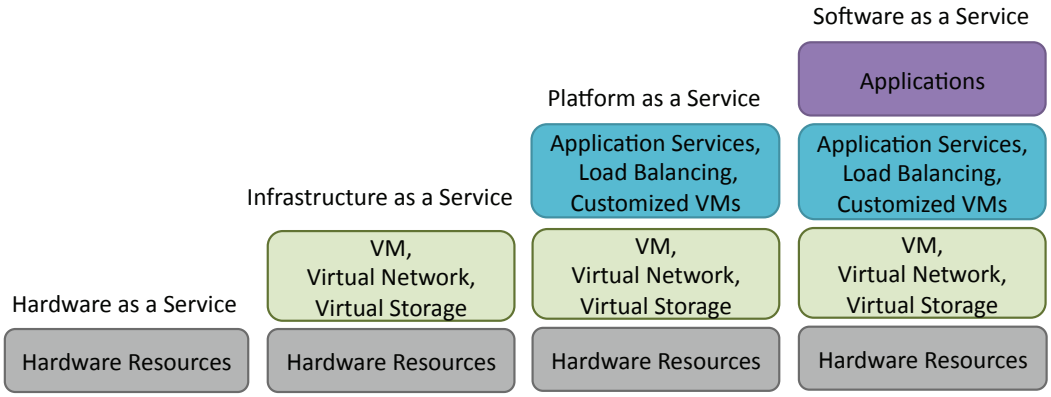


Figure 1.4: Cloud Service Models

difficult to manage on a shared physical infrastructure. All these characteristics position cloud computing as a powerful and yet simple to understand abstraction that allows clouds to fill the gap that grids didn't manage to address: by offering virtual resources to users, software heterogeneity is overcome, security can be improved and organizational boundaries can be leveled.

In the remaining of this section we give an overview of current cloud service and deployment models. Then, we describe the architecture of an IaaS cloud software stack.

1.4.1 Cloud Models

Different cloud models were developed and used in both industry and the scientific community. We describe these models in the following, by classifying them according to *what* services they provide to users and *how* they provide them.

1.4.1.1 Service models

Following the same NIST classification, the concept of cloud computing incorporates three service models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). The criteria that differentiates one model from another is the level of control that the user has over the cloud software stack. Figure 1.4 describes these models, and also another recent model, Hardware-as-a-Service (HaaS). One model builds on the abstractions provided by the previous one. To understand their characteristics, we describe each of these models and we give some illustrative examples of cloud providers.

HaaS clouds provide users with hardware "on-demand" (e.g., physical machines, storage, network). In a HaaS cloud users have full access to the physical machine and are allowed to do operating system updates and modifications. By using Definition 1.4, Grid'5000 [23] is an example of HaaS provider.

IaaS clouds provide users with virtual resources (e.g., virtual machines, virtual hardware, virtual network). Unlike in HaaS clouds, users have access only to virtual resources and have coarse-grained control over their location. For example, the user can select the cluster on which the virtual resources are placed. The user has administrative rights to her virtual machines and can decide what software is installed in them. Amazon Web Services [7] is the most commonly used commercial IaaS cloud

infrastructure. Amazon provides to its users different levels of computing power at different pricing, two storage options, key-based storage - S3 [9] - and block-based storage - EBS [6] - and advanced virtual network management capabilities [10]. Recently, a multitude of open source software stacks for IaaS clouds were developed. Some notable examples are Nimbus [126], OpenNebula [158], Eucalyptus [128], OpenStack [131] and CloudStack [47]. A different solution is Snooze [67], which proposes a hierarchical software stack, solving the scalability issues. These software stacks replicate the main services provided by Amazon, including virtual machine storage and virtual machine management. They also provide interfaces compatible with the one offered by Amazon EC2, basically by implementing a subset of the Amazon's EC2 API, thus allowing users to run the same tools and applications with minimal modifications.

PaaS clouds provide users with runtime environments to execute their applications. A PaaS cloud manages the complete life cycle of applications and hides from the users the complexity of managing virtual resources and installing and configuring the application required software. A PaaS provider can offer a diversity of services: database storage, (e.g., MySQL [59], MongoDB [35]), distributed key-value store (e.g., Redis [141]) or messaging functionality (e.g., RabbitMQ [138]), integrated development environments (IDEs), software development kits (SDKs) and deployment and management tools. An example of commercial PaaS is ElasticBeanstalk [8], which allows users to develop applications in familiar languages, like Java, Ruby or .Net and ensures the application deployment, auto-scaling and monitoring. Other examples, with a strong focus on enterprise applications, are Cloud Foundry [45], Heroku [81], OpenShift [130] and Windows Azure Cloud Services [19]. A research PaaS is ConPaaS [137], which provides services like MapReduce, Bag-of-Tasks, web and database storage.

SaaS clouds provide users with applications that run on the provider's infrastructure. The user can adjust the configuration of the application to fit her needs, but she has no control regarding the application capabilities or the underlying infrastructure. Google Apps [73] is a well-known example of SaaS.

1.4.1.2 Deployment Models

According to how the deployment is made available to cloud users, cloud deployments can be classified in: private, community, public, hybrid.

Private clouds provide services that are accessible only to users from the organization that owns the resources, as shown in Figure 1.5. The resource management is done by the organization itself. Examples can be found in a variety of companies or scientific centers which own their own clusters.

Community clouds are used by a community of users who share the same goals. As illustrated in Figure 1.6, resources are managed by one or more organizations, which might be geographically distributed. Users from these organizations have access to local cloud resources (the ones of the organization to which they belong) and to resources provided by the other organizations from the community. Scientific clouds provide an example of community clouds. Examples of scientific clouds are FutureGrid [74] and WestGrid [182].

Figure 1.5: Private cloud model

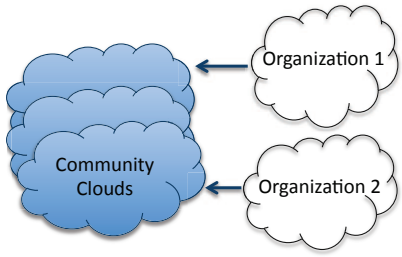


Figure 1.6: Community cloud model

Public clouds are ones in which cloud services are made available to users over a public network without restrictions, as shown in Figure 1.7. In this case, users have to pay a price for each utilization hour, usually using a credit card. Examples of public cloud providers are Amazon Web Services [7], Rackspace [139], Microsoft Azure [19] and more recently HP Cloud [42] and Google Compute [49].

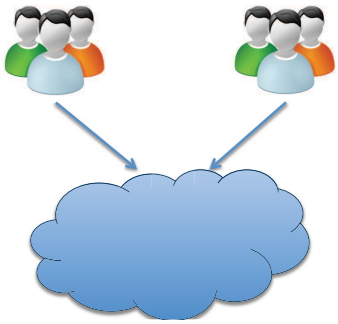


Figure 1.7: Public cloud model

Hybrid clouds are composed of two different types of clouds, as shown in Figure 1.8. A common combination is between a private and public cloud. An organization owning a private cloud might have critical periods in which the user's demand is higher than the infrastructure. In this case, to serve all user requests, the organization might use resources from public clouds (also known as cloudbursting).

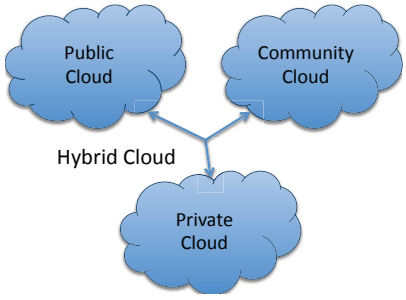


Figure 1.8: Hybrid cloud model

1.4.2 The Architecture of an IaaS cloud

In this section we focus on the architecture of an IaaS software stack. We describe this architecture as IaaS clouds are the building block for the other layers of the cloud computing stack, like PaaS or SaaS.

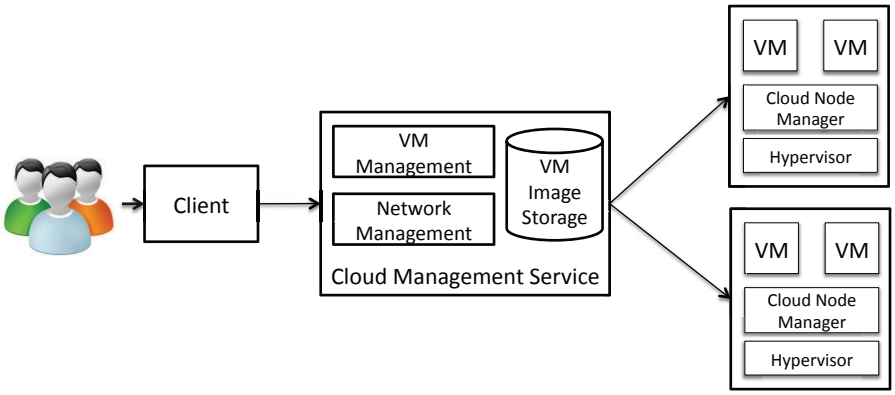


Figure 1.9: Generic IaaS cloud architecture

Figure 1.9 gives the architecture of a generic IaaS cloud manager. In general, an IaaS cloud manager provides multiple services, which are in charge of VM management, network management, VM image management and monitoring management. Users have accounts on the cloud, which are used for authentication purposes. Some cloud managers also provide accounting and quota management to restrict the user's resource usage. We describe next the main functionalities of an IaaS cloud manager: VM management, VM resource management, VM image and network management.

1.4.2.1 VM Management

The life-cycle of a VM is composed of a series of steps: VM request specification, VM scheduling, VM deployment and management during its runtime. We describe next these steps.

VM request specification Users submit requests for provisioning and managing virtual resources to the cloud manager through available APIs or by using command line

interface clients. Even if some widely recognized APIs are supported, like EC2 provided by Amazon, or OCCI [117] defined by the Open Grid Forum, cloud managers do not fully implement all the API specifications and provide their own APIs. A request contains information about the VM to be deployed, encoded in a cloud manager specific template. Basic information usually consists of the capacity of the VM (i.e., in terms of memory and CPU), network devices, disk images and hypervisor information. Usually, all user requests are authenticated, to ensure that the user has the rights to issue the command.

VM scheduling The user submitted requests are put in a queue. A scheduler deploys the VMs when enough resources are available on the host and by considering provider's objectives. For example, the OpenNebula's scheduler might schedule VMs to minimize the number of used nodes, to maximize utilization by spreading the VMs on nodes or to balance the load among nodes. Another example is the OpenStack's scheduler, which applies filters on the hosts on which the VM is deployed. Filters might include "zone" information, to ensure that VMs are deployed in clusters from specific geographical areas, capacity requirements, co-location and anti-co-location requirements. The scheduler allocates resources to the provisioned VMs based on the user specification. As mentioned in Section 1.3.2, the VM receives a share of CPU time proportional to the weight specified in its template.

VM deployment To deploy a VM, several steps are followed. First, the virtual machine image is transferred to the host. Then, the template of the VM is translated to a hypervisor specific description. Software libraries, like libvirt [104], provide unified a hypervisor interface and VM description template, hiding the hypervisor heterogeneity. Finally, a cloud manager component residing on the host instructs the hypervisor to start the VM. When the VM is booted, a *contextualization* process can be applied [94]. Basically, this process runs some user-defined scripts and configures specific parameters from the VM's operating system configuration to the values given by the user. Such parameters might include networking information (e.g. the VM's host name, IP, network gateway, DNS server) or user information (e.g. name or SSH key). For example, in OpenNebula, the contextualization process consists in encapsulating all user specified scripts in an ISO image. A script installed to run at boot time in the VM mounts the ISO image and executes the user defined scrips.

VM operations When the VM runs, the cloud manager periodically monitors it, retrieving information regarding its resource utilization. During the VM life-time, users have the possibility to save the VM state, suspend the VM, resume it from the previously saved state, and stop, in a forcibly manner or not, the VM. VM migration is usually not provided by the cloud's manager APIs as it can interfere with the scheduler behavior. Nevertheless, some cloud managers, like OpenNebula, provide such low level functionalities too, letting users to decide on which nodes their VMs are deployed and migrated.

1.4.2.2 Virtual Network Management

A cloud manager also provides means to manage the network connectivity of the VMs. The simplest way to manage the network connection is to assign to each VM an IP from a private range. This can be used either through a DHCP server or by configuring the network interface of the booted VM with an IP from a range that is currently available. The latest method is usually a part of the VM contextualization process. When a new

VM is booted, the cloud manager selects an IP from the specified range that is currently available.

1.4.2.3 VM Image Storage and Management

Users are allowed to customize their VM images and store them in a repository. These images can be made public so that all users can use them, or be accessible only to the user that created them. The repository can reside on a shared file system or be stored on the local disk of the cloud frontend. The first option is prefered when live migration needs to be used, as live migration commonly includes only memory state migration. This also has the advantage of avoiding the copy of the VM image to nodes, which is a costly operation, slowing down the VM boot process. [1] To optimize the VM boot process, "copy-on-write" images can be used. The copy-on-write process avoids creating a full VM image for each new VM, by creating a VM image image based on the original backing image. When the processes running in the VMs need to read data from the disk, they access the original image; when they need to write data to the disk, they access the new disk image.

1.5 Conclusion

In this chapter we presented the foundation on which this dissertation is based.

To understand the issues we address in our work, we analyzed the properties of scientific applications that usually run on HPC infrastructures. We investigated the different requirements in terms of communication, resource allocation and interaction with users. Scientific applications can be loosely coupled, i.e., composed of tasks that do not communicate with each other, or tightly coupled, i.e., composed of tasks that synchronize through frequent messages. Besides static applications, composed of a fixed number of processes, the HPC landscape also contains a variety of dynamic applications, for which resource requirements can evolve in time, sometimes in an unpredictable way. These applications can be either interactive, or passive.

Then, we introduced the existing HPC infrastructures: supercomputers, clusters and grids. We described the main technology used in cloud, virtualization, and investigated its capabilities. Providing users with virtual resources instead of physical ones became attractive from several reasons. First, users have access to customizable environments while the infrastructure administrators are relieved from the burden of installing a diversity of software packages and solving possible configuration conflicts. Then, virtualization provides fine-grained resource control and live migration, making the resource management process more efficient and flexible. The cloud computing paradigm is similar to the resource management mechanisms used on HPC infrastructures, with the exception of an important detail: fast on-demand self-service provisioning of resources. Users can provision virtual resources for their applications any time during their execution and improve their performance. This new concept is useful for the soon-to-become complex and dynamic scientific applications. Finally, as the IaaS cloud represents the building block for this dissertation, we gave a brief overview of how an IaaS cloud infrastructure is managed.

The next chapter describes the environment we considered in our work and discusses the state of the art in dynamic resource management.

[1] Techniques inspired by the Bittorent protocol [169] are used to speed-up the VM image transfer but in this case live migration would not be supported.

Chapter 2

Resource Management Solutions

This chapter analyzes the resource management solutions related to our work. We study solutions that have the potential to meet our previously stated requirements: SLO support flexibility, maximum and fair resource utilization. Before introducing these solutions, however, let us discuss the environment which we target.

This dissertation focuses on managing resources of clusters with homogeneous resources. From the cloud computing perspective, these clusters represent a private IaaS infrastructure. We focus on virtualized infrastructures, as they provide users with the flexibility to configure the right software environment for their application. On this infrastructure we aim to improve the execution of both static and dynamic applications. We assume an environment in which users might not be willing to cooperate, i.e., to relinquish their resources for "free" when another user has an urgent need. We also assume that users might have different performance objectives for these applications, e.g., minimum or maximum expected execution time, and they might value differently their execution.

This chapter is organized as follows. Section 2.1 gives an overview of resource management models and discusses user expectations regarding infrastructure resource management. Section 2.2 details the currently used and proposed resource management systems, which share a part of this dissertation's goals, but they lack fair resource regulation among users. It discusses batch schedulers that run applications on the "bare metal", systems designed for dynamic applications and virtual cluster management systems, which multiplex the physical resources through the use of virtualization. Then it gives a summary of solutions used in datacenters to manage web applications. Section 2.3 discusses efforts to provide support for automatic application adaptation in the cloud computing landscape. Section 2.4 discusses related solutions, which address the fair resource utilization aspect while still providing SLO support flexibility. Finally, Section 2.5 concludes the chapter.

2.1 General Concepts

In this section we introduce some useful concepts for understanding the state of the art in resource management. We classify the resource management models applied in distributed systems and we discuss some terms used to define user requirements.

2.1.1 Terminology

Before describing the state of the art in resource management, let us first define the terms that we use in the rest of this section:

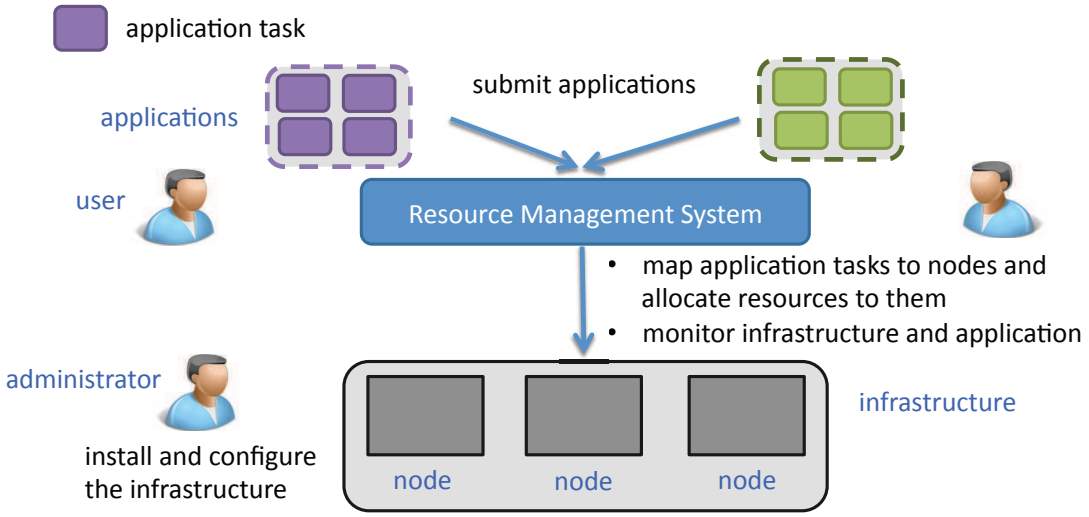


Figure 2.1: Overview the resource management process.

resource By resource we understand a consumable abstraction of a physical device, e.g., CPU, memory;

resource management system We define a resource management system as the entity that manages all the resources provided by the infrastructure.

2.1.2 The Resource Management Process

An overview of the resource management process is given in Figure 2.2. The infrastructure can be used by multiple users that want to concurently run applications. An infrastructure administrator manages all the software configuration required by the resource management system and the user applications.

The resource management process involves several steps:

application submission To run their applications on the infrastructure, users submit a request to a resource management system. Usually, a request contains the characteristics of the application:

- the executable and the data needed to start the application;
- the required resources using the granularity specified by the resource management system (i.e., number of nodes, number of cores, amount of memory); some resource management systems request users to specify the duration over which resources are allocated to the application tasks at the time at which they have to be allocated;
- constraints regarding the application task execution, e.g., some tasks might require to start after other tasks finish their execution.

resource allocation and application execution The resource management system allocates the resources provided by physical nodes to the application tasks and starts the application execution. To start the application execution, the resource management system might need to copy the application data on the nodes. This step is avoided if a shared file system is mounted on all the nodes. When the application finished its execution, the user has access to the produced data.

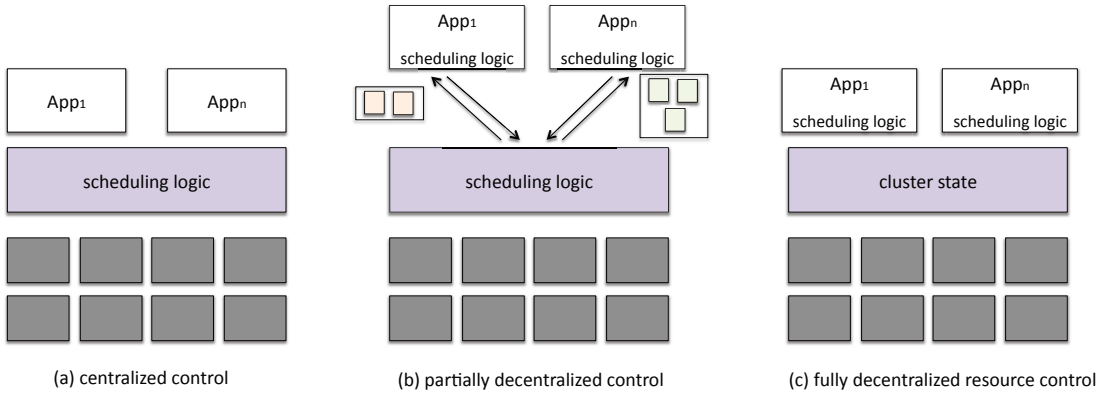


Figure 2.2: Overview of resource management models.

monitoring To report any failures and account users for used resources, the resource management system monitors both the infrastructure and the applications executing on it.

2.1.3 Resource Management Models

In this section we discuss three different characteristics, which are important in the design of a resource management system: (i) the resource control distribution; (ii) the granularity of resource allocations; (iii) the dynamicity of resource allocations.

Resource control distribution

This characteristic expresses the level of control applications have over their resource allocation. Allowing or constraining the application resource control is important as it can improve or limit the efficiency of the resource management system. [1] Based on the level of resource control provided to applications, we distinguish three resource management models: (i) centralized; (ii) partially decentralized; (iii) and fully decentralized. Figure 2.2 illustrates these types:

centralized: In a centralized model the resource management system decides how many resources to allocate to each application based on internal scheduling algorithms. The mapping of application tasks to resources is done serially. Usually, application requests are put in a queue. Then, the resource management system takes each request and checks if resources can be allocated to it, based on requirements in terms of priority, type of resources, size of resources. To support multiple application types, this entity either employs different scheduling algorithms for each application type, or provides a simple rigid interface that matches all requirements. This resource model is useful for an infrastructure designed for a few application types, which do not require a complex scheduling logic. Nevertheless, to support different application scheduling algorithms, a centralized resource management system might require a complicated design.

partially decentralized: A partially decentralized model splits the resource scheduling on two levels. A set of agents may take specialized scheduling decisions for their

[1] We use the term efficiency here to describe how good is the system in optimizing the application performance and resource utilization.

applications, i.e., they may decide how many resources the application needs or where they should be placed. These decisions are communicated to a global entity that considers them in computing the global resource allocation. When there are different application types executing on the infrastructure, a partially decentralized resource model is better than a centralized one. Each application can apply independently its own resource selection algorithms.

fully decentralized: A fully decentralized model relies on a set of agents that decide how many resources their applications need. There is no entity to regulate the agent's access to resources and the global resource allocation is the result of the decision process taken by each agent. A fully decentralized model is attractive as it gives full resource control to application agents. A drawback, or rather a difficulty, of this model comes from the need of coordinating agents to reach an efficient allocation.

Resource allocation granularity

This characteristic expresses the partitioning level of resources. The partitioning level is important if applications have diverse resource demands, as it can lead to poor infrastructure utilization. Based on the granularity of resource allocations, we identify two types of resource models: (i) coarse-grained; (ii) and fine-grained.

coarse-grained: a coarse-grained resource model allocates fixed units of resources to applications. These units may be physical nodes, or slots corresponding to fixed partitions from the physical node, e.g., 2 CPU cores and 1 GB RAM. This model is advantageous because it is simple to use. Nevertheless, it leads to poor resource utilization: applications might consume less than their allocated slots, leaving resources idle.

fine-grained: a fine-grained resource model allocates arbitrary resource amounts to applications, e.g., an application can request 1 CPU core while another application can request 10% CPU time on one core. A fine-grained resource model is advantageous to use, as it increases the resource utilization. However, to avoid resource fragmentation, isolation and migration mechanisms are needed.

Resource allocation dynamicity

This characteristic expresses the capability of the resource management system to change the application's resource allocation during its execution. Based on the dynamicity of resource allocations, we discuss two types of resource models:

static: In this model the amount of resources allocated to an application is fixed at the start of its execution. After it started executing, the application cannot change this amount. This model was useful in distributing resources to parallel applications running in supercomputing centers. Nevertheless, as currently these applications have more dynamic resource demands, static partitioning becomes an inefficient model.

dynamic: In this model the amount of resources allocated to an application can be changed during the application runtime. This model improves the infrastructure utilization and application performance.

2.1.4 SLO Support

Resource management systems are designed to meet two objectives: to provide maximum satisfaction to users while addressing infrastructure administrator needs, e.g., maximized resource utilization, minimum energy cost, etc. Nevertheless, users requirements can be different in terms of expected application performance: some users might want to finish their application before a deadline, others might want to execute it as fast as possible. In this document we use the term Service Level Objective (SLO) to define the user requirements. Service Level Objectives (SLOs) were used to represent constraints on a service provider and customer behavior [183]. In our context, the provided service is the application execution, the provider is the system managing the infrastructure and the customer is the user. For the users, SLOs describe the performance goals they set for their applications. We mention some examples of SLOs for interactive and batch applications. For example, for a batch application, SLOs may consist of receiving the result before a deadline, or obtaining an execution time as close as possible to the ideal. For an interactive application, SLOs may consist of limits on the total throughput in terms of number of requests served per second.

The capacity of the resource management system to provide SLO support is given by the mechanisms used to ensure that the user-specified SLO is not broken.

2.2 Resource Management Systems for Clusters

In this section we discuss solutions that multiplex cluster resources between multiple applications. As these solutions do not consider the value users might have for their applications, they are "unfair". We study at what extent such solutions are flexible by studying the characteristics of the used resource management model (resource control distribution, resource allocation dynamicity and resource allocation granularity). and what support they provide for different application performance guarantees. Then we analyze the granularity of the resource allocation and the fairness they provide in allocating resources to users.

2.2.1 Batch Schedulers

Batch schedulers are systems used to manage compute resources of supercomputers and clusters. Popular open source solutions are SLURM [193], Torque [160] with its scheduler Maui [112] and OAR [129]. Commercial solutions are, for example, Moab [119] and Tivoli Workload Scheduler [152].

Batch schedulers provide users a "job" interface to submit and run their applications on the infrastructure: users make a request for a number of resources and specify what application to run on them. The submitted requests are placed in the scheduler's queue and applications are executed on the nodes when enough resources become available.

Batch schedulers are centralized resource management systems, which employ a variety of scheduling algorithms derived from First-Come-First-Served (e.g., EASY [159] or Conservative Backfilling [159]). Their provided interface is generic enough to be used by a variety of applications.

Although in the past batch schedulers supported only static applications, recent efforts are made to improve the execution of dynamic applications. Such applications are difficult to run on infrastructures managed by batch schedulers. Let us take as an example the execution of a bag of tasks application: the tasks of this application are either submitted as individual jobs to the batch scheduler's queue, leading to a large management overhead,

or are managed through workarounds like pilot jobs [30] [2]. Currently, batch schedulers like SLURM, Torque and Moab, provide support for dynamic applications by shrinking and expanding the resources allocated during their execution. SLURM provides partial support as the change is initiated by the user and, to expand its resource allocation, the user needs to submit a new "job" and to merge the acquired resources with the original "job". In Torque and Moab the application provides a script called afterwards by the batch scheduler during the application execution.

SLO support was provided to users through advance reservations and deadline-driven application execution. To obtain an advance reservation, the user specifies the number of nodes, the expected execution time and the start date. The batch scheduler then "books" the resources for the specified time interval. Nevertheless, this technique leads to poor resource utilization, as jobs cannot be scheduled efficiently, leading to periods when resources are idle while jobs wait in the scheduler's queue. Moab supports deadline-driven execution: the user can specify the absolute deadline, the expected execution time and the number of nodes. In this case, the batch scheduler reserves the resources in a way that optimizes the resource utilization while ensuring that all deadlines can be met.

The resource model employed by batch schedulers is coarse-grained: each application has exclusive access to the physical nodes on which it is running. Although it provides maximum application performance, due to lack of interference from other applications, as some applications do not use all the node's resources, it can lead to poor resource utilization. Some efforts were made to use time-sharing, as a way to improve the resource utilization: gang-scheduling (i.e., scheduling all processes of a job simultaneously) is a well-known method to share the same nodes between multiple applications [65]. Nevertheless, gang scheduling is costly in terms of performance and thus is rarely used in practice. Recently, however, batch schedulers also include virtualization in the resource management process, e.g., Moab provides a cloud suite.

Resource regulation is done through priority queues and fair-sharing policies. Such mechanisms do not provide the right incentives for users to use resources.

2.2.2 Dynamic Resource Management Systems

Several resource management systems were developed to support the execution of dynamic applications on shared infrastructures [165, 97, 98, 122, 83, 153].

ReSHAPE ReSHAPE is a centralized resource management framework for MPI applications [165]. ReSHAPE relies on a scheduler to decide how resources are distributed to applications, while each application interacts with it through a specialized API. Supported applications are homogeneous and iterative: they are composed of a fixed number of iterations with each iteration requiring a similar computation amount. Applications use "resize points" in their code to contact the scheduler and indicate their performance. Then, the scheduler decides which application to expand or shrink, based on the application's performance benefit. Although the solution might improve the resource utilization and application response times, it is also designed for one application type and requires to re-engineer the applications.

CooRM CooRM [97, 98] is a partially decentralized resource management framework that supports different application types. Supported applications can be rigid, moldable,

[2] When the job is submitted a specific agent is started on the nodes. Afterwards, this agent communicates with an external scheduler to fetch application's tasks and execute them.

malleable or evolving. Eeach application is associated with a launcher that receives from a central scheduler "views" of the system, i.e., a view represents the availability of resources over time. The views offered to applications are: (i) non-preemptive, used by static applications and by dynamic applications to ensure to themselves a minimum resource amount; (ii) and preemptive, only used by dynamic applications to expand/shrink their allocation. Resources belonging to the preemptive view are allocated using a fair-share policy, and can be preempted any time during the application execution. The launcher uses these views and the application's resource selection algorithms to decide to start the application. If the application is dynamic, the launcher updates the application's allocation during the application runtime. Resources are allocated in a space-shared fashion , i.e., one node per application process.

YARN Yarn [122] is a partiallly decentralized system. In Yarn each application has its own application manager that requests resources from a global scheduler. The global scheduler allocates resources to them by considering specific application constraints. Currently, it supports only memory allocation.

Mesos Mesos [83] is a framework that allocates resources among applications based on the concept of "resource offer" [3]. A central scheduler presents applications with resource offers containing available resources on physical machines. The scheduler decides how many resources to offer while applications can run their own selection algorithms to decide what resources to accept (e.g., an application can choose to run its tasks on nodes that satisfy a data locality constraint). Resources are partitioned in a fine-grained manner through the use of virtualization. Mesos partially decentralizes the resource control, as it allows each application to select the resources on which it runs. The central scheduler, however, decides in the end what resources to give to each application.

Omega Omega [153], the "Google's next-generation cluster management system", presents applications with a "replicated" shared view of the cluster, called a "cell state". Each application keeps its own cell state, which is synchronized periodically to reflect allocation changes. Using this cell state, applications can select what resources from the cluster to use, according to locality or resiliency constraints, even by preempting other application tasks. The cell state is kept in a persistent data store and resource selection is done through transactions. If there are concurrent transactions, some of them might fail, depending on the validation policies implemented for the cell state, i.e., a framework with a higher priority has precedence in its allocations. In this case, the framework's scheduler needs to update its cell state and implement retry policies. As there is no central scheduler to decide the mapping of application tasks to nodes, Omega is fully decentralized.

2.2.3 Dynamic Virtual Clusters

Due to the features that it provides, virtualization became of interest to use in distributed computing [69, 85, 172] and was ultimately adopted by cloud computing as its base building block. Virtualization provides users with abstractions like *virtual environments* or *virtual clusters* that represent sets of one or more configured and interconnected VMs. Using these

[3] Actually Mesos allocates resources among frameworks. Users might not be aware of Mesos as they submit their applications directly to the framework's scheduler. Nevertheless, from the point of view of the resource management system, a framework can be seen as an application. Thus, for clarity, we use the term application.

abstractions in the management of physical infrastructures gives more flexibility to users in controlling the environment in which their applications run. Given the recent efforts to improve current virtualization mechanisms and better support HPC workloads, i.e., by virtualizing specialized network devices like Infiniband [72], using virtualization in HPC centers becomes feasible. In this section we summarize the resource management solutions that used virtual clusters.

Haizea Haizea [157] is a centralized lease-based resource management system for VMs. A lease is defined as a set of VMs made available to a user based on a contract that specifies the provisioning terms, like VM resource requirements and the lease duration. Haizea supports different types of leases, from best-effort to advance reservation, and handles the overheads related to VM management. By scheduling the VM deployment before the start time of the lease, the system ensures that the time required for VM management is not charged from the lease period and, thus, it does not delay the execution of jobs for which resources were reserved in advance. This model uses the scheduling algorithms implemented by batch schedulers.

Maestro-VC Maestro-VC [96] is a partially decentralized solution that creates and manages dynamic virtual clusters. This solution relies on a global scheduler that decides the resource allocation for each virtual cluster and local schedulers, running in the virtual clusters, which negotiate their virtual cluster allocation with the global scheduler. Thus, the local scheduler can take resizing decisions, allowing the application to run with less resources, or it can checkpoint the application, thus avoiding loosing the already performed computation. If resources need to be revoked from a virtual cluster, the global scheduler notifies the corresponding local scheduler. Unfortunately, this solution does not include policies to decide the resource amount each virtual cluster receives.

Violins Violins [146] are autonomic virtual clusters that are resized and live migrated between nodes, and possibly different physical domains, to meet a certain application resource utilization level. Each application from the system runs in its own VIOLIN. S tarting from the assumption that the application resource demand can be dynamic inside the VM, a VIOLIN can be resized in a fine-grained manner. The resource allocation decisions are taken by a global *adaptation manager*, which retrieves resource utilization from physical nodes and VMs, and decides the resource allocation and placement of each VIOLIN to satisfy the application resource demand. This solution is actually a centralized system, as the main allocation decisions are taken by a central entity, based only on VM utilization metrics. Moreover, they do not consider per-application performance objectives.

ORCA ORCA [76] is a middleware solution for federated domains that relies on three entities: consumers, providers and brokers. As illustrated in Figure 2.3, these entities interact with each other to schedule and share resources. Consumers, which are actually application agents, request and acquire tickets from brokers, entities that aggregate and manage resources from one or multiple providers. In order to actually receive the resources, the consumer needs to present the ticket to the provider, which in turn leases the resources to it. Basically, tickets represent the ownership of the acquired resources, and specify, among others, the identity of the provider to which those resources belong. The lease, instead, represents the actual resource allocation.

This resource model is attractive for federated environments as it allows providers to keep a level of control over their resources. The way leasing is implemented in ORCA allows

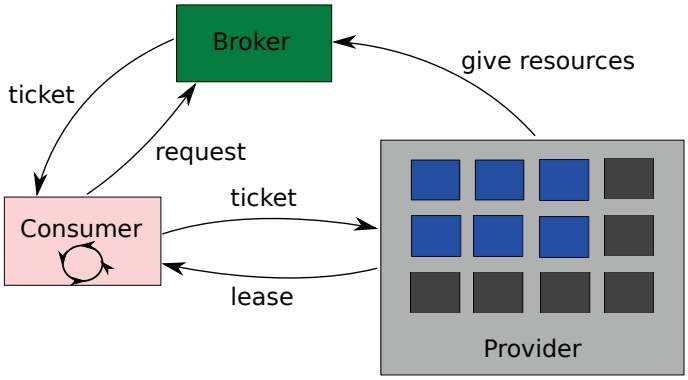


Figure 2.3: Overview of resource management in ORCA.

a flexible resource model usable by dynamic applications: leases can be renegotiated during their runtime; their duration or the number of allocated units can be changed. Several application agents were designed; these agents change dynamically their resource demand according to application needs. For example, virtual clusters belonging to different virtual organizations could be scaled automatically, with resources from different sites [140].

Nevertheless, this solution boils down to a centralized resource management system. Leases are scheduled by a omnipotent entity at each provider site and the lease abstraction by itself is not enough to ensure an efficient resource management in contention periods. To solve this last issue an attempt to implement a virtual economy was presented in Cereus [90], which uses the notion of re-chargable currency to control the user behavior. Nonetheless, further references on how such policies would be used in a real setting or how application agents would react in an economic setting are missing.

COD COD [32] introduces the idea of isolating different application types in virtual clusters, and controlling the resource allocation of each cluster. In this case, virtual clusters are composed of hardware resources, instead of VMs. Each virtual cluster is managed by a *virtual cluster manager*, which monitors the state of the service or application running in the virtual cluster. A central manager receives all resize requests from the virtual cluster managers and, based on the virtual cluster priority and its minimum guaranteed resource share, decides the resource allocations. The solution is validated by sharing dynamically a resource pool between three SGE [62]-managed virtual clusters. COD is a partially decentralized solution.

Viocluster Viocluster [145] is a solution to exchange resources between different sites based on the workload they have to process. As the sites might have different software configurations for their physical machines, the exchanged resources are VMs. Viocluster provides a decentralized resource management solution: each domain tries to adapt its resource demand to the needs of the applications running in it by negotiating resources with other domains through the use of a broker, which can be seen as a cooperative entity. The negotiation is done through contracts that specify policies to borrow and lend resources. The authors validate this solution by simulating two PBS [167]-managed clusters that exchange resources to satisfy their peak demands. Unfortunately, the solution is not further refined and it lacks priority differentiation mechanisms.

Self-tuning virtual machines "Self-tunning virtual machines" [134] is a project developed to run deadline-driven applications in controlled virtual environments. Although this solution does not target virtual cluster management, it is close to the goals of our work as it runs scientific applications in VMs in an autonomous decentralized manner. Applications run in single VMs on a node. Each virtual environment has allocated an amount of resource, which allows it to meet the application's deadline while "best-effort" applications use the remaining resources. The resource amount is determined by profiling the application. Using this profile, a feedback controller tunes the CPU allocation of the VM such that the application running inside makes enough progress in its computation to meet its deadline. In this way, more applications can run on the infrastructure. A global admission control is used to avoid overloading the system. The system is fully decentralized: applications are altruistic and adapt with no knowledge one from another.

2.2.4 SLO-driven Dynamic Resource Management in Datacenters

In this section we analyze solutions designed for datacenters, i.e., a cluster built from commodity hardware, usually used to host enterprise applications, and web servers. Although these solutions target web servers, they share a part of this dissertation goals, regarding maximum resource utilization and flexibility by supporting dynamic applications and SLO requirements. These solutions rely on the use of performance models to obtain estimations and predict the application resource allocation. In this case, resources are allocated to VMs in a fine-grained fashion as applications like web servers can have varying CPU and memory utilization. Optimal resource utilization is obtained by providing each application with a sufficient number of resources to minimize the risk of violating its SLO while reducing the number of used nodes. VM migrations are employed to move the VMs either reactively [186], or proactively [154].

The work of Xu In [189] the authors proposed a partially decentralized architecture to share the datacenter among multiple web servers. Each application runs in a VM managed by a local controller. The local controller uses fuzzy-logic to learn the relationship between resource usage and application workload and to predict future resource demands. This relationship is stored as a set of IF-THEN rules in a database and updated whenever new usage patterns appear. Based on this set of rules, the controller is able to output the resource demand according to incoming workload. These demands are then sent to a global datacenter controler that allocates resources to optimize the infrastructure's profit.

1000 islands Zhu et.al. [198] presented 1000 islands, an autonomic resource management framework. This framework relies on a hierarchical set of controllers which predict the application performance, scale the VM resource allocation, and migrate VMs between nodes. The controllers can adapt at three different time scales. The smallest scale is used to adjust the resource allocation per VM. Then, when the current node on which the VM was hosted can not provide the required capacity, VMs is migrated to more suitable nodes. Finally, the last time scale, and also the largest, is used to migrate VMs between groups of nodes, i.e., clusters.

AppRaise AppRaise [181] is a hierarchical resource management framework. Each application has its own controller which manages multiple VMs, depending of the number of application components. The controller estimates the resource utilization targets for its VMs using control theory algorithms and communicates them directly to controllers

managing node resources, i.e., node controller. The information about node controllers is retrieved from a system database. Each node controller applies arbitration policies in resource contention periods. Load balancing policies are missing, although the authors say they can be easily integrated.

iSSe iSSe [79] is a project which scales the application resource allocation in two ways: (i) first it scales up and down the resource demand of each VM; (ii) and if the first scaling is not sufficient, it scales the number of VMs. Here the idea is to perform fine-grained resource allocation by re-distributing resources among the VMs belonging to the same application before taking resources from other applications. Like previous works, iSSe supports only web applications. Furthermore, the solution is incomplete as it misses load balancing policies between nodes.

The work of Maurer In [113] the authors propose an architecture based on "escalation levels", which divide the actions involved in the resource management process. At the lowest escalation level, the resource configuration of the VM is changed. If this action is not possible, application components are migrated to other available VMs. Then, new VMs are deployed, new nodes are turned on, if nodes are available they are turned on or VMs from public clouds are rent. Although the idea is interesting, in the end, the authors address only the first escalation level by proposing rules to scale the resource allocation of the VM based on utilization thresholds. Policies to mediate resource conflicts between multiple applications are missing (e.g., before renting VMs from public clouds, the allocation of other existing VMs could be shrunk).

Friendly virtual machines "Friendly virtual machines" [197] is a fully decentralized system. Multiple "altruistic" applications, running each in its own VM, share the resources from the same node by adapting their resource demand based on a "feedback signal" received from the system. As a feedback signal, the authors use the virtual clock time (i.e., "the time interval between two clock cycles") of the VM: if a resource like CPU or memory is contended the virtual clock time increases. When the resource contention is high the application decreases its resource demand, i.e., if the application is multi-threaded it can reduce the number of threads, and when the resources are underutilized the application increases its resource demand. The resource adaptation strategy of each application relies on additive-increase/multiplicative-decrease policy: the application increases incrementally its resource demand and decreases it exponentially.

2.2.5 Summary

Table 2.1 summarizes the solutions to manage cluster resources. We outline several characteristics that are important in meeting our previously defined requirements: SLO support flexibility, resource allocation efficiency and resource utilization fairness. As these solutions, with the exception of [157], provide support, in the worse case in a limited form, for dynamic resource allocation, we did not include this requirement. We will not further discuss it. Let us summarize now how these requirements are met by the previously introduced solutions:

SLO support flexiblity The SLO support flexibility provided by a resource management system is given by four characteristics: (i) resource control distribution model; (ii) resource allocation dynamicity; (iv) the applied isolation unit, i.e., the isolation level the users see for their applications.

	SLO support flexibility		Maximum utilization	Fairness
Solution	Resource control distribution	Unit of isolation	Resource allocation granularity	Resource regulation mechanism
Batch schedulers	centralized	cluster	coarse-grained	priority classes quotas
RESHAPE [165]	centralized	cluster	coarse-grained	none
CooRM [97, 98]	partially decentralized	cluster	coarse-grained	none
Yarn [122]	partially decentralized	application	fine-grained	none
Mesos [83]	partially decentralized	framework	fine-grained	weights
Omega [153]	fully decentralized	framework	fine-grained	none
Haizea [157]	centralized	application	coarse-grained	lease pricing
Maestro-VC [96]	partially decentralized	framework	coarse-grained	none
Violins [146]	centralized	application	fine-grained	none
ORCA [76]	centralized	application	fine-grained	virtual economy
COD [32]	partially decentralized	framework	coarse-grained	priorities
Viocluster [145]	fully decentralized	framework	coarse-grained	none
Xu.et.al [189]	partially decentralized	application	fine-grained	none
1000 islands [198]	none	application	fine-grained	none
AppRaise [181]	partially decentralized	application	fine-grained	priorities
iSSe [79]	none	application	fine-grained	none
Maurer et.al. [113]	centralized	application	fine-grained	none
Friendly virtual machines [197]	fully decentralized	application	fine-grained	none
Self-tuning virtual machines [134]	fully decentralized	application	fine-grained	none

Table 2.1: Summary of virtual cluster solutions

Although several solutions employ a centralized model, a majority of solutions rely on a partially decentralized one: an application "agent" changes the resource allocation of its application, by negotiating with a central entity that holds the control over the infrastructure's resources. This aspect leads to supporting different user performance objectives and resource demand adaptation policies.

Another aspect in which we are interested in, is the isolation unit these solutions provide to users. We focus on the isolation *users see* when submitting their applications on the infrastructure: if they can run their applications in a controlled customizable environment, or they share the environment configuration with other users. From this perspective, only Haizea, Orca and Violins can isolate applications from each other, while the other solutions focus on isolating frameworks, used by multiple users. In this last case, users are obliged to submit applications to predefined environments, set up for a specific application type. This limits the flexibility of supporting different applications on the infrastructure.

Maximum resource utilization The maximum resource utilization requirement is given by the resource allocation granularity. Most of the solutions used in HPC clusters employ a coarse-grained resource model. This model provides better performance for scientific applications, as there is no co-location intereference, but may lead to poor resource utilization. Other solutions, used in datacenters, or in HPC clusters, like Violins, or ORCA, are closer to our goals as they propose a fine-grained resource model, meeting our efficiency requirement.

Resource utilization fairness The resource utilization fairness requirement is given by the mechanisms used to limit the user resource demand. The solutions mentioned in Table 2.1 either do not provide and resource regulation mechanism, or they mostly rely on the use of priorities, also called weights, and quotas to allocate resources among users. These mechanisms are not fair as they do not give any incentives to users to put truthful value on the resources. Users might always want the highest priority for their application and to execute it as fast as possible, even if other users might urgently need resources. In ORCA, the authors have tried to integrate a virtual economy, but details regarding its real-world validation are missing. Finally, in the "Friendly virtual machines" project fairness is given by the application altruism.

2.3 Cloud Auto-scaling Systems

As clouds bring the "elasticity" feature, useful for dynamic applications, it is important to see what support current private and public cloud providers offer to users for implementing elasticity policies and automatically scale their applications.

2.3.1 Commercial Clouds

Amazon EC2 provides an auto-scaling service, as part of their IaaS offering, accessible through a web service API or command line interface [5]. This service relies on the concept of "auto scaling group". An auto scaling group is defined by a launch configuration, a set of user configured parameters and a set of policies. The launch configuration specifies the virtual image used to create the VMs and the VM hardware configuration. The auto scaling groups are homogeneous: all the VMs provisioned for the group have the same launch configuration. The user can specify for these groups a minimum and a maximum VM size and deployment preferences, e.g., availability zones. The policies used to scale the

groups rely on "metric alarms": the user chooses metrics from the ones made available by a monitoring service, the value to be observed, e.g., average, the observation period and the threshold that triggers the alarm. When the alarm is triggered an action is taken.

Microsoft Azure provides an auto-scaling feature, called "Auto-scaling Application Block", as a part of the Enterprise Library Integration Pack for Windows Azure [19]. This feature is similar to the functionality provided by the Amazon EC2's service but it misses availability zone support. Nevertheless, besides scaling the number of VMs it allows updating the VM instance type too. This service relies on two types of rules: constraint and reactive rules. Constraint rules are used to specify a number of running VMs at each hourly interval; this number is independent of the application state. Reactive rules cope with changes in application state based on user defined metrics. Nevertheless, they seem focused more on web applications.

enStratus [63] provides a cloud platform with auto-scaling functionality for RackSpace [139], similar to Amazon EC2's auto-scaling service. StarCluster [161] is a solution for configuring and managing virtual clusters for scientific applications on top of Amazon EC2. StarCluster provides auto-scaling for SGE-based clusters, by monitoring the cluster queue.

2.3.2 Open-source Clouds

In the Open-source cloud landscape there are a few solutions that provide partial support for auto-scaling. The Nimbus Phantom protocol [135] provides a partial implementation of the Amazon auto-scaling service, but currently it has limited flexibility as there is no support for "alarm triggering". On top of it, Keahey et al. [92] have developed an auto-scaling service. The service is capable to deploy VMs on multiple cloud providers, monitor them and re-deploy them automatically in case of failures. The high availability of the service is ensured by Zookeeper [87]. The authors mention that cusomized scaling policies can be supported, policies that can increase/decrease the number of VMs based on application performance metrics and user specified preferences regarding the providers to be used. The Carina Environment Manager [109], built on top of OpenNebula, provides support for automatic application deployment and auto-scaling. Each application runs in an environment, managed by an "environment manager" that applies scaling policies. Policies can be specified in Ruby and are similar to Amazon's EC2 metric alarms. Scalr [151] is an open-source cloud platform with auto-scaling functionality for web applications. Users can select the desired monitored metrics and the thresholds used to trigger the scaling through a web interface.

2.3.3 PaaS

At the cloud PaaS layer there are many solution both commercial and research. To better understand the state of the art at the PaaS cloud layer, Table 2.2 gives an overview of the main solutions and outlines their auto-scaling support.

These systems provide runtime support for applications hiding from users the complexities of managing the underlying resources. Nevertheless, these solutions usually provide closed environments, forcing users to run only specific application types (e.g., web, MapReduce). If new programming frameworks appear, the PaaS provider needs to develop first the necessary support on its infrastructure, and then offer to the users the possibility to use it. Finally, these solutions offer poor auto-scaling support: only a few major comercial platforms allow users to define rules to scale automatically their applications. In this last

Solution	Auto-scaling support	Application type
AWS Elastic Beanstalk [8]	yes	web applications
AppFog [14]	?	web applications
AppScale [15]	no	web applications
CloudControl [44]	no	web applications
CloudFoundry/Stackato [45]	no	web applications
Heroku [81]	no	web applications
Mendix [116]	no	web applications
PiCloud [43]	no	batch (scientific) applications
OpenShift [130]	yes	web applications
Cloudify [46]	yes	web applications
Windows Azure [19]	yes	web applications MapReduce
ConPaaS [136]	yes (specific application types)	web applications MapReduce task farming
Aneka [28]	yes	web applications batch applications task farming

Table 2.2: Summary of PaaS solutions.

case, auto-scaling is done based on resource utilization metrics and does not consider the application performance.

2.3.4 Summary

Auto-scaling is an important feature for cloud systems, which was highly regarded by the scientific community due to the increasing dynamicity of scientific applications. Current PaaS solutions that provide auto-scaling and auto-scaling services offered by the commercial provides rely on the "infinite capacity illusion", which, with the increasing user base, becomes more difficult to maintain.

2.4 Value-aware Resource Management Systems

In this section we investigate multiple systems that provide abstractions that allow users to communicate to the system the satisfaction they might get from a certain application performance, or result. Previously presented systems used resource management models that provided limited means to users to express the value they might have for their applications. These systems mostly relied on priorities and resource quotas to establish the resource distribution among users and their applications. These mechanisms, however, cannot enforce a rational user behavior, and, thus, provide limited fairness. By beeing aware of the user's value for her application and ensuring that this value is correct, a system can take resource allocation decisions that can lead to a better user satisfaction.

2.4.1 Utility Function-based Systems

The concept of utility function was borrowed from economics, where it is used to specify the preference, or valuation of the buyer for the acquired resources. This valuation, mathematically expressed as a function of some application performance metric, i.e., execution

time, tells how satisfied the user is if the performance metric is met. Utility function-based systems use these functions to ensure a fair resource allocation, assuming that users are cooperative and communicate them truthfully.

Lee et.al. proposed a scheduler that considers utility functions, and schedules applications to maximize the aggregate utility [103]. The utility function expresses the user's satisfaction as a function of the time elapsed from the application submission. Users could "shape" their own utilities as piecewise linear functions.

In the context of datacenters, utility functions were applied by Walsh et.al. [180] to manage the node allocation between multiple applications (e.g., web servers). Walsh et.al used two types of utilities: (i) a service level utility; (ii) and a resource level utility. The service level utility maps the user satisfaction to application performance. For example, for a web server the user can have a value of 1200 for a response time between 0 and 30 milliseconds, which decreases linearly with the increase in the response time. The resource level utility translates this user valuation in required amounts of resources using a performance model of the application. For example, to obtain a response time less than 30 milliseconds, 3 nodes would be required while for a response time of 70 milliseconds only one node is required. The value of this function varies in time, according to the application's workload, i.e., number of requests it needs to process per second. Walsh applied this idea in a software prototype, called Unity. Each application has an application manager that predicts the workload (i.e., requests per second) demand and computes its own resource level utility. This function is transmitted to a global entity that periodically computes the number of servers allocated to every application such that the sum of all utilities is maximized.

The same idea was applied by Bennani [123]. In this case the global entity sends all possible node configurations to the applications managers, which compute the current utility and return the result. Then, the global entity chooses the configuration that maximizes the weighted sum of all utilities.

Van [125, 173] improved the solution of Walsh [180] by using classes of VMs instead of nodes, adding a provider objective (i.e., maximize the number of nodes that can be turned off) and optimizing the number of VM migrations. Their solution relies on Entropy [80]. Entropy is a VM manager that expresses the VM placement in a cluster as a Constraint Satisfaction Problem and uses a Constraint Solver to obtain the optimal VM-to-node mapping.

2.4.2 Lottery Scheduling

Lottery scheduling [179] is a well-known implementation of the proportional-share policy for operating systems. Proportional-share was used by operating systems schedulers to share resources between a dynamic number of tasks. Given that each task has an associated weight, the worse resource share (e.g., CPU time) it receives over a period of time is proportional to its weight and inversely proportional with the sum of weights of other running tasks. For example, let's take two applications A and B that want to use the CPU resource of a physical node. The application A has a weight of 1 and the application B has a weight of 2. In this case, A receives 33% CPU time and B receives 66%. Proportional-share policies lead to *dynamic resource allocation* for applications: their resource share varies in time according to the resource contention. Thus, starvation is avoided and resource utilization is maximized. Starvation is avoided as each task receives a share of the contended resource. Utilization is maximized, as all the resource amount is divided between the running tasks (if some tasks do not use their share it gets redistributed between the others).

Lottery scheduling relies on two interesting abstractions: tickets and currency, which provide hierarchical resource management capabilities. In this policy applications are funded with tickets that can be issued in different currencies, used to manage priorities among groups of tasks. These currencies are converted afterwards to a "base" currency, used by the scheduler to compute the resource share for each task; the resource share that each task receives is proportional with its number of tickets issued in the base currency. This is achieved by holding a "lottery" at each scheduling period. The lottery starts with selecting a random number. Then, to determine the winner, the algorithm traverses a list that keeps all the tasks and sums the number of tickets hold by each task. The winner of the lottery is the task for which the sum goes above the initial random number. Support for space-shared resources (i.e., memory) was also introduced through "reverse lottery": the more tickets a task has the lower the probability of loosing a resource unit is.

Lottery scheduling leads to flexible resource allocation, as lottery tickets can be "inflated", (thus, increasing the application's resource share), or "deflated" (thus, reducing the application resource share) or exchanged (allowing for cooperative resource management among applications).

2.4.3 Market-based Systems

The idea of applying market-based mechanisms dates back to 1968, when Sutherland applied an auction to distribute compute time on the PDP-1 computer to users [166]. Users were assigned different amounts of currency, which they used to bid for a time slot by marking the desired amount to be spent on a board. The slots were reserved in advance to the users with the highest bids and a telephone message was sent to them to inform them of their reservation. Since then, a variety of solutions were proposed to use a market to schedule jobs on clusters [68, 178, 163, 38, 191, 37, 17, 155], to schedule jobs on grids [25], to run parallel applications [142], to allocate storage in peer-to-peer networks [50], database storage [164] or network bandwidth [102].

Compared with previous resource management models, a market has two advantages in managing distributed resources:

Increased user satisfaction A market can lead to an overall increased user satisfaction in two ways. First, a market allows users to express their value for acquired resources in a fine-grained manner and enforces these values afterwards. As opposed to traditional priority-based systems, which force users to assign predefined priorities to them, this aspect allows users to better express the urgency of their applications and, thus, it also allows the system to make better scheduling decisions for them.

Second, a market introduces the notion of cost and gives users incentives to take judicious decisions regarding the resource allocation they desire for their application. Other systems that do not use the notion of market, rely on the use of priorities. In such systems, users are inclined to cheat regarding the priority of their application: users that are assigned high priority by the administrator could run less urgent applications in periods with high resource demand, as they do not have any incentives not to do so. Knowing how much users truthfully value their resource allocations allows the system to decide the most efficient resource distribution.

Decentralized resource control A market provides decentralized resource control: users and providers are autonomous entities, which act independently of each other to meet their own performance objective. It is true that a centralized solution can be designed to support these objectives too, but such a solution could proove to be

complex and difficult to maintain. Even if it might not lead to an efficient global solution, a market opens up the possibility to support a variety of objectives for both users and providers at a low complexity cost: *users and providers communicate through prices*, which is a generic enough mechanism to allow each of them to meet their own objective.

In the rest of this section we analyze two commonly used market models: commodity markets and auctions. These types of markets differ one from another in the way the market clearing price is computed. This market clearing price expresses the fact that the user demand matches the resource supply.

2.4.3.1 Commodity Market-based Systems

In commodity markets the resource price is established using demand and supply functions and both consumers and providers buy and respectively sell at this price. These market types rely on algorithms to adjust the resource price to reduce the excess demand close to zero, also called tatonnement algorithms [34]. These algorithms either use estimations of the excess demand, as in G-commerce [185], or they rely on the participants to send their demand as a function of price like in On-Call [127]. In this last case, the solution would be impractical in a large-scale system.

One particular case is Libra [155], which brings a different model: it applies a proportional-share policy on each node to allocate CPU proportionally to the application's deadline while a global pricing mechanism, based on the infrastructure utilization, is used to balance supply with demand. Admission control ensures that the jobs accepted in the system do not lead to deadline misses for the other jobs. What it is interesting here is the use of the proportional-share policy that maximizes the infrastructure utilization, as opposed to a case in which jobs are allocated exclusive-access to nodes. Aneka [192] implements a dynamic pricing model too, but this time by using advance reservations as a substrate resource allocation model. Unfortunately, there is one aspect that these works did not considered: as the price is set at the beginning of the execution, applications coming in the system in a low demand period will be charged with a small price, while urgent applications coming later, might not get all the resources they need for their execution.

2.4.3.2 Auction-based Systems

Auctions are encountered in a large variety of situations from real life, from selling real properties to travel tickets. An auction is an economic mechanism to allocate a set of items among a group of interested entities. This process begins by having all the participants, also called bidders, submit their bids to the auctioneer. Then the auctioneer decides two things: (i) which bidders get the resources; (ii) and how much they should pay for them. Auctions are simple-to-implement mechanisms that clear the market fast; they allow users with the most urgent demand to get their resources with smaller delays than in a commodity market.

Based on the visibility of the bids and the policy used by the auctioneer to decide the resource price, there are a variety of auction types. If all the bids are visible to the bidders than the auction is "open-bid", otherwise they are called "sealed-bid". Sealed-bid auctions are more frequently applied in distributed systems as they are faster than open-bid auctions and provide better incentives to users to declare truthful information.

Auction-based resource allocation Auctions were used to efficiently allocate distributed resources to users [37, 17, 194]. In the cloud computing landscape there are

two efforts of allocating VMs through an auction. Zaman et.al. [194]) uses a combinatorial auction, i.e. auctions which allow users to submit the same bid for combinations of resources, to allocate combinations of VMs from different classes. SpotInstances [89], a service introduced by Amazon to lease its unused capacity to users, also relies on an auction: users submit bids for VMs and their VMs are provisioned to them as long as the VM price is below the bid. In this case, the user decides when to run her application on resources and how to cope with price volatility, e.g., the user risks to be outbidded at any-time. Nevertheless, their auction model is not public.

Combinatorial auctions were used for clusters to allocate sets of nodes like in Mirage [37], or CPU time like in Belagio [17]. The users specify how much they want to bid, the time duration and the desirable combinations of nodes. The infrastructure scheduler selects the users which get the resources and the amounts allocated to each of them.

Auction-based job scheduling At the application level, multiple attempts were made to use auctions to schedule "jobs", usually static MPI applications, on clusters [163, 191] or to run bag-of-tasks applications on grids [2]. In this case, users submit jobs with associated bids to a scheduler, and the scheduler decides which job gets to run based on the output of the auction. No attempts are made to change the application allocation in time. The output of such mechanism is an increased overall user satisfaction. This comes from the fact that the bids assigned by the users to their applications reflect the valuation the application execution has for them. An auction, which assigns resources to the highest bids, ensures that the most valuable applications are running on the infrastructure.

Spawn Spawn [178] represents a first step in giving more flexibility to users in designing applications that adapt dynamically their resource demand to the market fluctuations. Spawn [178] considers tree-like dynamic applications that are funded by users at a specific rate, funds which are then distributed down the tree, among the application's tasks. Each task uses its funds to bid for a CPU time "slice", with a variable size, on the nodes. Basically, each node runs a "sealed-bid second-price" auction (i.e., the price at which the resource is sold is the bid of the next-highest looser bid). The funding mechanism gives freedom to applications *to adapt their resource demands to the market conditions*: when the infrastructure is free, the applications can expand and when the infrastructure is contended the applications shrink. Nevertheless, as interesting as this aspect may be, besides price volatility, Spawn suffers from scaling limitations, as auctions are run on each node. All tasks are required to communicate bids to a node, and then wait for the node to accept or reject their bid.

Popcorn Popcorn [142] is similar to Spawn regarding the concept of assigning budgets to applications and letting them adapt to price fluctuations. Popcorn provides a paradigm to write parallel applications in Java by splitting their computation in "computelets" for which the program buys CPU time on nodes participating in a market (the access to the market was done through a web page).

REXEC REXEC [38] is a market-based decentralized system to execute serial and parallel applications on a cluster. This solution implements a different type of "auction", derived from the traditional proportional-share resource allocation policy. Contrary to other auction types, this mechanism allows a direct resource allocation to the participants in the auction proportional with their bid. Users specify a maximum rate their application is willing to pay for CPU time and each node "sells" its CPU time on the market using a

proportional-share policy: the CPU time each buyer receives is proportional with its rate and inversely proportional with the sum of all competing rates, i.e., the total rate of the node. Applications are submitted to the nodes with the minimum total rate. Intuitively, applications with higher rates receive higher shares.

Tycoon Tycoon [100] uses the same proportional-share allocation model as REXEC: a traditional proportional-share scheduling algorithm runs on each node, augmented with the notion of cost. To run their applications in Tycoon, users first need to create accounts on each node on which their application runs and specify two values for it: a "bid" and a time interval. The bid divided over the time interval represents the user's willingness to pay for CPU time and it can be modified over time. As in this situation the user is required to submit a bid for each node eligible to run her application, the authors propose a "Best Response" algorithm, which distributes a user specified budget over a set of nodes based on user's preferences (i.e., weights).

The Dynamic Priority Scheduler from Hadoop A dynamic priority scheduler is proposed for Hadoop [149], which uses the same auction type to distribute map/reduce slots between users. In Hadoop, jobs are composed of map and reduce tasks. A scheduler decides the mapping of these tasks to slots, i.e., a partition of the physical node. The original schedulers of Hadoop allow priority management among groups of users by deciding to allocate the slots to the job's tasks based on the queue it was submitted. Sandholm et.al. extended this concept by assigning a queue per user and allowing the user to tune its priority dynamically. This can be done by assigning to each user a "queue priority budget", and allowing her to spend it over time by changing its spending rate, i.e., the user's willigness to pay for a slot and for a time period. The queue priority budget and the spending rate allow the user to control how many slots get assigned to her jobs over time, and thus, to control the job execution time.

2.4.4 Summary

We investigated in this section several mechanisms that might allow resource management systems to be more aware of how much users value their applications.

Utility functions allow users to specify their valuation as a function of the application performance. Although utility functions allow designing flexible resource management systems, there are several drawbacks in using them. First, utility-based managers still rely on a global scheduler that decides the resource allocations for each application, and as a consequence, these systems are centralized. Moreover, selecting the optimal configuration is a complex problem, limiting the solution space to the ones that rely on coarse-grain allocation. Finally, such systems do not ensure proper resource regulation among users. Resource allocation fairness is translated in maximizing the minimum application utility, or the sum of all utilities. Nevertheless, this system-wide objective relies on users to be cooperative, or altruistic, and communicate their real utility.

Lottery scheduling allows a flexible and modular resource management through the concepts of tickets and currencies. This opens the door for dynamic application adaptation policies, as applications can adjust their resource demands dynamically in time by inflating/deflating their ticket amounts or cooperate with each other to reach a maximum resource utilization. Nevertheless, in this context, there is no explicit control on the ticket inflation or deflation, and efficient resource utilization can be achieved only in an environment where users are altruistic and trust each other.

Market-based resource management is a well-studied topic in the context of distributed systems. Works that apply market mechanisms focus on providing better user satisfaction than traditional resource management systems, usually batch schedulers. This goal is met by a market as it forces "selfish" users to put a right value on the resources their applications consume, and to communicate these values to the resource management system. For example, a user will put more value on the execution of a highly urgent application than on the execution of any other tasks. A resource management system that is aware of these values can decide which applications should run such that users benefit the most from their execution, i.e., the user social welfare is maximized. Thus, markets meet our "fair resource utilization" requirement.

We investigated two commonly used market implementations: commodity markets and auctions. Commodity markets set the resource price based on the demand and supply, through the use of tatonnement algorithms. For urgent applications requiring fast access to resources, this process might prove slow. In contrast to commodity markets, auctions clear the market fast. Thus, a variety of solutions applied them for job scheduling and resource allocation. In this case, either the user submits a bid for her job, and the allocation is decided at the job start, or the user submits the bid for the set of resources, also static during the requested user interval. From these solutions we investigated the ones that can be applied for dynamic resource sharing among applications. However, the systems applying it neither consider application adaptation, nor user SLOs. Thus, more work remains to be done to fullfill our "flexibility" requirement.

2.5 Conclusion

In this chapter we analyzed the current state of the art in managing cluster resources. We started this analysis from envisioning a resource management system capable of meeting three objectives: SLO support flexibility, resource allocation efficiency and fairness. We defined the SLO support flexibility as the capacity to support different application types and SLOs, resource allocation efficiency as the capacity to maximize the resource utilization and fairness as the capacity to regulate the application access to resources.

We analyzed multiple existing management systems for clusters capable to provide dynamic resource allocation to applications. We consider this capability as a first step in SLO support flexibility.

We investigated batch schedulers, which rely on the job abstraction to allocate resources to applications. These systems use a coarse-grained resource allocation model, where the node is the basic resource allocation unit, mostly to provide maximum performance for applications running on the infrastructure. These systems suffer from important limitations regarding the SLO support flexibility, as they have poor support for dynamic applications, poor isolation among applications and limited SLO support.

Besides batch schedulers, we investigated multiple dynamic resource management frameworks. These frameworks address the dynamicity aspect of applications, improving their performance, and they also decentralize the resource control, allowing applications to adapt autonomously and leading to a flexible resource management. Nevertheless, they lack isolation among applications and SLO support.

Moving in the cloud landscape, we investigated the use of virtualization technologies in resource management. By virtualizing the infrastructure, applications can run in customized and isolated environments and thus the resource management becomes more flexible. In this context we surveyed multiple works. These solutions have the potential to meet our objectives as they can support different application types and allocate resources

among them dynamically. Nevertheless, either they focus on one SLO and application type, e.g., web servers, or they leave the policy space largely unexplored. This is also the case of cloud computing providers, which offer PaaS services, as they do not regulate the application access to resources and do not provide SLO support.

Besides these resource management systems, which rely mostly on traditional methods to allocate resources, we analyze two main methods used by a resource management system to make sound resource allocation decisions and to distribute resources among applications based, not only on their SLO, but also on how much users actually value their execution. These two methods are: (i) the use of utility functions; (ii) and the use of currency.

The utility function-based systems assume a cooperative behavior among applications and do not provide sharing incentives to users. Users might overstate their utility in order to get higher resource allocations. Moreover, current implementations rely on a centralized resource manager and require the application to communicate its utility function each time its resource demand changes. Needless to say, such implementations might suffer from scalability issues.

In contrast to utility function-based mechanisms, the use of currency as a mean to express user's or application's value for resources can lead to a flexible and scalable resource management. Market-based methods, as they rely on currency as a communication mechanism, treat each application as a selfish entity, seeking to follow only its own objective. Contrary to previous resource management systems, market-based solutions address the fairness requirements concerning resource utilization. Although many market-based solutions exist, they focused more on how to allocate resources among users and analyzing the satisfaction users get from using them. Our goal is to use the charateristics provided by markets to allow applications to adapt independently while providing fair resource utilization. Autonomous application adaptation was not a focus of previous market-based solutions.

Based on these reasons, we propose a solution which uses a market-based resource allocation policy to ensure a fair resource regulation mechanism while providing flexibility to users to design dynamic resource demand adaptation policies and meet their SLOs. At the same time, the market-based resource allocation policy provides fine-grained resource allocation, maximizing the infrastructure utilization. We introduce this approach in the following chapters.

Chapter 3

A Market-based SLO-Driven Cloud Platform

This chapter details our approach. We have outlined in Introduction the main requirements users and infrastructure administrators would have from a resource management platform: flexibile SLO support, maximum resource utilization and fairness. The objective of this thesis is to investigate and propose methods to achieve these requirements. To meet this objective we propose a *SLO-driven* cloud platform that manages applications through autonomous controllers which allocate resources to them dynamically. We designed this platform to manage private shared infrastructures.

This chapter is structured as follows. Section 3.1 gives a brief overview of the proposed platform. Section 3.2 details the main design principles and Section 3.3 presents an overview of the platform's architecture. Section 3.4 summarizes the chapter.

3.1 Overview

Our system relies on the use of an IaaS cloud to manage the hardware resources of the private infrastructure. The IaaS cloud provides users with isolated and controlled software environments, allowing them to install all the required software dependencies for their applications with no involvement from the administrator.

The key characteristic of our system is the use of a market mechanism to provision virtual machines and regulate the amount of resources applications can get. Market-based approaches were applied in a variety of contexts in distributed systems ranging from clusters, grids, peer-to-peer systems and to clouds too. Users are assigned currency amounts, which reflect the priority the user has in the system. Users can distribute these currency amounts between their applications, reflecting the true importance their execution has for the user. To run applications on the infrastructure, users start a virtual platform composed of a set of virtual machines and a controller that manages it. The controller manages the total cost of executing the application and provisions VMs from the market by submitting bids for them.

Using a market has two advantages. First, it allows users to take better decisions regarding how much resources their application uses, improving the overall user satisfaction. Second, the market provides a *decentralized resource regulation* mechanism that allows each application to control its own resource allocation according to its user-assigned priority. Basically, resource demand adaptation policies can be designed using the price as a feedback signal while virtual currency distribution policies ensure a fair adaptation.

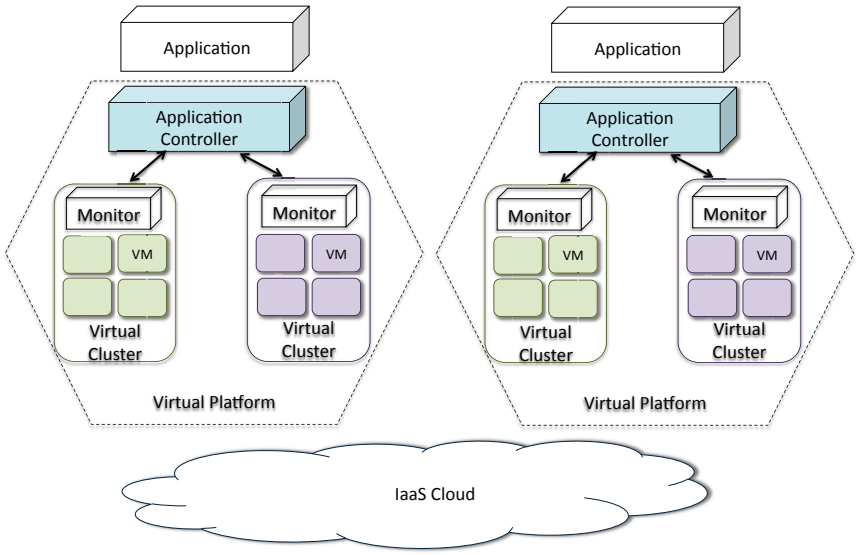


Figure 3.1: The architecture of a virtual platform.

The way our system implements the market approach and how applications adapt on it is a unique contribution: our system provides *fine-grained resource management* and allows users to design *vertical and horizontal resource demand adaptation policies* to meet their SLOs given the current resource load and the priority the application has for its user.

3.2 Design Principles

We describe now the main design decisions we took in implementing our system. Our system relies on four design principles:

- each application runs in its own private virtual platform, which can scale up and down according to the application resource needs;
- the access to resources is controlled using market-based allocation policies;
- the resource allocation is done using a proportional-share algorithm;
- the implementation is modular and extensible.

3.2.1 Autonomous Private Virtual Platforms

To support different application types and user-defined objectives, our system runs each application in its own virtual environment, called a virtual platform. Users can install whatever packages are needed for their application with no interference from the infrastructure's administrator.

Definition 3.1 *A virtual platform is an autonomous entity composed of one or more virtual clusters and a controller, also called application controller, that manages them on behalf of the application.*

Definition 3.2 *A virtual cluster is a group of VMs that have the same resource configuration and use the same base VM image.*

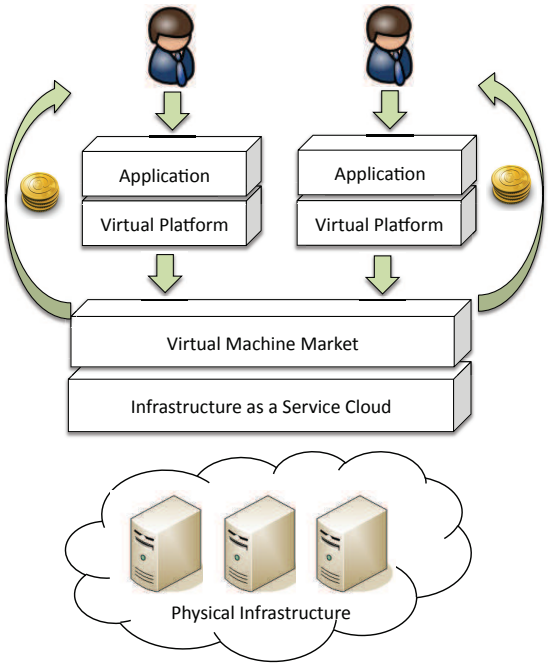


Figure 3.2: The use of a cloud market.

Figure 3.1 illustrates the architecture of a virtual platform. To meet user-defined objectives, the application controller running in the virtual platform interacts with the application, adapts dynamically the virtual platform resource demand to the infrastructure load and the application's performance and reconfigures the application. Each virtual cluster has a monitor that sends application performance metrics to the application controller.

From this point of view, systems like VIOLINS [144] are similar to ours. In their case, however, there was one central entity that took decisions regarding the allocation of all virtual platforms. As in our system each virtual platform controls its own resource allocation, different user-defined resource demand adaptation policies can co-exist on the infrastructure. Although different policies could be implemented in a centralized resource control solution, such a solution would require extensive modifications each time a new policy or a new application would need to be added. With this principle in mind, we can position our system on top of the IaaS cloud layer.

3.2.2 Market-based Resource Regulation Mechanisms

To ensure a fair resource utilization, our system uses a market. Using a market to manage resources of a distributed infrastructure was a well-studied problem in the context of clusters and grids [190]. Markets are efficient in automatically balancing the user demand with the infrastructure resource supply. The reason why markets became so popular in the distributed systems community is due to the notion of *cost*, which makes users more aware of how many resources to acquire for their own use. By using a market, users have incentives to assign the right priorities to their applications; thus applications with urgent resource needs can get resources in time even in high load periods, while applications with less urgent resource needs run in low load periods. Moreover, markets provide decentralized resource control, which is advantageous, as we have already explained while introducing the previous design principle.

Figure 3.2 details how the market is implemented by our system. To run applications on

the infrastructure, users are assigned budgets by a bank in the form of a virtual currency. We use a *credit* as a currency unit. This currency is internal to the organization and is controlled by the infrastructure administrator using an *open loop* policy. The currency is rechargeable, meaning that the charges incurred by the market return to user accounts after a while. The currency management acts as a mechanism to manage the priorities among users over long time intervals. Users distribute budget amounts to their applications, reflecting the maximum cost the user is willing to support for running her application, or the true priority the user has for her application. Resources like CPU and memory have a price set through market policies. Limited by its budget amount, the application and the virtual platform in which it is running adapt their resource demand to meet user-defined objectives. In our system each virtual platform adapts its resource demand considering the current resource prices. The resource price is dynamic and fluctuates based on the resource demand. To implement this dynamic resource pricing mechanism our system uses an auction. The implementation details of this auction are given in Chapter 4.

3.2.3 Fine-grained Resource Allocation

To provide fine-grained resource allocation at a low complexity, our system uses a *proportional-share auction*. This type of auction is derived from the classic proportional-share allocation mechanism used by operating-systems schedulers to allocate CPU time to user tasks proportional to their priorities. In our system, the proportional-share auction is applied to allocated CPU and memory resources to VMs. By using a proportional-share auction, VMs can receive a share from the infrastructure capacity proportional to their submitted bids and inversely proportional with the sum of all running bids. These resource allocations are fractional and dynamic, as they vary in time according to the total infrastructure demand.

3.2.4 Extensibility and Ease of Use

We designed our system to be generic enough to allow developers to implement other resource demand regulation mechanisms with minimal modifications to the system components. To ease the user's task of running and scaling elastically her application, our system provides a set of predefined controllers and resource demand adaptation policies. Moreover, a generic and simple API allows users to design their own controllers with policies different than the ones provided. We discuss further details regarding the proposed resource demand adaptation policies in Chapter 4.

3.3 Architecture Overview

We applied these design decisions in the architecture of a platform composed of four main components, as described in Figure 3.3. Users interact with our system through a generic interface, composed of a set of CLI tools and APIs that can be remotely called. The Application Management System automates the execution of applications on the infrastructure. This component handles the tasks of provisioning virtual clusters and deploying applications on them, and also monitoring and scaling elastically the applications. The VM Scheduler implements the resource regulation algorithms needed to limit the amount of resources allocated to each application. The Virtual Currency Manager implements the credit distribution policies and is thus in charge of limiting the amount of resources that a user can use over long time intervals. Finally, the IaaS Cloud Manager handles all the operations regarding virtualization of physical resources.

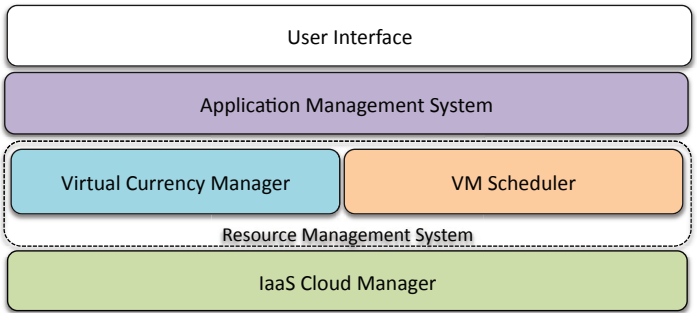


Figure 3.3: System Overview.

We validated this architecture in two contexts:

in simulation We implemented a scheleton of this architecture in CloudSim, an event-driven Java simulator. The resulted prototype is called Themis. We simulated the performance of the system in terms of user satisfaction when each running virtual platform has to adapt its resource demand to cope with a user-given SLO. We compared the performance achieved with our system with two other well-known policies: FCFS and EDF. FCFS is a policy used by current cloud resource managers while EDF is a policy that allocates resources to applications by knowing their SLO (i.e., deadline).

on a real-world testbed We implemented the architecture in a real prototype, called Merkat. As a IaaS Cloud Manager we used OpenNebula. We tested Merkat on the Grid'5000 testbed with different scientific applications, reproducing real life scenarios found in an organization like EDF.

3.4 Conclusion

In this chapter we proposed a platform for dynamic resource allocation and application management on clouds. Our system relies on the concept of autonomous virtual platforms to isolate applications and scale them dynamically according to their load and user requirements. We build our system to be easily extensible by users with their own resource demand adaptation policies. To regulate the resource amounts users get in contention periods, our system uses a market, ensuring that the most urgent applications get the resources they need. Moreover, users are forced to behave correctly through virtual currency management.

We designed predefined application controllers and resource demand adaptation policies, which can be used by users that are not willing to design their own resource demand adaptation policies. These controllers are designed for static and dynamic applications. We detail in the next chapter the design of these policies, together with the resource management model.

Chapter 4

Resource Allocation Model and Dynamic Application Adaptation

This chapter introduces the resource management model we use in this dissertation. As we previously mentioned, we consider that a resource management system should meet three requirements: SLO support flexibility, resource allocation efficiency and fairness. To meet these requirements, we investigate an approach that relies on the implementation of a market to allocate resources to all running virtual machines on the infrastructure and lets applications to dynamically adapt their resource demand. In this chapter we describe the resource management algorithms that implement the market. We detail the resource allocation policy and we illustrate how applications can adapt themselves to meet different user SLOs.

This chapter is organized as follows. Section 4.1 gives an overview of the resource management process and Section 4.2 describes the theoretical principles behind it. Then, Section 4.3 discusses the design of the resource allocation algorithms. Finally, Section 4.4 describes how applications can adapt their resource demand.

4.1 Overview

In our system, resources are shared among applications through a market implemented using a proportional-share auction policy. By using a market users assign more realistic priorities to their applications: if the system is too loaded, a user that doesn't need to run her application immediately will postpone its execution until a less loaded period,when the execution cost is lower.

Users are assigned budgets of credits, which they use to run their applications on the infrastructure. Each user can execute one or more applications on the infrastructure. In order to do so, a user has two possibilities: (i) to submit a bid for each VM, using the VM Scheduler interface, and manually start her application; (ii) to estimate how much budget she would need to run the application and start a virtual platform which manages this cost and bid for VMs on behalf of the user during the application runtime.

4.1.1 The Proportional-share Market

As we mentioned before, our market is based on a proportional-share auction. This auction involves three steps: (i) VM bid submission; (ii) VM allocation computation; (iii) price computation. Let us focus on these three steps:

VM bid submission To provision VMs, a bid needs to be submitted for their resources: CPU and memory. The initial VM bid and the execution cost of the application can be computed based on past price history and the user's current budget. In our implementation we compute the VM bid based on the current resource price. The bid submitted for a VM is persistent: it can be specified once and is considered by the VM scheduler at each time period during all the VM life-time. Nevertheless, the value of the bid can be modified during the VM runtime to cope with price fluctuations.

VM allocation computation Based on the value of this bid, and the other concurrent bids, the VM Scheduler allocates an amount of CPU and memory to the VM. These amounts are proportional with the value of the bid and the infrastructure capacity, and inversely proportional with the sum of all existing bids. To keep the correct CPU and memory amounts allocated to each VM, the VM Scheduler migrates them between physical nodes. We describe the allocation and migration process in Section 4.3.

Price computation Auctions introduce the notion of price, forcing users to pay for the resources they consume. Each auction has its own strategy to compute the resource price. For example, the price may be the bid of the highest looser, as in the case of second price sealed bid auctions. Our approach relies on the use of a proportional-share auction. In this auction, the resource price is computed as the sum of all bids divided by the total infrastructure capacity. If this price is smaller than a predefined price, i.e., reserve price, than the reserve price is used.

Now, as we previously stated, each virtual platform is managed by an application controller. The application controller receives from the user its SLO and a renewable budget. The application controller is capable to take independent decisions to alleviate the issues that may arise during the execution of an application, like SLO violations, on the market-based IaaS. An application controller might provision or release VMs, suspend or resume them, or change the bid submitted per VM. The application controllers run in an uncoordinated fashion, the only mechanism binding them together being the resource price.

4.1.2 Application Resource Demand Adaptation on the Market

When running the application on the infrastructure, it is not sufficient just to acquire resources at the beginning of its execution. Infrastructure resource availability or application resource requirements might change. Moreover, when provisioning resources from a market with a dynamic price, the user also needs to cope with the price volatility. If the price increases, the current user budget might not be enough to cover the cost of the acquired resources. If the price decreases, more resources could be provisioned, or the user could spend less for the already acquired resources. To optimize the user's budget, we propose two adaptation methods:

Vertical scaling : We propose a method to adapt the bid for each VM to keep the application resource allocation at a level that allows the application to meet its SLO at a given user budget. This is advantageous not only for the user running the application but also for the other users too, which can run more applications on the infrastructure. Applications that might take advantage of this method are static applications, which cannot change the number of processes during their runtime. Nevertheless, this method is useless when the cost to keep the application desired allocation becomes higher than the user's supported cost. The user would waste its budget as the

application would not meet its SLO anyway. In this case, the application controller might suspend the VMs and resume them when the cost becomes affordable again.

Horizontal scaling : We propose a method to adapt the number of VMs. Applications that might take advantage of this method are dynamic applications. For example, in the case of a bag-of-tasks, the application controller would release some of its VMs when the price becomes higher than the user's supported cost, and acquire the VMs when the price becomes affordable.

Note that these methods do not ensure full SLO satisfaction, as the probability of having the SLO violated highly depends on how contended the infrastructure is and how much the user's budget is. To provide better guarantees, these policies can be combined with price prediction and statistical resource guarantees [148]. This combination will provide users with hints regarding the initial budget they should spend for their applications and will help the application controller to take better bidding decisions.

4.2 Resource Model

In this section we explain the motivation behind using a proportional-share auction in managing VMs. Then, we detail the theoretical aspects of this auction model and its design and implementation.

4.2.1 Terminology

Before describing the resource model let us first define the terms that we use in the rest of this section:

node capacity We define the node capacity as a resource vector $< c = CPU, m = memory >$. A node with n cores has assigned a maximum CPU capacity $c = n * 100$ resource units. Thus, an application with n processes can consume 100% CPU time on each of the other cores. Regarding the memory capacity, a node has a maximum capacity, m resource units, which is the total available physical memory from which we deduct an amount reserved to the hypervisor. For example, a node with 4 cores and 8GB of RAM for wich we consider that one memory unit equals 1MB, would have a maximum capacity of $< c = 300, m = 7292MB >$ if 900MB is reserved for the hypervisor use.

infrastructure capacity is defined as the sum of all node capacities.

reserve price Each resource unit acquired from the infrastructure has a reserve price. This price is fixed by the infrastructure administrator, e.g., 1 credit per resource unit, and it is used when the infrastructure is underloaded, i.e., the total user demand is less than the infrastructure capacity.

resource price Each resource unit is acquired at a price established by the market. This price is computed based on a proportional-share auction.

VM maximum allocation $alloc_{max} =< c = CPU, m = memory >$ specifies the maximum allocation a VM should receive. A user can set $alloc_{max}$ based on how much she estimates that her VM will use. For example, if the user uses the VM to execute a sequential task which consumes at maximum 900 MB of RAM, she can specify an $alloc_{max}$ of $< max_c = 100, max_m = 900MB >$.

VM share $share =< c = CPU, m = memory >$ specifies the allocation that a VM receives after the VM scheduler applies the auction to compute all the VM allocations while considering the total infrastructure capacity. We discuss in Section 4.2 this auction model.

VM allocation $alloc =< c = CPU, m = memory >$ is the allocation enforced by the hypervisor on the node on which the VM resides. This allocation is obtained from the VM share, capped to the VM maximum allocation, as described in Section 4.3.1.

VM bid $b =< b_c, b_m >$, the bid assigned to a VM, represented by a vector with two values: a bid for each of the VM's allocated resources. This bid is set by the user or by the virtual platform application controller.

4.2.2 Motivation

Our resource model relies on a proportional-share auction. We have three reasons in applying a proportional-share auction to allocate resources to virtual machines:

Fine-grained resource sharing: Current IaaS or virtual cluster based solutions share resources in a coarse-grained manner: users run their applications in fixed-size virtual machines. The virtual machine resource configuration is selected from different "classes" that the resource provider makes available, e.g., in the case of Amazon EC2 [7], or is predefined by the user at VM boot time, e.g., in OpenNebula [158]. We believe that resource utilization can be improved by allowing users to change the resource allocation of their VMs during their runtime. This is useful especially in contention periods as applications not using all resources allocated to a VM can release them for better use, or applications not capable to shrink their number of VMs can run with less resource per VM. Proportional-share policies can provide fine-grained resource sharing by allocating fractional resource amounts to VMs, proportional to their bids. This capability is currently provided by hypervisors. Nevertheless, current IaaS cloud managers provide poor support for it: users don't have any automatic means to dynamically tune the resource allocation of their VMs.

Low implementation complexity: Implementing an auction for fine-grained divisible resources is challenging due to its complexity. If each user wants to provision bundles of resources, that is VMs with different arbitrary maximum allocations, applying an auction to determine the set of winners usually becomes complex. Thus, a market mechanism capable to allow users to provision VMs with arbitrary resource allocations at a low complexity, like the proportional-share auction, is desirable.

Ease of use: A resource allocation policy should be easy to understand and use, in order to allow users to design adaptation policies for their applications. Usually, an auction establishes from the group of bidders a set of winners and a set of losers. While the winners get the resources they bid for, which are in our case the VMs, the losers might starve as their bids are too low. Or, even worse, their VMs might be shutdown without warning, if they were already running on the infrastructure. An example of such an auction is the Amazon Spot Instances [89] provisioning model. In the proportional-share auction model, starvation is avoided because even the users with the lowest bids get a resource share, and in high price periods, instead of killing the requests, only their resource shares decrease. Thus, instead of anticipating the moment their VMs are killed, users can use the decrease in resource share as a signal

regarding the infrastructure contention. Based on this signal they can increase their bids or suspend the VMs if their budget does not allow.

4.2.3 Principles

We define and apply the proportional-share auction on our infrastructure as follows. We consider the infrastructure as a huge physical node. A resource *auctioneer* gathers requests for a resource type in term of *bids* and determines the resource amount that should be reserved on a physical node for each request. In this section we assume that a user makes this request to get virtual machines and run her application in them. Nevertheless, in our system these requests are made by the application controller.

Currently there are two auctioneers: one for CPU cycles and one for memory. To provision a single VM the user needs to send a bid to both auctioneers and to provision a set of VMs the user needs to bid for every VM. As this is highly inconvenient for the user, we provide her with an interface to submit simultaneously a set of bids for a set of VMs. Section A details this interface.

To compute the resource allocation for all VMs, the resource auctioneer applies the following rule:

Definition 4.1 *Given a set of resource bids $b_i(t)$, with $i \in \{1..n\}$ for a time interval t and a resource with a capacity of C units, the auctioneer computes a resource share of a_i resource units over this time interval for each bid i, equal to:*

$$a_i(t) = \frac{b_i(t)}{p(t)}, p(t) = \frac{\sum_i^n b_i(t)}{C} \tag{4.1}$$

where $p(t)$ is the price for a unit of that resource.

Inversely, to obtain some guarantees that a VM i will receive an allocation for the next time interval equal to $a_i(t+1)$, by knowing the current resource price $p(t)$ and assuming that this price doesn't change in the next time interval, the bid $\hat{b}_i(t+1)$ can be computed. This bid is computed as follows:

$$\hat{b}_i(t+1) = \frac{a_i(t+1)}{1 - a_i(t+1)/C} \cdot p(t) \tag{4.2}$$

The allocation obtained for this bid can be different from the previously requested $a(t+1)$, as other VMs might be started or stopped at the same time interval as i. To provide higher resource allocation guarantees several works use statistical methods [148] and admission control [150].

Note that each user pays the same price for a resource unit, but the number of resource units that it receives depends on her bid. The resource price and shares are recomputed and enforced at every time interval t.

4.3 VM Management

This section describes our approach of implementing the proportional-share auction to allocate resources to VMs on the infrastructure. VMs are provisioned and managed by the VM Scheduler service. Figure 4.1 gives a generic overview of the service's architecture. The VM scheduler performs the following functions: (i) it collects data regarding node and VM resource utilization; (ii) it deploys the VMs on the physical nodes; (iii) it enforces the VM

resource allocations computed for the current time period; (iv) it migrates VMs between the nodes. Migration is required as VMs receive a resource share that changes in time, due to other VMs being started or shut down. To enforce the VM resource allocations, the VM scheduler starts a control daemon on each physical node. This daemon receives allocation commands from the VM scheduler and transmits them to the node's hypervisor.

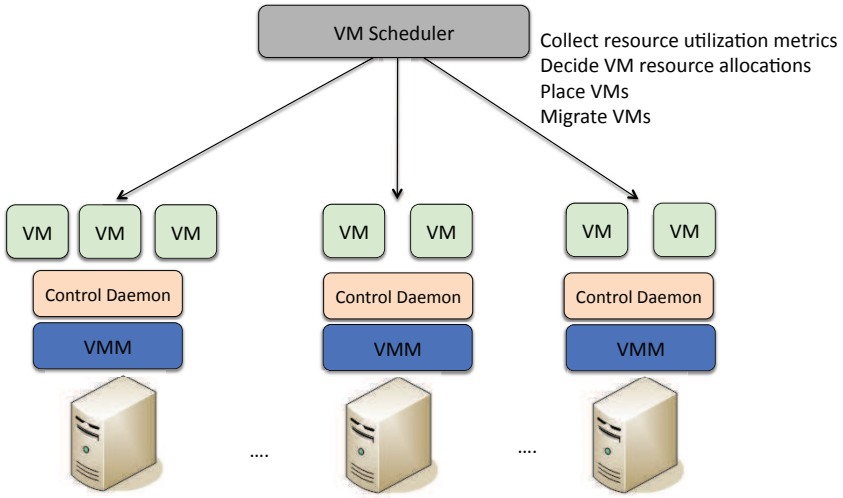


Figure 4.1: VM Scheduler overview.

4.3.1 VM Allocation

To request a VM, a user, or an application controller, submits to the VM Scheduler a bid vector (a bid for each of the VM's allocated resources) together with the maximum resource amount allocated for the VM. Thus, the request for one VM has the following form: $<< b_c, b_m >, < alloc_{maxc}, alloc_{maxm} >>$. This request is *persistent*: the system will keep the VM running and allocate resources for it until the user explicitly requests to stop the VM. The bid request is accepted by the system under the following two constraints: (i) the bid value for each resource needs to be higher than the reserve price; (ii) the VM maximum allocation should not exceed the node capacity.

Based on the requests received from the users, the market scheduler computes the VM allocation as described by Algorithm 4.1 in two steps: (i) it computes the VM allocation considering the infrastructure as a physical node; (ii) it corrects the allocations to fit the VMs on the nodes. The allocated amount depends on the VM's bid and the VM maximum allocation.

The algorithm receives as input the list of currently running and submitted VMs and a resource capacity vector (for CPU and memory) and computes the VM allocation. The capacity can be of a physical node or of the entire infrastructure. A simple algorithm would compute the VM resource allocations using only Equation 4.1. Nevertheless, in this case resources remain idle even if there are applications which might use them. Some VMs might not use their entire share, because their maximum VM allocation is less than it. At the same time, other VMs could use more resources than their share.

Algorithm 4.1 VM resource allocation using proportional share.

```
1: ComputeVmAllocations(vms, capacity)
2: alloc ← ∅    // The allocation vector for all VMs.
3: avail ←< 0, 0 >    // Remaining node available capacity.
4: for r ∈ {cpu, memory} do
5:    avail[r] ← capacity[r]
6:    currentVms ← vms    // temporary copy of vm list
7:    for vm ∈ currentVms do
8:       alloc[vm.id][r] ← 0
9:    while avail[r] > 0 and currentVms ≠ ∅ do
10:      sumbids ← 0
11:      for vm ∈ currentVms do
12:         sumbids ← sumbids + vm.bid
13:      price_local ← sumbids / avail[r]
14:      avail[r] ← 0
15:      for vm ∈ currentVms do
16:         alloc[vm.id][r] ← alloc[vm.id][r] + vm.bid / price_local
17:         diff ← alloc_max[vm.id][r] − vm.alloc_max[r]
18:         if diff > 0 then
19:            alloc[vm.id][r] ← vm.alloc_max[r]
20:            avail[r] ← avail[r] + diff
21:            currentVms ← currentVms \ {vm}
22: return alloc
```

Let us take the following example. We consider 1 node, with a capacity of $< cpu = 800, memory = 10 >$ (the node has 8 cores) and 2 VMs: the first one has a maximum allocation of $< cpu = 200, memory = 5 >$ and the second one has a maximum allocation of $< cpu = 400, memory = 5 >$. Now, if the first VM has a bid of 20 credits for each resource and the second VM has a bid of 10 credits, with Equation 4.1, the first VM would receive a share of <cpu=533, memory=6.66> while the second VM would receive a share of <cpu=267, memory=3.34>. Needless to say, this resource distribution is not efficient: the first VM does not use all its share while the second VM could use more than what it gets.

To avoid this case, the available resource capacity is distributed between the virtual machines by also considering the VM maximum allocation. The result is *alloc*, a vector containing the *actual resource allocation* each VM gets. In our example, $alloc =<< cpu = 200, memory = 5 >, < cpu = 400, memory = 5 >>$.

4.3.2 VM Migration

As the system load or the bids change in time, to ensure an allocation corresponding to the submitted bids, VMs might need to be migrated between nodes. Otherwise, some nodes might not be able to accommodate the new VM resource demands while other nodes might have idle resources. This process is called load balancing.

Motivation

To illustrate the need of load balancing, Figure 4.2 describes a situation in which the VM allocation changes from one scheduling period to the next one. In this example we consider 3 physical nodes with a capacity of 100 CPU units and 10GB RAM which host two applications: one application runs in 3 VMs while the other one in 2 VMs. Each

VM has a maximum allocation of 100 CPU units and 20.48 RAM units. The VMs of the first application run at a bid of 10 CPU credits and 20 RAM credits, while the VMs of the second application run at a bid of 30 CPU credits and 60 RAM credits. Using Equation 4.1, this configuration leads to an allocation of 33 CPU units and 20.48 RAM units for the first application and 100 CPU units and 20.48 RAM units for the second one. Thus, the VMs of the first application are placed on the first node while the VMs of the second application are placed on the second node. Nevertheless, the application running in the last 2 VMs finishes and the VM scheduler re-computes the VM allocations: this time, each VM from the first node is allocated its maximum allocation. As the current node cannot accommodate the VMs anymore, two of them need to be migrated to the other nodes.

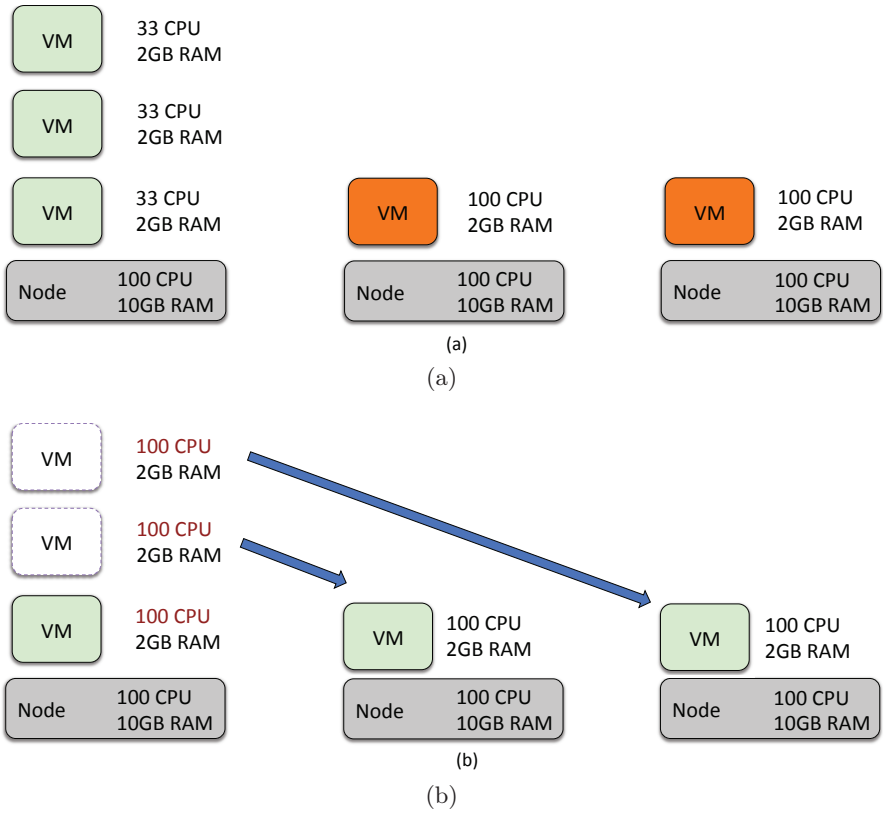


Figure 4.2: Migration example: (a) shows the initial configuration; (b) shows the migrations needed to ensure the ideal VM allocation.

Nevertheless, there are cases in which the VM Scheduler cannot allocate to each VM its ideal allocation, and thus an allocation error needs to be supported for each VM. This case is explained in Figure 4.3, which considers the previous scenario but with a slight modification: the VMs of the first application run at a bid of 12 CPU credits instead of 10 CPU credits. Thus, the first three VMs receive an allocation of 37.5 CPU units while the last two VMs receive an allocation of 93.75. As these "ideal" allocations cannot be enforced, the VM allocations need to be recomputed. To solve this issue, the VM scheduler applies the proportional-share algorithm, described by Algorithm 4.1, on each physical node. Thus, after applying the algorithm, the first three VMs receive 33.3 CPU units while the last 2 VMs receive 100 CPU units. The difference between the ideal and the real VM allocation is the allocation error.

To explain the allocation error, we give the following definitions:

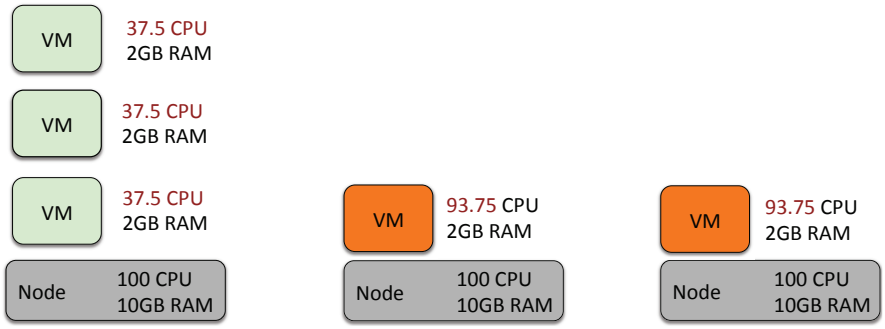


Figure 4.3: Vm allocation error example.

$alloc^r$ is the resource allocation a VM receives from the capacity of the current node, proportional to the bids of all the VMs running on this node.

$alloc^i$ is the resource allocation the VM receives from the capacity of the infrastructure, proportional to the bids of all the running VMs. This is the ideal allocation of the VM.

Definition 4.2 *The allocation error of a VM is:*

$$e_r = \frac{alloc^i - alloc^r}{alloc^i} \tag{4.3}$$

Algorithm 4.2 illustrates how this error is computed for the running VMs. The algorithm calls the $ComputeVmAllocations$ method (explained in Algorithms 4.1) to compute the ideal allocations VMs get from the total infrastructure capacity, and the allocations VM would get from their current placement on the physical nodes. Then, the algorithm applies Equation 4.3 to compute the allocation error for the CPU and memory resources of all running and submitted VMs.

Algorithm 4.2 VM error computation.

```
1: ComputeErrors(vms, nodes)
2: e ← ∅    // Vector of errors
3: capacity ← ∑_{node∈nodes} node.capacity
4: ai ←ComputeVmAllocations(vms, capacity)
5: for node ∈ nodes do
6:     ar ← ComputeVmAllocations(vms, node.capacity)
7: for vm ∈ vms do
8:     for r ∈ {cpu, memory} do
9:         e[vm.id][r] ← (ai[vm.id][r] − ar[vm.id][r])/ai[vm.id][r]
10: return e
```

Load Balancing

The load balancing process is done with the goal to minimize the allocation error supported by the VMs, and thus to obtain for each VM a resource allocation as close as possible to the ideal allocation. As having a high number of migrations leads to a performance degradation for the applications running in the VMs, the load balancing process tries to make a trade-off between the number of performed migrations and the VM allocation error.

Algorithm 4.3 details the load balancing process. The algorithm receives the list of current nodes, *nodes*, running VMs, *vms*, VMs to be booted at the current scheduling period, *newvms*, and three thresholds: (i) maximum allocation error supported for the VMs, E_{max}; (ii) maximum number of migrations performed by the algorithms, M_{max}; (iii) maximum number of iterations performed by the algorithm, N_{iter}. The first two thresholds limit the number of migrations performed by the algorithm. The last threshold ensures that the algorithm stops: a solution is generated even in cases in which the VM allocation error cannot be reduced below E_{max}. Based on this information the algorithm computes a migration plan, containing the VMs to be migrated and their destinations, and a deployment plan, containing the nodes on which new VMs should be deployed.

To select the VMs to be migrated at each scheduling period, the algorithm relies on a tabu-search heuristic [71]. Tabu-search is a local-search method for finding the optimal solutions of a problem. Tabu-search works starting from a potential solution and trying to find the best solution incrementally. The algorithm is composed of a series of iterations; at each iteration an improved solution is searched by altering the current solution. To avoid being stuck in a suboptimal solution, tabu-search uses a list that memorizes the last changes.

Algorithm 4.3 tries to reduce the maximum allocation error among all the VMs when its value is above the threshold E_{max} by moving VMs from their original nodes to other, more suitable, nodes. Note that, as each VM has two allocated resources, the allocation error is a vector. Thus, to reduce the error to a scalar value, the algorithm computes the maximum between the vector values. For example, if a VM has a memory allocation error which is higher than the CPU allocation error, the algorithm tries to find a node which gives a better memory allocation. The starting solution of the algorithm is computed from the current VM-to-host mapping by placing the newly requested VMs. These VMs are first placed on the nodes with the largest available resource amount.

At each iteration the algorithm tries to improve the solution by moving the VM with the maximum allocation error that is not in the tabu list to the physical node that minimizes it and with enough capacity to host the VM (Lines 15-20). The move is registered in the tabu list to avoid reversing it in the next iterations (Line 21). If the move improves the maximum allocation error and the number of migrations required by the current solution is below M_{max}, then the move is applied to the current solution (Line 30). Otherwise, the algorithm proceeds in performing additional moves until there is no improvement for the last N_{iter} iterations. Finally, the migration and deployment plan are computed (Lines 35-40).

4.4 Virtual Platform Resource Demand Policies

We define two methods used by virtual platforms to adapt their resource demand on the proportional-share market. We call these two methods *vertical* and *horizontal* scaling. Vertical scaling is done by adjusting the VM bids and horizontal scaling is done by provisioning/releasing VMs. Besides these methods, in cases in which the application runs with a low budget, virtual platforms can use suspend/resume policies to avoid high price periods.

4.4.1 Motivation

One use case for these resource demand adaptation methods is adapting the application resource demand to market conditions so that the user's SLO is met with a minimum

Algorithm 4.3 VM placement algorithm.

1: **ComputePlacement**($nodes, vms, newvms, N_{iter}, E_{max}, M_{max}$)
2: $migrationPlan \leftarrow \emptyset$
3: $deploymentPlan \leftarrow \emptyset$
4: $nIterations \leftarrow 0$
5:
6: **for** $vm \in newvms$ **do**
7: $node \leftarrow$ find less loaded node from $nodes$
8: $node.vms \leftarrow node.vms \cup \{vm\}$
9: $solution_{old} \leftarrow nodes$
10: $solution_{best} \leftarrow nodes$
11: $tabu\_list \leftarrow \emptyset$
12: e = ComputeErrors(vms, nodes)
13: $e_{best} \leftarrow \max_{1 \leq i \leq n} \max\{e_{ic}, e_{im}\}$ // *Find maximum between CPU and memory errors*
14: **while** $nIterations < N_{iter}$ and $e_{best} > E_{max}$ **do**
15: $(vm, e_{max}) \leftarrow$ find vm with $e_{max} = \max_{1 \leq i \leq n} \max\{e_{ic}, e_{im}\}$, $vm \notin tabu\_list$
16: $source \leftarrow vm.node$
17: $destination \leftarrow$ find node which minimizes e_{max}, $node \notin tabu\_list$
18: $vm.node \leftarrow destination$
19: $source.vms \leftarrow source.vms - \{vm\}$
20: $destination.vms \leftarrow destination.vms \cup \{vm\}$
21: $tabu\_list \leftarrow tabu\_list \cup \{(vm, source)\}$
22: e = ComputeErrors(vms, nodes)
23: $e'_{max} \leftarrow \max_{1 \leq i \leq n} \max\{e_{ic}, e_{im}\}$
24: $nMigrations \leftarrow 0$
25: **for** $node \in solution_{old}$ **do**
26: **for** $vm \in node.vms$ **do**
27: **if** $vm.node \neq node$ **then**
28: $nMigrations \leftarrow nMigrations+1$ // *Count the total number of migrations required to reach the current solution*
29: **if** $e_{best} - e'_{max} > 0$ and $nMigrations < M_{max}$ **then**
30: $solution_{best} = nodes$ // *Keep the best solution so far*
31: $e_{best} \leftarrow e'_{max}$
32: $nIterations \leftarrow 0$
33: **else**
34: $nIterations \leftarrow nIterations + 1$
35: **for** $node \in solution_{old}$ **do**
36: **for** $vm \in node.vms$ **do**
37: **if** $vm.node \neq node$ **then**
38: $migrationPlan \leftarrow migrationPlan \cup (vm, vm.node)$
39: **for** $vm \in newvms$ **do**
40: $deploymentPlan \leftarrow deploymentPlan \cup (vm, vm.node)$
41:
42: **return** $(migrationPlan, deploymentPlan)$

resource cost. These policies use the dynamic resource price as a feedback signal regarding the resource contention and respond by adapting the application resource demand given the user's SLOs. We envision at least three cases in which these policies can be applied.

The suspend/resume policies can be applied by applications with low budget to avoid high contention periods. For example, a user might want to run a best-effort application and, thus, she might want to pay for resources only the reserve price.

The vertical scaling method can be applied by static MPI applications, which may run under a user-given time constraint. To optimize the user's budget, an application controller might tune the bid for its VMs. When the infrastructure is under-utilized it might decrease its bid as there is no point in paying extra for the used resources. When the infrastructure is contended, it can use the "saved budget" and increase its bid to ensure an allocation which is "just enough" to meet the user's deadline.

The horizontal scaling method can be applied by maleable or evolving applications. For example, let us take the case of a malleable application: a user might want to run best-effort bag of tasks on the infrastructure. In this case, the only thing the user would need to do is to assign to its application a budget which is enough for it to get, let's say, 100 VMs at the reserve price. When the infrastructure is under-utilized, the application controller will expand the application by increasing the number of VMs up to 100, while in contention periods it will shrink the application by shutting down VMs. In the case of an evolving application, the application controller will interact with the application and receive signals regarding when to expand its resource demand.

4.4.2 Basic Policies

To run static applications on the proportional-share market, two basic policies can be applied:

suspend the application execution This policy is as follows. The application controller checks periodically a suspend condition. This condition specifies when the application controller cannot ensure a user defined virtual platform metric (e.g., application execution time, VM allocation), due to the market conditions and user's budget limitations. If this condition is true, to avoid cases in which all application controllers would suspend at the same time, which would lead to price oscillations, the application controller delays it suspend for a random wait time interval. At the end of this wait time interval the suspend condition needs to be re-checked to avoid suspending the application when the price drops.

delay the application execution This policy is used by the application controller to start or resume the execution of the application. The application controller computes the initial payment for obtaining a specified amount of allocated resources, e.g., 25% CPU time for each of its VMs, by using the current market price. If this payment is greater than the budget limit, the application controller postpones the execution/resume of the virtual platform with a random amount of time.

4.4.3 Example: Best-effort Execution of a Static Application

As an example of how these policies can be used by an application controller, let us consider an user that wants to execute a static MPI application with $nvms$ VMs at a rate of b credits (meaning that the application receives b credits at the beginning of its execution, and at each scheduling period afterwards) on the infrastructure. This budget is distributed equally

between all the *nvms* VMs. The user doesn't have any SLOs for her application; thus the application is best-effort and she assigns a low budget for it.

Algorithm 4.4 Best-effort application execution

```
1: T_wait = 600
2: bid ← compute initial payment(p, 0.25 · alloc_max, nvms)
3: while bid ≥ b do
4:    sleep for random period between 0 and T_wait
5:    bid ← compute initial payment(p, 0.25 · alloc_max, nvms)
6: start virtual platform
7: while application is not finished do
8:    < alloc > ← get allocation metrics for VMs
9:    a_min ← find minimum value from < alloc >
10:   bid ← compute initial payment(p, 0.25 · alloc_max, nvms)
11:   if a_min < 0.25 · alloc_max and bid ≥ b then
12:      sleep for random period between 0 and T_wait
13:      < alloc > ← get allocation metrics for VMs
14:      a_min ← find minimum value from < alloc >
15:      bid ← compute initial payment(p, 0.25 · alloc_max, nvms)
16:      if a_min < 0.25 · alloc_max and bid ≥ b then
17:         suspend application
18:   if application is suspended then
19:      bid ← compute initial payment(p, 0.25 · alloc_max, nvms)
20:      while bid ≥ b do
21:         sleep for random period between 0 and T_wait
22:         bid ← compute initial payment(p, 0.25 · alloc_max, nvms)
23:      restart application
```

Algorithm 4.4 describes the steps taken to execute the virtual platform. The application starts when the price is low enough to allow each of its requested VMs to get a maximum allocation, $alloc_{max}$. After the application starts, when the price becomes too high to allow each of its requested VMs to receive at least 25% of their maximum allocation, the VMs are suspended. After the application is suspended, the algorithm invokes again the resume policy, to re-start the application when the price drops.

4.4.4 Vertical Scaling

To minimize the cost of execution a user's application with a given SLO (e.g., application execution time, number of requests processed per time interval, etc.), we provide an algorithm that adjusts the resources for each VM based on information regarding the application's performance and allocation. Basically, this can be achieved by adjusting the VM bid, and we called this policy *vertical scaling*.

This algorithm is useful for static applications, which can run with less resource per VM if the user's SLO allows (e.g., batch applications), or for applications with dynamic resource demand per VM (e.g., web servers).

Algorithm 4.5 presents this bid adaptation policy. The algorithm receives as input: (i) the current resource bids (*bid*) and past bid history *history*; (ii) the budget to be spent for the next time period (bid_{max}) (iii) the VM allocation; (iv) the VM share *share*; (v) application performance values (v, v_{ref}) and thresholds (v_{low}, v_{high}). Algorithm 4.5 presents the vertical scaling policy. Its output is the resource bids for the next time period. The given thresholds, together with the application allocation act as an alarm: when the application performance metric traverses them the algorithm takes an action regarding

Algorithm 4.5 Vertical scaling policy

VerticalAdaptation(bid, bid_{min}, $last\_bid\_change$, bid_{max}, $alloc$, $alloc_{min}$, $alloc_{max}$, $share$, v, v_{ref}, v_{low}, v_{high})
$resources \leftarrow \{cpu, memory\}$
$T \leftarrow \left|\frac{v_{ref}-v}{v}\right|$
for $r \in resources$ **do**
  if ($v < v_{low}$) and $alloc[r] > alloc_{min}[r]$ or ($share > alloc_{max}$) **then**
    $bid_{tmp}[r] \leftarrow max(bid[r]/max(2, 1+T), bid_{min})$
    if $last\_bid\_change < 0$ **then**
      $\delta \leftarrow |bid[r] - bid_{tmp}[r]|$
      if $\left|\frac{\delta - |last\_bid\_change|}{\delta}\right| < 0.1$ **then**
        $\delta \leftarrow \delta/2$
        $bid_{tmp}[r] \leftarrow max(bid_{tmp} - \delta, bid_{min})$
      $bid[r] \leftarrow bid_{tmp}[r]$
  if ($v > v_{high}$ and $alloc[r] < alloc_{max}[r]$) or ($alloc[r] < alloc_{min}[r]$) **then**
    $bid_{tmp}[r] \leftarrow bid_{tmp}[r] \cdot max(2, (1+T))$
    if $last\_bid\_change > 0$ **then**
      $\delta \leftarrow |bid[r] - bid_{tmp}[r]|$
      if $\left|\frac{\delta - |last\_bid\_change|}{\delta}\right| < 0.1$ **then**
        $\delta \leftarrow \delta/2$
        $bid_{tmp}[r] \leftarrow bid[r] + \delta$
      $bid[r] \leftarrow bid_{tmp}[r]$
  // bids are re-adjusted due to budget limitations
if $bid[memory] + bid[cpu] > bid_{max}$ **then**
  $w \leftarrow \emptyset$
  if $alloc[r] \geq alloc_{max}[r]$, $r \in resources$ **then**
    $w[r] \leftarrow 0$
    $bid_{max} \leftarrow bid_{max} - bid[r]$
  else
    $w[r] \leftarrow 1 - \frac{alloc[r]}{alloc_{max}[r]}$, $r \in resources$
  for $r \in$ resources with w[r]$\neq 0$ **do**
    $bid[r] \leftarrow \frac{bid_{max}}{\Sigma w[r]} \cdot w[r]$
return bid

the VM bid. When the performance value (e.g., estimated application execution time) is above the upper threshold, v_{high} (e.g., 95% of application reference execution time), or when the application receives an allocation below its specified minimum, the resource bids are increased. To optimize the cost per VM, when the allocation for one resource reaches the maximum, the bid increase for that resource stops. When the performance value drops below the lower threshold, v_{low} (e.g., 75% of application reference execution time), the resource bids are decreased. To avoid cases in which the application receives a resource amount that is too small to allow it to make progress in its computation, its resource allocation is kept above a minimum value [1].

The value with which the bid changes is given by T (line 2), and it represents the "gap" between the current performance value and the performance reference value. A large gap allows the application to reach its reference performance fast. To avoid too many bid oscillations the algorithm uses the value of the past bid change, i.e., *last_bid_change* in its bid computation process. For example, if the bid was previously increased and at the current time period the bid needs to be decreased with a similar value, the bid oscillates indefinitely. Thus, the algorithm decreases the current bid with half of its value (lines 10, 18).

When the current budget is not enough to meet the performance reference value, bids are recomputed in a way that favors the resource (i.e., CPU or memory) with a small allocation, i.e., the resource with a small allocation represents a bottleneck in the application's progress. We consider that having lower application performance is still useful for the user, e.g., a scientific application finishing 90% of its computation before its deadline. Thus, the bids are increased using a proportional-share policy, where the resource "weight", $w[r]$ is the difference between the actual and maximum allocation of the VM (line 27). If there is a resource with a maximum allocation, the bid distribution becomes straightforward: the algorithm already decreased the bid for this resource, thus the other resource receives the all remaining budget (line 22). [2]

For further cost optimization, when the application performance is not met due to price constraints, the application can be stopped by using the suspend policy. Nevertheless, as applying these policies depends on the application type, we haven't included them in the algorithm.

4.4.5 Example: Deadline-driven Execution of a Static MPI Application

To give an example of how the vertical scaling policy can be used by an application controller, let us consider a user that wants to execute a static MPI application with *nvms* VMs and a budget b, renewed at a time interval t_{renew} with r credits. The SLO requested by the user is to finish the application execution before a deadline. The reasoning behind this policy, is that as long as its deadline can be met, the application can reduce its allocation to minimize its execution cost.

Application Model

We consider the following application model. The application is composed of a fixed number of tasks. Each task requires one cpu core and a specified amount of memory. We don't model the communication between tasks. To finish their execution the applications need to perform a certain computation amount (e.g., 1000 iterations). There is a large number of

[1] Note that in this case the application controller might just as well suspend the application. This decision can be taken in a loop external to the algorithm.

[2] In this case the application controller could decide to suspend the application.

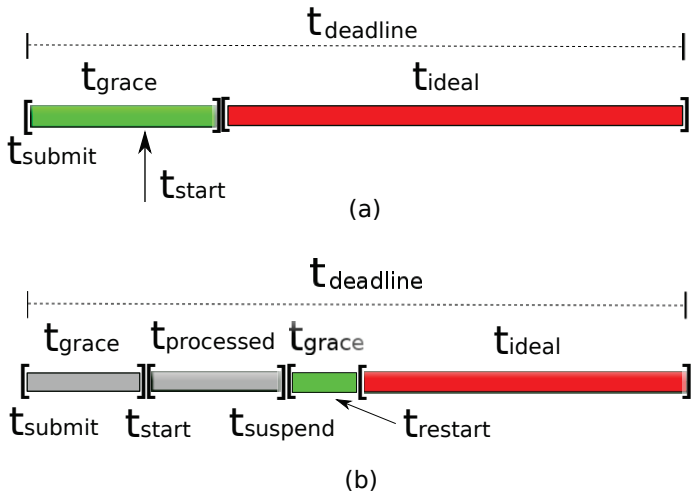


Figure 4.4: The time in which the application can start or resume.

iterative applications (e.g., Zephyr [195]) that follow this model. These applications have a relatively stable execution time per iteration. The execution time of an iteration can be tuned by modifying the resource allocation that each task receives. For example, if each task receives one full core, one iteration can take 1 second. Nevertheless, if the resource allocation drops at half, the same iteration can take 2 seconds.

Terminology

Before explaining how an application controller uses the previously defined policy we give some definitions of the terms used.

When submitting her application, the user needs to provide the ideal execution time of the application, t_{ideal}, the ideal execution time per iteration step, $t_{refideal}$ and the total number of iterations the application needs to perform, n_{steps}.

The terms that define the application performance are:

$n_{current}$ - the number of iterations the application has performed;

t_{step} - the execution time per iteration step;

t_{ref} - the reference time per iteration step: $t_{ref} = \frac{t_{deadline} - now}{n_{steps} - n_{current}}$;

The terms used in computing the risk of not meeting the deadline are illustrated in Figure 4.4:

t_{submit} - the time at which the application is submitted;

t_{start} - the time at which the virtual platform is started on the infrastructure, and thus the time at which the application is started;

$t_{deadline}$ - the remaining time until the deadline set by the user; this value is initially set to $deadline - t_{submit}$; during the application execution is computed as $deadline - now$, where now represents the current time. if the execution time per iteration step is higher than t_{ref} then the application risks to miss its deadline.

t_{grace} - the time interval in which the application can be started or restarted without any risk of not meeting the deadline; When starting the application t_{grace} is computed as $t_{deadline} - t_{ideal}$, as seen in Figure 4.4(a). After the application started executing, t_{grace} - computed as $t_{deadline} - t_{refideal} \cdot (n_{steps} - n_{current})$ as noticed in Figure 4.4(b). As t_{grace} is used now for resuming the application, we are interested in knowing in how much of the remaining time to deadline the remaining computation can be finished.

Algorithm

Now, let us describe the application controller behavior. Algorithm 4.6 gives the main steps executed by the application controller. The application controller starts the application when the resource price is low enough to afford a given allocation for each application VM.

During the application execution, to keep the application execution time below a given deadline, the controller adapts the VM bid. The bid is adapted at each application monitoring period, based on the value of t_{step}. If the t_{step} is faster than the reference $0.75 \cdot t_{ref}$, the bid is decreased. Otherwise, if t_{step} is larger than the reference $0.95 \cdot t_{ref}$ the bid is increased. We use these thresholds to avoid too many bid oscillations, and thus, resource re-allocations.

The maximum limit at which the bid can be increased is given by bid_{max}, by distributing all the application's budget over the remaining time to deadline. We apply this distribution to avoid cases in which the application will run out of credits in the middle of its execution. As we assumed that the application receives a budget renewed with r credits every t_{renew}, the total budget the application receives until its deadline is: $(b + r \cdot (t_{deadline}/t_{renew})$.

As part of this adaptation policy, the application controller checks 3 conditions: (i) if it needs to stop the application execution; (ii) if it needs to suspend the application; (iii) if it can restart the application execution when the application is suspended. These conditions are checked as workload variation on the infrastructure leads to high prices. In this situations the application cannot meet its deadline and it is useless to spend more budget for its execution. The first condition becomes true when the application misses its deadline (Line 13). The second condition happens due to high price periods (Line 24). In a high price period the application controller cannot afford to keep the reference execution time per step, t_{ref}, and thus it suspends the virtual platform. Finally, the third condition is true when the price drops at an affordable value. If the application cannot resume in t_{grace} time then the controller stops its execution (Line 39).

4.4.6 Horizontal Scaling

Some users, instead of scaling vertically the resource allocation of a VM, may prefer to set a fixed allocation for it and to scale the number of VMs instead. For example, an elastic framework (e.g., Condor, Hadoop) might make better use of VMs with fixed CPU allocations (e.g., 1 core per VM). Such a framework might expand its resource demand when the infrastructure is free and shrink it when the infrastructure is contended. In this case it would be more convenient to give a hint to the user on how many VMs she can provision in the next time period of the application execution. Algorithm 4.7 describes the method to obtain this hint. Because the proportional-share allocation policy does not provide allocation guarantees, to keep an allocation for each VM closer to the maximum allocation the resource bid needs to be adjusted to resource price fluctuations. This bid value depends on the user's available budget and the prices for two resources (CPU and memory).

Algorithm 4.6 Deadline-driven application execution

1: $T_{wait} = 600$
2: bid ← compute initial payment(p, $0.75 \cdot alloc_{max}$, nvms)
3: **while** $bid \geq b$ **do**
4: sleep for random period between 0 and T_{wait}
5: bid ← compute initial payment(p, $0.75 \cdot alloc_{max}$, nvms)
6: $t_{grace} \leftarrow t_{deadline} - t_{ideal}$
7: $p \leftarrow$ get current price
8: **if** $t_{grace} < 0$ **then**
9: stop application
10: $last\_bid\_change \leftarrow bid$
11: start virtual platform
12: **while** application is not finished **do**
13: **if** $t_{deadline} < 0$ **then**
14: stop application
15: get t_{step}, n_{step}
16: compute t_{ref}
17: $(< alloc >, < share >) \leftarrow$ get allocation metrics for VMs
18: $t_{step}, n_{current} \leftarrow$ get application execution time per step and current number of steps
19: $alloc \leftarrow$ compute $min < alloc >$
20: $share \leftarrow$ compute $min < share >$
21: $bid_{max} \leftarrow (b/t_{deadline} + r \cdot t_{renew})$
22: $bid \leftarrow$ VerticalAdaptation(bid, bid_{min}, $last\_bid\_change$, bid_{max}, $alloc$, $alloc_{min}$, $alloc_{max}$, $share$, t_{step}, t_{ref}, $0.75 \cdot t_{ref}$, $0.95 \cdot t_{ref}$)
23: $last\_bid\_change \leftarrow bid$
24: **if** $t_{step} > t_{ref}$ and $bid \geq b$ **then**
25: sleep for random period between 0 and T_{wait}
26: get t_{step}, n_{step}
27: compute t_{ref}
28: $bid \leftarrow$ VerticalAdaptation(bid, bid_{min}, $last\_bid\_change$, bid_{max}, $alloc$, $alloc_{min}$, $alloc_{max}$, $share$, t_{step}, t_{ref}, $0.75 \cdot t_{ref}$, $0.95 \cdot t_{ref}$)
29: **if** $t_{step} > t_{ref}$ and $bid \geq b$ **then**
30: suspend application
31: **else**
32: $last\_bid\_change \leftarrow bid$
33: **if** application is suspended **then**
34: bid ← compute initial payment(p, $0.75 \cdot alloc_{max}$, nvms)
35: **while** $bid \geq b$ **do**
36: sleep for random period between 0 and T_{wait}
37: bid ← compute initial payment(p, $0.75 \cdot alloc_{max}$, nvms)
38: $t_{grace} \leftarrow t_{deadline} - t_{refideal} \cdot (n_{step} - n_{current})$
39: **if** $t_{grace} < 0$ **then**
40: stop application
41: restart application

Algorithm 4.7 VM upper bound computation

1: **GetUpperBound**(P, $nvms_{max}$, bid_{max}, $alloc_{max}$)
2: $resources \leftarrow \{cpu, memory\}$
3: find maximum value of N $\in [1, nvms_{max}]$ for which $\sum_{r \in resources} \cdot \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}} < bid_{max}$
4: $nvms \leftarrow N$
5: $bid[r] \leftarrow \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}}$, $r \in resources$
6: return (nvms, bid)

The algorithm receives the resource prices, $P[r]$, the requested number of VMs by the user, $nvms_{max}$, and the user budget, bid_{max}. The output of the algorithm is a number of VMs, such that each VM might receive an allocation close to a maximum for all its resources at the current price. The value of the bid for each resource can be easily computed from Equation 4.2 by replacing a_i with $N \cdot a_{max}$, where N is the number of VMs and a_{max} is the maximum allocation per VM. To find the upper bound on the number of VMs, the algorithm performs a binary search between 1 and $nvms_{max}$ by checking at each iteration if the sum of bids the controller needs to submit is less than its budget.

Algorithm 4.8 Horizontal scaling adaptation policy

```
1: HorizontalAdaptation(nvms_old, bid_max, alloc_max, v, v_low, v_high)
2: if v > v_high then
3:     nvms ← nvms + 1
4: if (v < v_low then
5:     pick vm to release
6:     nvms ← nvms − 1
7: (N, bid) ← GetUpperBound(nvms, bid_max, alloc_max)
8: if N < nvms then
9:     release (nvms - N) VMs
10: else
11:    request new nvms − nvms_old VMs
12: return nvms
```

The user can compute the number of VMs required to meet its application performance goal using the output of the Algorithm 4.7. To illustrate the use of this policy, Algorithm 4.8 describes a straight-forward method that scales horizontally the application. The algorithm provisions VMs as long as a reference value (e.g., execution time, number of tasks) is above a given threshold and releases them otherwise. To ensure that the VMs receive a maximum allocation given the user's budget constraints, the maximum number of VMs for the next time period is computed using Algorithm 4.7.

4.4.7 Example: Elastic Execution of a Task Processing Framework

To illustrate the use of a horizontal policy, we consider a user that wants to execute bag-of-tasks applications on the infrastructure. The SLO for this application type is to minimize the workload processing time.

Application Model

To manage the bag-of-tasks execution the user uses a task processing framework (e.g., Condor) for which it assigns budget at a rate b. The framework has its own scheduler that manages the framework's resources. This scheduler receives application submission requests, keeps them in a queue and dispatches them to nodes when resources become available. At any time, the user can retreive information about the number of running and queued tasks.

Algorithm

To minimize the workload completion time for this framework, we designed a horizontal policy that scales elastically the framework's resource demand. The policy, summarized by Algorithm 4.9, uses Algorithm 4.8 to provision/release VMs. It provisions extra VMs as long as there are tasks in the framework's queue and the framework's budget, b, is enough.

If the controller cannot afford the current number of VMs or no tasks are left in queue, the controller releases them.

Algorithm 4.9 Condor horizontal scaling

1: $ntasks$ = get queued tasks
2: (bid, nvms) = HorizontalAdaptation($nvms, b, alloc_{max}, ntasks, 0, 5$)

4.4.8 Hybrid Scaling

In cases in which applications can scale their demand both horizontally and vertically, a combination of the Algorithm 4.5 and Algorithm 4.8 can be applied.

Algorithm 4.10 illustrates a possible combination. The application scales its resource demand first vertically. Then, when the vertical scaling cannot meet the SLO, the algorithm switches to the horizontal scaling policy. There are two cases when the switch happens. The first case is when the reference value cannot be met and the allocation of its VMs is already at maximum. In this case the policy increases its VM number incrementally until the upper bound is reached. The second case is when the actual value is "better" than the reference value and the allocation of its VMs, or its bid, reaches its minimum. In this case the policy decreases the VM number until the lower bound is reached.

Algorithm 4.10 Hybrid scaling adaptation policy

1: **HybridAdaptation(**$nvms_{old}, bid_{min,max}, alloc, alloc_{min,max}, v_{low,high}$**)**
2: **if** ($v > v_{high}$ and $share[cpu] > alloc_{max}[cpu]$ and $share[memory] > alloc_{max}[memory]$) or ($v < v_{low}$ and (($bid[cpu] = bid_{min}[cpu]$ and $bid[memory] = bid_{min}[memory]$) or ($alloc[cpu] < alloc_{min}[cpu]$ or $alloc[memory] < alloc_{min}[memory]$))) **then**
3: return HorizontalAdaptation($nvms_{old}, bid_{max}, alloc_{max}, v, v_{low}, v_{high}$)
4: **else**
5: $bid \leftarrow$ VerticalAdaptation(bid, bid_{min}, bid_{max}, $alloc$, $alloc_{min}$, $alloc_{max}$, $share$, v, v_{ref}, v_{low}, v_{high}, $history$)
6: return ($bid, nvms_{old}$)

Of course, this simple policy might not prove effective in cases when a fast adaptation is needed. Nevertheless, it illustrates how the vertical and horizontal policies can be combined.

4.5 Conclusion

In this chapter we have discussed the design of the resource management policies currently implemented in our approach. To share resources among applications, we use a proportional-share auction to allocate CPU and memory for all provisioned VMs. This policy provides fine-grained resource allocation, improving the infrastructure utilization, and an easy to use and understand interface for users, allowing them to design simple adaptation policies. Nevertheless, the proportional share that each VM receives might change in time, due to other applications running in the system or due to the VM's own resource demand. To ensure a proportional share for each of the resources allocated to a VM, VMs need to be migrated between nodes. To solve this problem, we proposed a migration algorithm, based on a tabu-search heuristic, which keeps the number of VM migrations below a given threshold while tolerating a certain allocation error for VMs.

On top of the proportional-share market, applications can adapt in two ways: (i) vertically, by changing their bids (which are re-considered at each scheduling period) to cope with fluctuations in price; (ii) horizontally, by changing their resource demand (i.e. number of virtual machines) to cope with modifications in their workload (e.g. changes of computation algorithms, additional started modules, etc.). We have illustrated each adaptation type with an application example. We applied the vertical adaptation policy to a static MPI application to change its resource demand in order to meet a deadline with a minimum execution cost. Then, we applied the horizontal adaptation policy to a task processing framework to expand its resource demand when tasks are submitted to it, and to shrink its resource demand when tasks finish processing.

We discuss in the following chapters the applicability and limitations of this resource management through experimental evaluation in a simulated (Chapter 5) and real (Chapter 6) environment.

Chapter 5

Themis: Validation through Simulation

In this chapter we describe the evaluation of our proposed resource model through simulation. The goal of the simulation is to understand the behavior of our system when applications adapt their resource demand. Simulation is often used as a tool to develop and validate new algorithms in the context of distributed systems. Researchers use simulations to validate these algorithms on more comprehensive scenarios than it would be possible, or too difficult to model in reality. In our case, due to time and resource limitations, running a prototype in a real environment for days, or possibly weeks, and with a large number of nodes, would have been unpractical. Thus, we needed a simulator to model the entire software stack and run larger workloads.

Let us summarize here the characteristics of our solution. Users have budgets of credits and want to run multiple applications on the infrastructure. To run their applications, users assign credit amounts to them and an SLO. As their budgets are re-charged, users don't loose the credits if their application doesn't meet its SLO. We consider that these credit amounts express how much users value the execution of their application on the infrastructure. Unlike other market-based solutions, in which, to run applications users assign bids for them [38, 163, 142, 2], in our system applications behave strategically, adapting their resource demand to meet their SLO, at a minimum cost for the user [1]. The system allocates resources through a proportional-share policy, unknowingly of the application SLOs. Intuitively, as each application adapts independently of the others, our system might not be as efficient as a centralized one.

To understand the behavior of our system we implemented Themis, [2] a simulation prototype. With Themis we address the following questions:

- How does the system perform in terms of user satisfaction when their applications behave strategically and adapt their resource demands?
- What is the performance overhead of the application adaptation, considering that application adaptation leads to VM operations?

The first question is important as user satisfaction should be considered in evaluating the efficiency of the system [39]. Other metrics like application wait time, or number of broken

[1] We explained in Chapter 3 that, in reality, each application runs in a virtual platform managed by an application controller. For simplicity, in this chapter we assume that the application and the application controller represent the same entity.

[2] Themis means "law". Themis was the goddess of justice in Greek mythology.

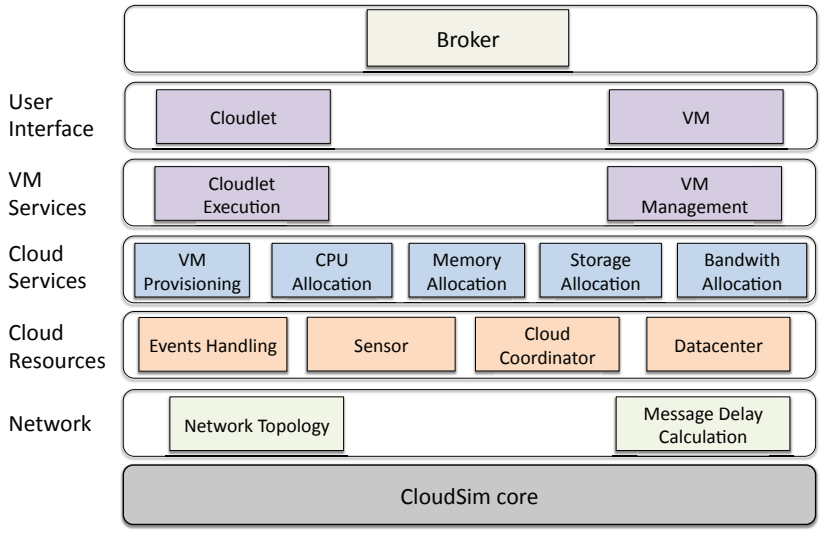


Figure 5.1: CloudSim Overview.

SLOs do not reflect how satisfied are users with using the system as this satisfaction is given by how much a user values the application execution. The second question is important as the number of VM operations might affect the performance of the system and make it unusable.

To answer these questions we modeled three types of users, which expect different results from their applications, i.e., partial and full, and in different conditions, i.e., to a deadline or as fast as possible. We assigned to each user a utility function expressing the satisfaction the user gets from executing her application. We measured the total satisfaction the system can provide for each user type, the number of VM operations and we analyzed the overall application behavior. Simulations with a real workload trace show that even with simple adaptation policies and using only current knowledge, the system behaves well in terms of application performance and number of VM operations (e.g., migrations, suspend/resume).

The remaining of this chapter is organized as follows. Section 5.1 details our prototype. Section 5.2 describes our experimental setup while Section 5.2.4 gives an overview of the simulated application adaptation policies. Section 5.3 presents and discusses our results while Section 5.4 concludes the chapter.

5.1 Prototype

We implemented Themis using CloudSim [27]. We describe next CloudSim and our implementation.

5.1.1 CloudSim

CloudSim is an event-based cloud simulator, implemented in Java, which can be used to implement a variety of policies, from VM scheduling to application dynamic VM provisioning. To better understand CloudSim, Figure 5.1 gives an overview of it. Users can model applications and workload submission scenarios. At the top level, users use Brokers to create applications and model the workload submission scenarios. In CloudSim applications are composed of *Cloudlets*. A cloudlet has a predefined length, specified in millions of instructions per second, (MIPS). To be executed, cloudlets are assigned to VMs; this as-

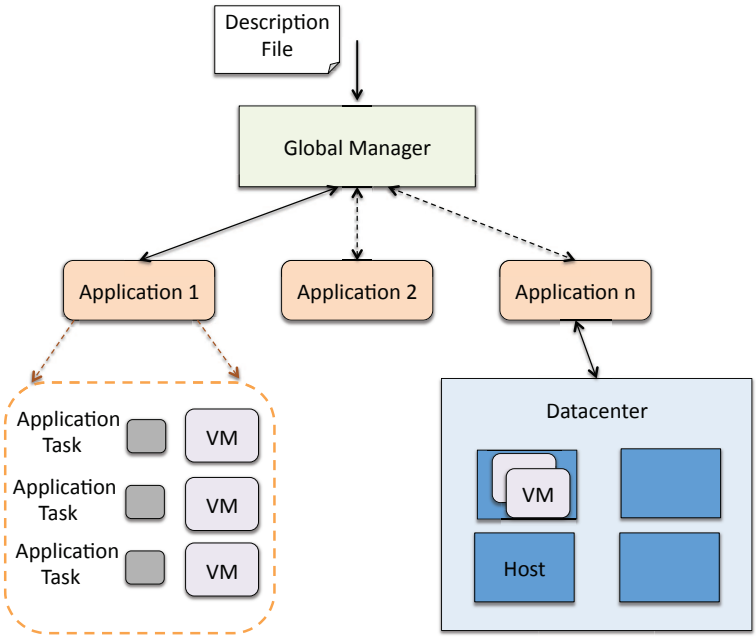


Figure 5.2: Themis Overview.

signment can be done either using a one-to-one mapping, or a many-to-one mapping. Each VM has its own cloudlet scheduler, which shares the processing resource among cloudlets. VMs can be created, destroyed or migrated.

At the IaaS layer, CloudSim can simulate one or more Datacenters. In the latter case, a Cloud Coordinator manages the datacenter access and can be used to design load-balancing and inter-cloud provisioning policies. To monitor specific datacenter performance parameters, e.g., energy, utilization, Cloud Coordinators use Sensors.

Each datacenter has its own VM allocation (provisioning) policy that assigns VMs to hosts. Each host is assigned a predefined resource capacity, that is processing (i.e., CPU, expressed in MIPS), memory, storage and network bandwidth. A host can have multiple cores, each with its own MIPS capacity. Each resource is managed by a provisioner, which allocates an amount of resource to each VM. The processing provisioner can apply a variety of allocation policies to distribute the processing capacity among VMs. Currently implemented policies are space-shared and time-shared allocation.

At the bottom level, CloudSim provides the capability to model the network behavior. In CloudSim entities[3] can send messages to inform each other of specific events, e.g., a broker that wants to create a VM sends a message to the datacenter which allocates resources for the VM. The message latency is modeled using a network topology based on a matrix. This matrix, which contains latencies between different CloudSim entities, can be loaded when the simulation is started.

5.1.2 Themis

Figure 5.2 describes the architecture of Themis. At the beginning of the simulation, a datacenter is created and a Global Manager entity is started. The Global Manager entity uses a XML description file to start the simulation. The simulated environment is composed of

[3] An entity in CloudSim is an instance of a class.

a datacenter, its VM allocation policy, and multiple applications, created dynamically during the simulation run. Applications are created according to their submission times, taken from a workload trace, and are destroyed when they finish their execution. Each application task is a cloudlet. In our simulation, there is no distinction between the application and its application controller. The application applies the resource demand adaptation policies by itself and interacts with the datacenter to change the bids for its VMs. The VM allocation policy is run periodically and applies Algorithm 4.3 to compute the VM allocations and the deployment and migration plan. When VMs are deployed or resumed, the datacenter informs the corresponding applications by sending them a message.

The XML description file, depicted in Figure 5.3, contains the following information:

- type of VM allocation policy (Line 2). Besides our resource model, we have implemented several other "traditional" policies, e.g., FCFS (First-Come-First-Served) and EDF (Earliest Deadline First).

- datacenter characteristics (Lines 3-8). The datacenter capacity is specified: number of nodes and node capacity composed of number of cores, core processing capacity, memory, storage, bandwidth. Besides the total capacity, the user can specify the reserve price for each resource.

- scheduling interval (Line 9). This interval is used by the datacenter to allocate the VMs to nodes.

- a parameter specifying if the workload should be generated or read from a file (Line 10). Currently, Themis has two execution modes: (i) workload generation - it generates the workload for the simulation and it runs the simulation afterwards; (ii) workload execution - it runs the simulation using the workload specified in a file.

  In the workload generation mode, i.e., the *generate* parameter is set to *yes*, Themis can use a *swf* log file to generate a workload based on realistic traces, usually taken from supercomputing centers, or it can generate a synthetic workload. If the workload is realistic, the file type is set to *trace*, otherwise is set to *file*.

  In the workload execution mode, the *generate* parameter is set to *no*, a *repeat* parameter specifies the number of applications to be read from the file and the start position. A *scaled* parameter is used to change the application inter-arrival factor and obtain different contention levels.

- Generated workload characteristics (Lines 12-23). When the workload needs to be generated, there are different parameters that need to be specified, depending of the workload type, i.e., realistic or synthetic. In both cases, applications have a deadline and a budget. The deadline is generated by multiplying the application execution time with a *deadline factor*. The deadline factor is generated using an uniform distribution with a value between a specified minimum and maximum (Line 12). The budget is generated afterwards, starting from a *base* budget (Line 21) chosen using an uniform distribution. This base budget is divided over the deadline factor to express the value the application has for the user. Applications with tighter deadlines have higher value than applications with loose deadlines. Other parameters, like application execution time (Line 17), number of application tasks (Line 18), VM maximum allocation (Line 16), inter-arrival time (Line 19), number of times the batch is re-submitted (Line 20), are specified when the generated workload is synthetic.

```
1: <simulation>
2:    <policy type='proportional' migration='tabu'/>
3:    <datacenter>
4:          <nodes vmm='Xen' cores='1' mips='100' mem='2000' bw='976562' storage='10000000'>
5:                128
6:          </nodes>
7:          <price mips='1' mem='1' bw='1' storage='10'/>
8:    </datacenter>
9:    <schedule time='60'/>
10:   <workload generate='yes' start='0' repeat='100' scaled='1.0'>
11:         <file type='trace' name='HPC2N.swf' urgent_percentage='0' value_ratio='4'/>
12:         <jclass n='1' deadline_min='1.5' deadline_max='10' factor_repeat='1' time_max='0'>
13:               <characteristics type='rigid'>
14:                     <ntasks min='1' max='1'/>
15:               </characteristics>
16:               <vm cpu='100' memory='900'/>
17:               <duration min='1800' max='5200'/>
18:               <size njobs='80'/>
19:               <arrival start='1' factor_rigid_min='60' factor_rigid_max='120'/>
20:               <variation number='2' interarrival='35' repetition='2'/>
21:               <budget min='80000' max='80000'/>
22:               <policy budget='renewable' type='deadline' rate='10000'/>
23:         </jclass>
24:   </workload>
25: </simulation>
```

Figure 5.3: Simulation description file.

– Resource demand adaptation policy (Line 20). We designed several resource demand adaptation policies. To measure their performance, the Themis user can specify which policy type should be used in the simulation, e.g., deadline, performance.

5.2 Experimental Setup

In this section we discuss the experimental setup. As CloudSim does not model the cost of VM operations, we discuss how we express it in Themis. Then we detail the expected user behavior and the workload we used in our evaluation. Finally, we discuss the chosen evaluation metric.

5.2.1 VM Operations Modeling

We implemented in CloudSim a model for several VM-related performance overheads:

Resource allocation We add a performance overhead when the allocated memory is less than the total demanded memory. We assume pessimistically that the application's performance degrades proportionally with the allocated fraction.

VM boot/resume We simulate the VM boot and resume times separately. The VM boot time is modeled by sending the application a message that the VM was created with a delay of 30 seconds. The VM resume time is modeled by assigning to the VM a processing capacity of 0 for 30 seconds.

VM migration We compute the migration time as the time to transfer the VM memory state by using the available network bandwidth of the current host. When computing the available network bandwidth, we consider all the suspend and resume operations

that occur on the considered host at the current scheduling period. We assume that the bandwidth is shared fairly between all the requests. Finally we model the migration performance overhead as 10% of CPU capacity used by the virtual machines in which the application is running. We chose this overhead as previous work found that migration brings 8% performance degradation for HPC applications [124].

5.2.2 User Modeling

User Type	Required Time	Required Results
full-deadline	deadline	full
partial-deadline	deadline	partial
full-performance	as soon as possible	full

Table 5.1: User types.

After studying the needs of HPC users from an organization (e.g. at Electricité de France), we determined two classification criterias: the required time of their application results and the required application results.

Based on the required time of their application results, we found that there are two classes of users:

Deadline users: they want the application results by a specific deadline. For example, a user needs to send her manager the output of a simulation by 7am the next day.

Performance users: they want the results as soon as possible but they are also ready to accept a bounded delay. This delay is defined by the application deadline too. For example, a developer wants to test a newly developed algorithm. She wants the results as fast as possible, but if the system is not capable to provide them, she might be willing to wait until the morning.

Based on the required application results, we also found two classes of preferences:

Full results: to provide useful results the application needs to finish all its computation before its deadline.

Partial results: some users might value *partial* application results at their given deadline; for example, for a user who implemented a scientific method and needs to run 1000 iterations of her simulation to test it, finishing 900 iterations is also sufficient to show the good method behavior.

We combined these classes and obtained three user types, as shown in Table 5.1. We think that these categories are broad enough to be representative for other organizations as well. We omitted the *partial-performance* category as it did not seem realistic.

5.2.3 Evaluation Metrics

The performance of a resource management system can be measured in different ways. Traditional metrics include application wait time, resource utilization or number of missed deadlines. Nevertheless, these metrics do not reflect accurately the total user satisfaction. The satisfaction the system provides to its users, or the total value, is an important metric in showing how well resources are managed.

This satisfaction can be computed if the system knows the value users have for the execution of their applications. To quantify the total user satisfaction, the aggregate user satisfaction can be used. In a market-based system which uses an auction for application scheduling, the user satisfaction can be reflected in the bid she submits to the system [39]. For example, a user who submits a high bid will be more satisfied of the outcome of using the resources than a user who submits a low bid, as her application is also more urgent. In our case, however, applications receive budgets from users, not bids. Thus, we model the user satisfaction as a function of the budget assigned by the user to its application and the application execution time, i.e., a utility function. As there are three different user types, we obtain three utility functions.

Nevertheless, before discussing the signification of utility functions, we define the following terms. t_{exec} is the application execution time. $t_{deadline}$ is the time from the submission to deadline. t_{ideal} is the ideal execution time, i.e., if the application runs on a dedicated infrastructure. $work_{done}$ represents the number of iterations the application managed to execute until it was stopped and $work_{total}$ represents the total number of iterations. B is the application's budget per time scheduling period and per task. B is assigned by the user and reflects the application's importance.

Table 5.2 summarizes the used utility functions. The full-deadline user values the application execution at her full budget if the application finishes before deadline. Otherwise, we express her dissatisfaction as a "penalty", which represents the negative value of her budget. The partial-deadline user is satisfied with the amount of work done until the deadline. Thus the value of the application execution is proportional with this amount. The full-performance user becomes dissatisfied proportional to her application execution slowdown. We bound the value of her dissatisfaction at the negative value of her budget.

User Type	**Utility Function**
full-deadline	B, if $t_{exec} \leq t_{deadline}$, -B otherwise
partial-deadline	B, if $t_{exec} \leq t_{deadline}$, $B \cdot \frac{work\_done}{work\_total}$ otherwise
full-performance	B, if $t_{exec} = t_{ideal}, max(-B, B \cdot \frac{t_d + t_i - 2 \cdot t_{exec}}{t_d - t_i})$ otherwise

Table 5.2: Utility functions.

5.2.4 Application Policies

To meet the SLOs of these three user types, we derive a set of application-specific policies. To ensure the best chance to finish its execution before a deadline, an application starts as soon as the price drops enough so the application can afford a minimum allocation for each VM (e.g. 25% of cpu time). Then, during its execution it applies the different policies according to its SLO. These policies are the following:

Full-deadline: This policy is used by full-deadline users. The policy is actually the Algorithm 4.6. Let us remember it here. Applications start when the price is low enough to ensure a good allocation. During their execution they adapt their bids to keep a low price in low utilization periods and to use as much resource as their SLO allows in high utilization periods. If the application cannot pay for the resources needed to meet its SLO it suspends its execution. In this way it leaves resources for other applications and avoids wasting credits for nothing. The application resumes if the price drops enough to allow it to finish its execution within the deadline. If, during its execution, the application sees it cannot miss the deadline it stops.

Partial-deadline: This policy is designed for partial-deadline users. It is similar to the previous policy as it also uses Algorithm 4.5 to scale vertically according to its SLO. Nevertheless, there are two differences: (i) the application suspends only when a minimum allocation cannot be ensured (e.g. 30% cpu time or 30% physical allocated memory); (ii) as any work done at the deadline is useful, the application does not stop its execution when it sees it cannot meet its deadline anymore.

Full-performance: This policy is designed for full-performance users. The policy is similar to the **full-deadline** policy as it follows the same algorithm. Nevertheless, during its execution, the application, instead of tracking a performance reference metric, tries to keep a maximum allocation given its budget. When the application cannot have a minimum allocation at the current price, the application suspends.

5.3 Experiments

We measure the performance of our system in terms of user satisfaction, number of successfully finished applications and number of VM operations.

To evaluate the system performance we use a real workload trace. Real workload traces are preferred to synthetic workloads as they reflect the user behavior in a real system. Such traces are archived and made publicly available [16]. As a workload trace, we chose the HPC2N file. This file contains information regarding applications submitted to a Linux cluster from Sweden. The cluster has 120 nodes with two 240 AMD Athlon MP2000+ processors each. We assigned to each node 2 GB of memory. The reason behind choosing this trace is the detailed information regarding memory requirements of the applications. Nevertheless, we also considered the case in which applications don't have any memory information. In this case we assign a random amount of memory, between 10% and 90% of the node's memory capacity. We ran each experiment by considering the first 1000 jobs, which were submitted over a time period of 18 days. We scale the inter-arrival time with a factor between 0.1 and 1 and we obtain 10 traces with different contention levels. A factor of 0.1 gave a highly contended system while a factor of 1 gave a lightly loaded system.

We consider that all applications executed by Themis have a deadline and a rechargeable budget. As we couldn't find any information regarding application deadlines, we assigned synthetic deadlines to applications. The budgets assigned to applications are inversely proportional to the application's deadline factor.

5.3.1 User Satisfaction

To measure the total user satisfaction we run the 10 obtained traces for each of our policy. Figure 5.4 describes the proportional-share market performance when each of the different SLO policies are run in terms of total user satisfaction and, for clarity, number of successfully finished applications (i.e., their deadlines were met). Figure 5.4(a) shows the percentage of finished applications for each policy. It is interesting to see that in all cases, the percentage remains almost the same. Nevertheless, fewer applications finish in the case of partial-deadline users, as applications are not stopped before their deadline. Figure 5.4(b) describes the total satisfaction that our system provides to users when applications use the each of the three different policies. The best satisfaction is provided by the partial-deadline policy, as users still gain a positive value from getting the results at their deadline, despite of not having all of them. The worse satisfaction is provided by the full-performance policy, as users are also more demanding: they get dissatisfied really fast and they perceive a negative value if their jobs don't finish at their deadline.

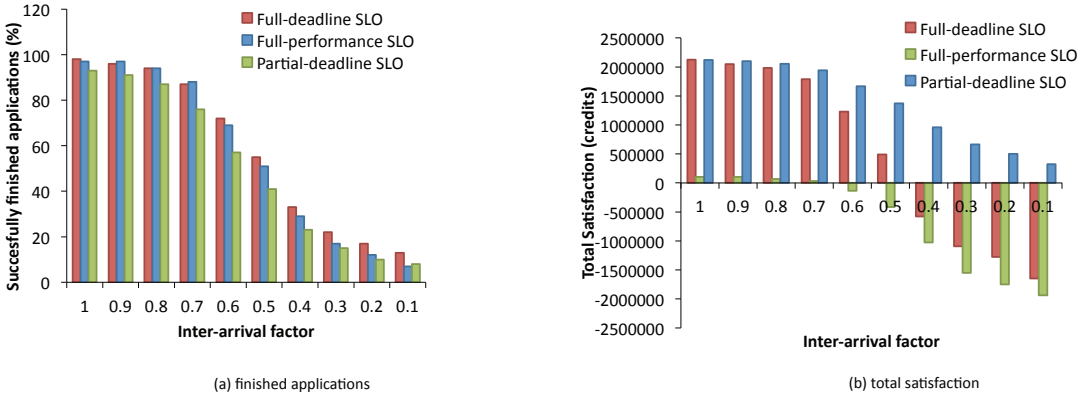


Figure 5.4: Proportional share market performance in terms of: (a) percentage of successfully finished applications and (b) total user satisfaction.

Figure 5.5 describes the proportional-share market performance for our 10 traces, compared to two traditional resource allocation policies: FCFS and EDF. The first policy is usually applied by IaaS cloud managers to schedule VMs: it keeps the requests in a queue and schedules them when resources become available. The second policy is used to minimize the number of missed deadlines. Cloud managers cannot practically apply this model without limiting their support to a predefined set of application goals (e.g., meeting deadlines). Nevertheless, we choose it for comparison purposes: we compare our system with a centralized SLO-aware system. We did not include the full-performance users because, as we have already seen, they are too demanding and the provided satisfaction is poor.

We notice that, again, for partial-deadline users the satisfaction provided by the proportional-share mechanism is overall better than for the other two mechanisms. This comes from the fact that each user perceives positively the fact that her application finished. Nevertheless, when full-deadline users run their applications, the system shows poor performance. What is interesting here is that, when the contention is not high, despite reaching almost the same number of finished applications as EDF, our system performs better in terms of user satisfaction. Nevertheless, its performance degradation increases with the system load.

The performance gap between our mechanism and EDF can be explained as follows. First, as the inter-arrival time decreases, EDF is capable to take better scheduling decisions: thus more applications with smaller deadlines, and in the same time higher budgets, get to run on time. This provides better user satisfaction than our system and FCFS. Second, our system is decentralized: each application acts selfishly and independently from each other in order to meet its own application SLO while with EDF, the central scheduler executes the applications in the order of their deadline, i.e., the application with the smallest deadline is executed first.

To illustrate why this uncoordinated behavior can be bad, let us take the case of an application that suspends and resumes at price fluctuations. When multiple applications suspend, the other running applications receive a higher resource allocation and they drop their bids. This leads to a favorable condition for the suspended applications to resume again, as they notice a price decrease on the market. When the other applications resume, the previously encountered condition repeats leading them to another suspend.

The performance degradation is the "price" payed by the nature of our system, that allows applications to behave selfishly.

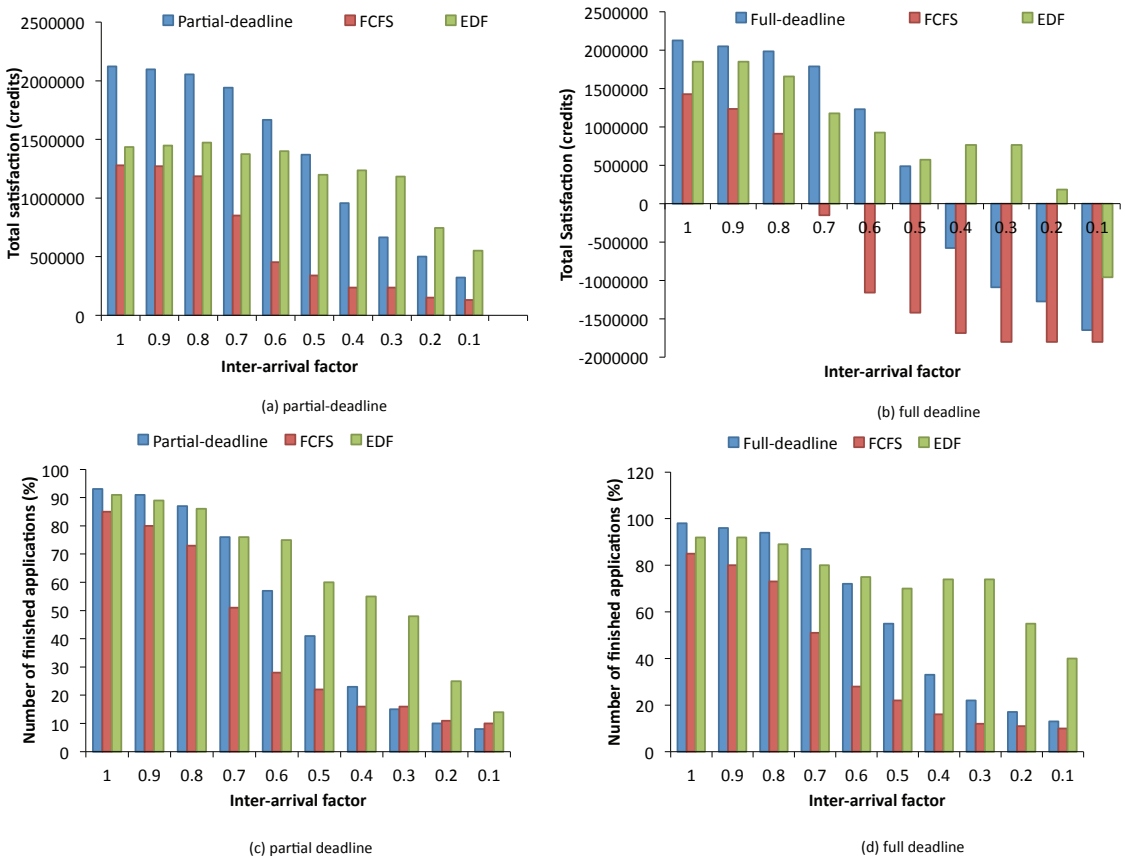


Figure 5.5: Proportional share market performance in terms of: total user satisfaction for the full-deadline (a) and partial-deadline (b) users and percentage of successfully finished applications for full-deadline (a) and partial-deadline (b) users.

5.3.2 Application Adaptation Overhead

As the application adaptation algorithms lead to VM operations, we measure their costs. We keep the same settings from the previous experiment. To perform this experiment we selected the *full-deadline* policy, but any other policy would have been appropriate too. We record the number of migrations and suspend/resume operations during the experiment run.

Figure 5.6 describes the average number of VM operations per hour performed by the VM allocation algorithms. We counted a maximum of 201 VM migrations/hour for a system of 120 nodes, when the system is highly loaded. In the same time there was a maximum of 12 VM suspends/hour (note that this is not the total number, but the average recorded per hour). We notice that the number of VM operations increases with the system load, but in highly loaded periods it decreases. This is explained by the fact that when there is a high load, many applications don't resume their execution.

This fact can be seen in Figure 5.6(b). The highest number of application suspends happened at an inter-arrival factor of 0.6: 268 applications, from which 140 resumed their execution. The number of suspended and resumed applications decreases with the high load, as when there is a lot contention more applications don't even start their execution.

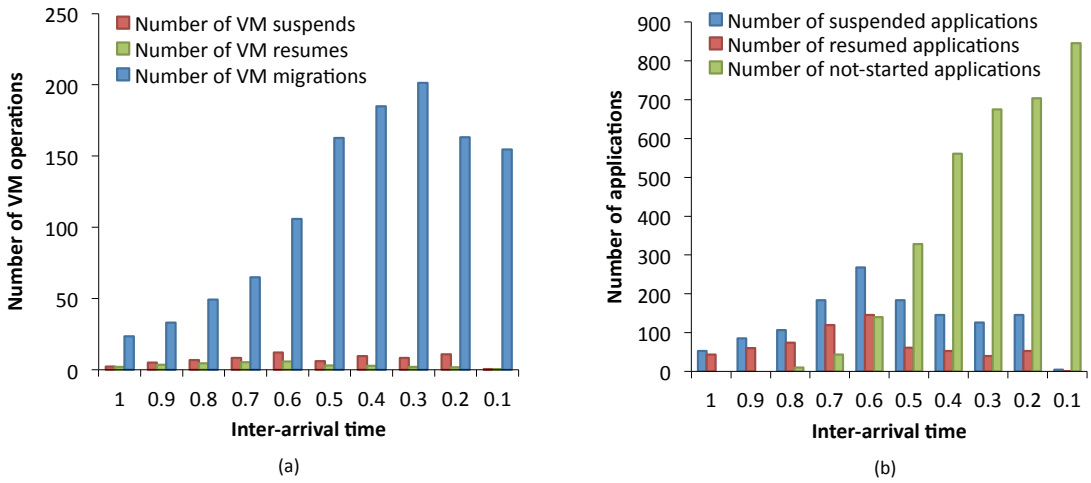


Figure 5.6: Number of (a) VM and (b) application operations.

5.4 Conclusion

In this chapter we presented Themis, a simulation-based prototype of our solution. We have implemented Themis in CloudSim and used it to run a real workload trace. We have analyzed the performance of the implemented proportional-share market mechanism when different user types want to run applications on the infrastructure. We considered a scenario in which each application has its user-given SLO and adapts continuously to meet it, given the infrastructure resource availability.

Our results show that: (i) despite an uncoordinated selfish behavior, compared to centralized resource management policies our system can achieve reasonable performance, measured in terms of user satisfaction; (ii) given the scale of the system, VM operations remain in reasonable bounds. Early results obtained with Themis were published at two international conferences [53, 55] and a workshop [52].

The main issue of this mechanism comes from the selfish application behavior which leads to a performance degradation (or also known as the Price of Anarchy), compared to when applications collaborate or when they operate under strict, centralized control.

The following chapter discusses the real-world validation of our approach.

Chapter 6

Merkat: Real-world Validation

This chapter describes the evaluation on a real testbed of the architecture proposed in Chapter 3. We have implemented our market-based approach in a prototype called Merkat [1]. We designed Merkat to answer to three requirements: SLO support flexibility, efficiency in resource allocation and fairness in resource utilization.

Let us summarize its architecture here. In Merkat, users run their applications in autonomous virtual platforms. These virtual platforms adapt their resource demands by following an application performance objective. The core of a virtual platform is an application controller which takes the resource adaptation decisions. The resource demand of each virtual platform is limited through the use of a market: resources have a dynamic price and users can support limited costs. The applied resource model follows a proportional-share policy. To implement this approach, Merkat is composed of four components: an Application Management System, a VM Scheduler, a Virtual Currency Manager and an IaaS Cloud Manager. We implemented in Merkat two application controllers: (i) an application controller for static MPI applications; (ii) and an application controller for malleable task processing frameworks.

We deployed Merkat on Grid'5000 and tested it with several scientific applications. Our evaluation shows that:

- Merkat allows efficient resource utilization by sharing resources among applications in a fine-grained manner;
- Merkat is flexible enough to allow users to specify different performance objectives for different application types;
- Merkat provides fair differentiation between users with different valuations for their applications, leading to better user satisfaction than traditional resource management systems.

The rest of this chapter is structured as follows. Section 6.1 describes the Merkat's architecture and how applications are managed. Section 6.2 gives some implementation details of our prototype. Section 6.3 describes what applications Merkat currently supports while Section 6.4 details the experimental setup. Finally, Section 6.5 gives the results of our experiments and Section 6.6 concludes the chapter.

[1] Contrary to popular belief, Merkat does not come from switching the "a" and "e" in the word market. It comes from "meerkats", small animals from Africa which form social groups.

6.1 Architecture of Merkat

Figure 6.1 describes the architecture of Merkat. Merkat relies on three services: an Application Management System, a VM Scheduler, a Virtual Currency Manager. These services are built on top of an IaaS cloud manager, which provides basic functionalities for managing VMs and users. Users and administrators can interact with these services through a set of CLI tools or by using the XML-RPC protocol. Merkat is designed to be generic and extensible to support a variety of workloads and resource provisioning policies.

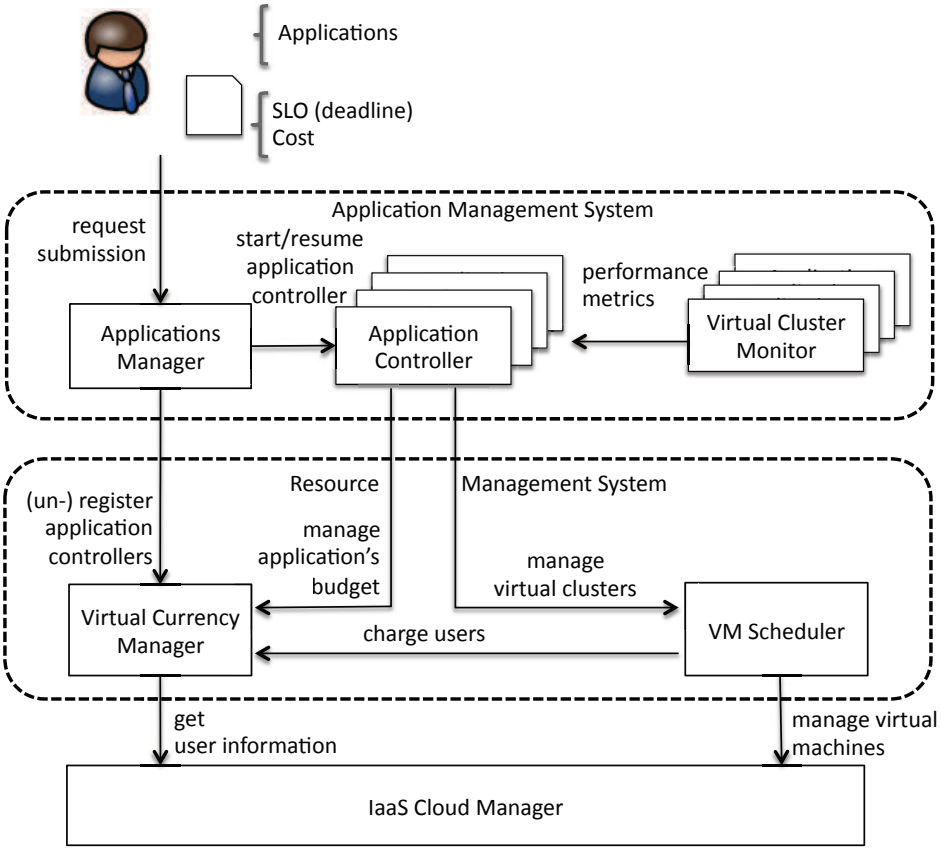


Figure 6.1: Interaction between our system Components.

6.1.1 Components

In this section we describe the Merkat's components.

6.1.1.1 Application Management System

The Application Management System is in charge of executing applications on the infrastructure. This component is composed of an Application Manager and a set of independent application controllers. The Applications Manager acts as an entry point for running applications on our system. Each application runs in a virtual platform, composed of one or more virtual clusters, according to the application components, and managed by an application controller. Each virtual cluster can have its own virtual cluster monitor.

To start an application on the infrastructure, a user submits a request containing a virtual platform template to the *Applications Manager*. The virtual platform template contains information regarding the application controller, adaptation policy parameters and the virtual cluster configuration. We discuss the content of a virtual platform template in Section 6.1.3. Based on the information contained in the virtual platform template, the Applications Manager creates a new *application controller* (i.e., a daemon process) and delegates to it further application management responsibilities.

To be able to provision VMs, each application controller needs to have an account registered with the Virtual Currency Manager and a budget of credits. The Applications Manager creates this account when the user submits its request. The account is charged periodically by the VM Scheduler based on the resource consumption of the application's virtual clusters. When the application execution ends, the Applications Manager requests the Virtual Currency Manager to destroy the associated account.

The responsibility of an application controller is to manage the user's application. An application can have one or more components, requiring different software configurations. Thus, for each application component, the application controller deploys a virtual cluster, starts the application component in it and scales it according to user defined policies. Policies can be defined to target user SLOs while optimizing the user budget. As these application components might have different performance metrics, a different virtual cluster monitor running user specific monitoring policies can be started in each virtual cluster. This virtual cluster monitor sends periodically application performance metrics to the application controller. The user can interact with her application controller during its execution to modify any adaptation policy parameters or to retrieve statistics regarding the application execution.

Algorithm 6.1 Application controller main loop.

```
 1: vmgroups = deploy remaining virtual clusters
 2: if application needs resume then
 3:     ResumeApplication()
 4: else
 5:     StartApplication()
 6: for vmgroup in vmgroups do
 7:     if vmgroup['poll'] == 'yes' then
 8:         StartMonitor(vmgroup)
 9: while not stopped do
10:     (vclusterId, state, message) = get message from monitor
11:     if application is running then
12:         ProcessMonitorMessage(vclusterId, message)
13:     if NeedsReconfiguration() then
14:         ReconfigureApplication()
15:     if NeedsSuspend() then
16:         SuspendApplication()
17:         send suspend message to Applications Manager
18:     wait for a poll period
19: transfer files from virtual clusters to user's file system
20: send shutdown message to Applications Manager
```

The main controller loop is illustrated in Algorithm 6.1. Users can automate the deployment and management of their applications by writing custom application controllers. These application controllers might include complex policies like prediction, profiling and learning modules to determine the allocation that the application needs for its execution

Algorithm 6.2 Callback executed for a user-defined objective.

```
ProcessMonitorMessage(vclusterId, message)
t_exec = parse message
if t_exec < T then
  shutdown()
```

and use this information in changing the number of provisioned VMs. The main callbacks used for tuning a controller are underlined in Algorithm 6.1.

As an example, let's take a user who wants to execute a fluid dynamics simulation (e.g., the flow of water in the reactor of a nuclear plant) and stop it as soon as it reaches a stable state (i.e., the execution time per simulation time step drops considerably). This user can write a simple controller by implementing the *ProcessMonitorMessage* callback illustrated by Figure 6.2. This callback simply compares the performance value read by the monitor, t_{exec} with a threshold, T. The value of this threshold can be specified in the virtual platform template. If the performance value drops below this threshold, then the application manager shuts down the application.

6.1.1.2 Virtual Currency Manager

The Virtual Currency Manager manages the virtual currency and distributes it among users and their applications. Virtual currency is desirable in closed environments, i.e., private infrastructures shared by the users belonging to the same organization, as it provides incentives to users to request only as many resources as needed. Price inflation is bounded as there is no external currency introduced. The Virtual Currency Manager distributes the currency among users by retrieving the list of user accounts from the IaaS Cloud Manager and allocating an amount of credits to each account proportional to a priority defined by the infrastructure's administrator. In managing credits assigned to users, two decisions need to be made:

Deciding the system's total credit amount. Deciding the total credit amount in the system is important. If the credit amount is too big the resource prices become too inflated and the system will not be able to ensure any SLO guarantees at all. If the credit amount is small and new users arrive, current users might be funded at a rate too small to allow them to run highly urgent applications. Setting this amount can be done by the infrastructure administrator based on the number of users in the system and the price history. For example, if the administrator sees that the price was too high for long periods of time, or there are too few users in the system, it can reduce the total credit amount.

Deciding a credit circulation model. As we target a closed environment, our system applies a rechargeable currency model [90]: the payments users make for resources are redistributed after a fixed time interval to their accounts. The time interval value establishes the user behavior: if the time interval is small, we obtain a weight-based system and users don't have any incentives to take judicious resource allocation decisions. If the time interval is too long, we have to deal with user starvation. The administrator can decide the time interval based on past observations. We use this model to regulate the users' resource demand over long periods of time.

The Virtual Currency Manager also manages the credits assigned by users to their running applications. The amount of credits assigned to the application limits the priority

of the resource demands the application makes. Users have two options: (i) set their application accounts to be renewed automatically; (ii) or renew them manually. Automatic budget renewal policies are used to avoid cases in which the application's budget gets depleted in the middle of its execution: if specified by the user, the Virtual Currency Manager performs at each predefined interval a transfer from the user account to the application account.

6.1.1.3 VM Scheduler

The VM Scheduler is in charge of allocating VMs on the infrastructure and limiting the resource consumption in terms of memory and CPU for every VM according to infrastructure load and VM bid. The used algorithms are described in Chapter 4. To regulate the resource consumption, the scheduler applies a periodical auction. To ease application management, the scheduler exposes an API for virtual cluster management.

The scheduler is designed to be generic enough to support different allocation policies. To meet this goal, the allocation policies are decoupled from the VM management code. New policies can be integrated by implementing the following callbacks:

- *schedule(vmlist, newvms)* : This method receives a list of currently running VMs and a list of VMs submitted during the previous scheduling period. The goal of this method is to produce the CPU and memory allocations for each running and submitted VMs, together with a migration and deployment plan.

- *chargeByVmgroup()* : This method computes the payment for each virtual cluster. This payment is sent to the Virtual Currency Manager.

- *setPrice()* : This method computes the price for each resource type, i.e., CPU and memory.

- *computePayment(allocation)* This method provides the caller with a hint on how much she should pay for the specified allocation, in terms on number of VMs and allocation for each VM. In the current resource model, the payment is computed by considering the current resource price.

To ease the development of adaptation policies, the VM Scheduler also provides additional information that helps the application controllers to take more informed decisions: (i) the current resource price; (ii) estimations regarding the cost of the virtual cluster if a certain allocation (i.e., number of VMs and resource amount required by each VM) is specified; (iii) and estimation of the allocation (i.e., number of VMs) obtained at the current bid.

The VM Scheduler interacts with the IaaS Cloud Manager to boot/shutdown VMs or to perform operations on them like suspend/resume or live migration (i.e., if load balancing is needed). When suspending or resuming the virtual clusters we don't use any synchronization mechanisms [12]. We acknowledge that such mechanisms could be integrated. For now it is up to the user to implement additional application-specific mechanisms in its own controller.

6.1.1.4 IaaS Cloud Manager

VMs are allocated on the infrastructure by using an IaaS Cloud Manager. The IaaS Cloud Manager supports the following operations:

- user management and authentication;
- VM operations: boot, migrations, suspend, resume and shutdown;
- VM image management;
- VM network management;
- VM monitoring.

To allow infrastructure administrators to easily replace the IaaS Cloud Manager, a generic API encapsulates all the operations done by the IaaS Cloud Manager. Each component of our system interacts with the IaaS Cloud Manager through this API.

6.1.2 Merkat-User Interaction

To give users the possibility to design their own application management and resource demand adaptation policies, Merkat provides an API for each of its components. Besides this API, Merkat also provides CLI tools to access each of its components. To run applications on the cloud, users need to use the authentication methods provided by the Cloud Manager.

6.1.3 Virtual Platform Template

To deploy applications, users need to submit a request containing a virtual platform template to the Applications Manager. This template has two parts: (i) a part describing the SLO policies and more generic information regarding the application execution; (ii) and a part describing the virtual clusters (or components) of the application. This example defines a virtual platform used to run an MPI application, Zephyr [195].

6.1.3.1 Generic Information

Figure 6.2 illustrates the generic content of a virtual platform template. Users need to specify the following information:

- budgeting information (Lines 5-6);
- the type of application controller to load (Line 3); the placement of the application controller is decided by the user by specifying the virtual cluster in which it should be deployed (Line 7).
- the virtual cluster configurations (Lines 10-12);
- the application resource demand policy (Lines 13-20).

In this example, the application controller uses a policy to scale the application resource demand to meet a given deadline (10800 seconds from the submission time in our case). The user assigns to its application an initial amount of *80000* credits that is renewed with *80000* credits at each scheduling period. If the application hasn't started or already finished, the application controller keeps the VMs idle a maximum time of 600 seconds. During the application execution, the application controller gets the estimated execution time of the application from the virtual cluster monitor at every 60 seconds. To support multiple SLOs users can also specify in the configuration file any parameters needed in the development of their resource demand adaptation policies. As additional parameters, the user specifies a profile file used by the application controller to select the number of processors with which the application is started and the application reconfiguration time (i.e., the time required to restart the application with a different number of processors) used to increase the number of processors if the deadline is not met.

```
1:  <platform>
2:  <name>Zephyr</name>
3:  <controller>MPIMal</controller>
4:  <keepidle>600</keepidle>
5:  <budget>80000</budget>
6:  <renew>80000</renew>
7:  <group name='zephyr-0' poll='yes' controller='yes' process='mpirun' depends='zephyr-1'>
8:          ......
9:  </group>
10: <group name='zephyr-1' poll='no' controller='no'>
11:         ......
12: </group>
13: <policy>
14:         <type>deadline</type>
15:         <reference>10800</reference>
16:         <params>
17:         <treconf>180</treconf>
18:         <profile>/profiles/profile_zephyr</profile>
19:         </params>
20: </policy>
21: <poll>60</poll>
22: </platform>
```

Figure 6.2: Virtual platform template.

6.1.3.2 Virtual Clusters

The virtual clusters composing the virtual platform might have dependencies between them that establish the order in which they need to be started. For example, some resource management frameworks (e.g. Condor [105], Torque [160]) require starting first the master node and then the worker nodes by communicating to them the master hostname and IP address. As another example, in the case of an MPI application, the MPI master process needs to know the IP addresses of the VMs to start application processes in them. For simplicity we support only a one-level dependency: a "slave" virtual cluster can depend on only one "master" virtual cluster. However, there are no constraints regarding the number of virtual clusters.

To describe a virtual cluster, a user needs to detail its configuration in the virtual platform template file. Figure 6.3 illustrates the configuration of a virtual cluster for a MPI application. A virtual cluster configuration includes the following information:

- commands to manage application components, i.e., start, suspend, resume, stop (Lines 12-25); in our example the user uses one script for all the commands, but with different parameters. These commands could use the internal parameters of the virtual clusters, that would be created or modified at runtime: VM IPs, names or an application controller id;
- files to be copied in/from the VMs before/after the application is started/stopped (Lines 6-11);
- VM bids per resource unit (Line 2); if the user has defined a policy for the virtual platform, these bids will be replaced by the ones computed with the policy;
- VM template in a format interpreted by the IaaS Cloud Manager (Line 27);
- additional log files (Line 5); these files will be periodically read by the virtual cluster monitor;
- any virtual cluster on which the current virtual cluster depends on (Line 1);

```
1: <group name='zephyr-0' poll='yes' controller='yes' process='mpirun' depends='zephyr-1'>
2:   <bid>100</bid>
3:   <nvms>1</nvms>
4:   <monitor>MPIMal</monitor>
5:   <log>/PIPE</log>
6:   <boot>
7:     <command>chmod a+x init.sh</command>
8:   </boot>
9:   <boot>
10:     <command>./init.sh 1</command>
11:    <param name = 'procs'/>
12:    <param name = 'appid'/>
13:  </boot>
14:  <resume>
15:     <command>./init.sh 3</command>
16:     <param name = 'procs'/>
17:     <param name = 'appid'/>
18:  </resume>
19:  <suspend>
20:     <command>./init.sh 2</command>
21:     <param name = 'appid'/>
22:  </suspend>
23:  <shutdown>pkill -9 mpirun</shutdown>
24:  <template>APP.one</template>
25:  <input>
26:    <file>init.sh</file>
27:  </input>
28: </group>
```

Figure 6.3: Virtual cluster configuration example.

– virtual cluster name, that will be assigned as a hostname to its VMs, together with the number of the VM in the virtual cluster (Line 1).

– if an application controller or virtual cluster monitor should be deployed in the virtual cluster or not, and the name of the process to be monitored (Line 1).

In our example, the user specified a script, "init.sh" to be used for all operations regarding the application. This script takes as input for both start and restart the number of processors on which the application starts, and the application controller identifier, used for naming the directory in which the application output will be stored. A virtual cluster monitor reads the application progress information from a log file, called "PIPE". The same virtual cluster monitor detects the end of the application execution by monitoring the MPI process, "mpirun". When this process finishes, the application controller assumes that the application ended (successfully or not) its execution and tells the Applications Manager to shut down the virtual platform.

6.1.4 Application Monitoring

As previously stated, each virtual cluster has its own monitor, which is a daemon process started in the first VM of the virtual cluster. This monitor keeps the states for its application component, which can be: (i) *RUNNING*: the application process is running; (ii) *SUSPENDED*: the application process was suspended; (iii) or *DONE*: the application process finished its execution. This state is transmitted to the application controller together with the performance metrics. Based on this state, the application controller can issue commands regarding virtual platform shutdown or suspend to the Applications Manager. For example, the user might want to gracefully shut down, or suspend her application. In both cases, the application controller forwards this command to the application and waits for the application to save any useful data and stop its execution. When the application

controller receives a *DONE* state from the monitor, it informs the Applications Manager to shut down the VMs.

6.1.5 The Application Controller's Life-cycle

During its lifecycle, an application controller passes through different states and performs different actions. Figure 6.4 describes the states of the application controller. Figure 6.5 illustrates how users and our system services interact in order to manage an application during its life-time. In this figure, a user interacts with our system through the use of a client program, i.e., a CLI tool.

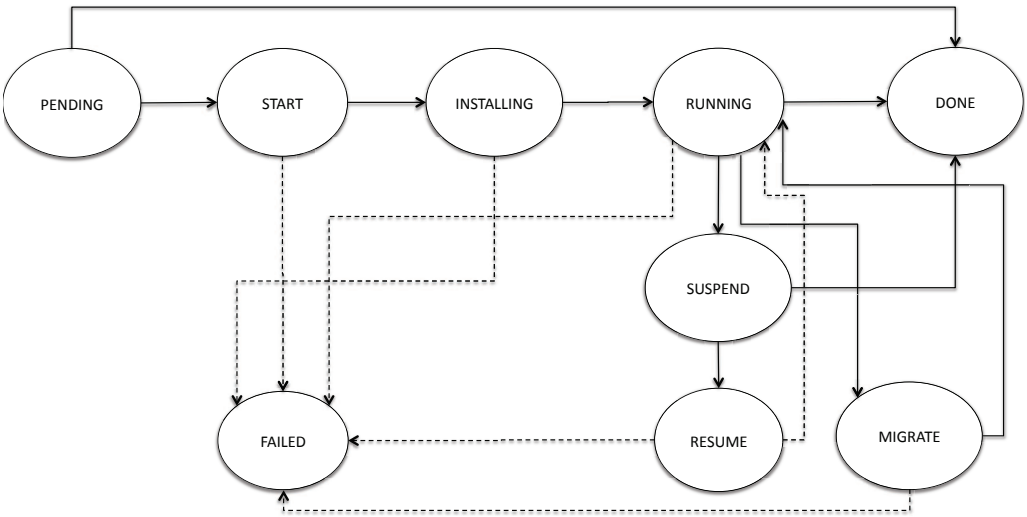


Figure 6.4: Application controller states.

6.1.5.1 Application Bootstrap

To start their applications, users submit a virtual platform template as described in Section 6.1.3.1, to the Applications Manager. The Applications Manager can deploy virtual platforms on different clusters managed by our system. An Applications Manager can be aware of the different clusters on which Merkat was deployed by specifying in its configuration the address of what VM Schedulers to use. If there is more than one VM Scheduler, than the Applications Manager will perform load balancing between them by submitting virtual platforms to the cluster with the smallest price, as shown in Figure 6.6.

When the user submits a request, the Applications Manager creates an application controller, registers it with the virtual currency manager and puts it in a *PENDING* state. If the user specified any initial deployment conditions, the Applications Manager checks them. If these conditions are not met (e.g., the controller might check if the price is too high for the user given budget), the deployment of the application controller is postponed. If no postponing is possible (e.g., the application needs to execute before a user given deadline, and the deadline will be missed due to the high price), the Applications Manager deletes any application controller information and puts it in a *DONE* state.

Based on the virtual platform template, the Applications Manager requests one initial VM from the VM Scheduler and starts an application controller in it. During the VM initial deployment the application controller is marked in a *START* state. After the VM is started the Applications Manager transfers the application controller files in it, starts

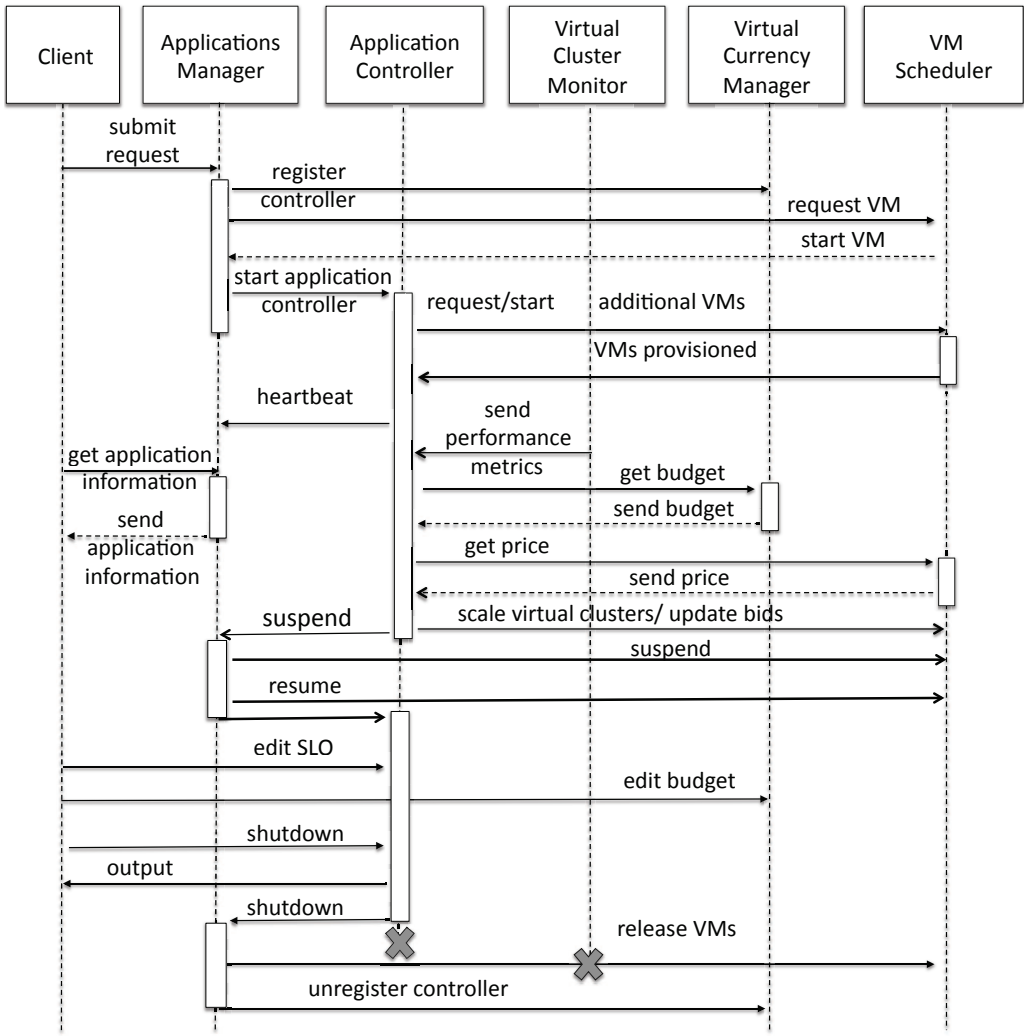


Figure 6.5: Typical application execution flow.

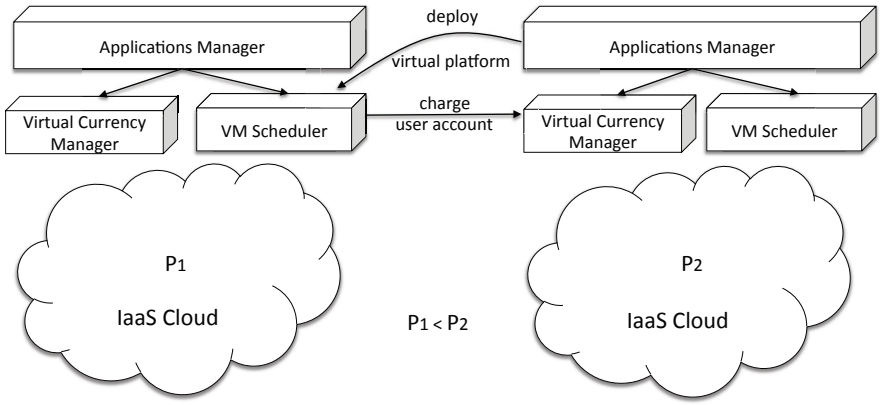


Figure 6.6: Interaction between components from different clusters.

the application controller and puts it in the *INSTALLING* state. If any of these steps fail to execute, the application controller is marked as *FAILED*. The application controller then starts the application components in all the provisioned VMs and communicates the Applications Manager that it is in *RUNNING* state.

6.1.5.2 Monitoring and Elastic Scaling

During the application runtime the application controller periodically receives performance values (e.g., number of tasks, progress) from the monitors running in the controller's provisioned virtual clusters and retrieves information about current price and budget. Based on this information, it scales the application resource demand by expanding or shrinking its virtual clusters or it modifies the virtual cluster bid to ensure the virtual cluster receives the right allocation. During its lifetime, the application controller periodically sends heartbeat messages to the Applications Manager. When the Applications Manager detects that a controller hasn't sent a heartbeat for given a period of time, it marks it as *FAILED*. It is up to the user to implement the correct policies to manage failed VMs during the application runtime. For example, the user can extend the application controller to test periodically all running VMs and, if they are timeouted, to mark them as failed. Failed VMs can be replaced and application components can be started in the new VMs.

6.1.5.3 Suspend and Resume

To stop the application execution due to resource scheduling constraints, application controllers might use suspend/resume mechanisms. In our case, as resources have a dynamic price, the application might be suspended if its execution cost becomes too high for the user and resumed afterwards, when this cost becomes lower.

The suspend mechanism is implemented as follows. The application controller suspends the application and then announces the Applications Manager to stop the virtual clusters (we assume that a shared file system exists so that this state can be saved on it). When the Applications Manager receives this decision it sets the controller state to *SUSPEND* and destroys its virtual clusters.

When the user conditions to resume her application are met, the Applications Manager re-starts the application controller. This process is identical to the application controller bootstrap. When this operation is performed the application controller is marked in a

RESUME state. Nevertheless, if there is no possibility for the resume conditions to be met in the future, the Applications Manager deletes any application controller information and puts the application controller in a *DONE* state.

6.1.5.4 Migrate

Virtual platform migration can be done through "cold" migration: the virtual platform is stopped on the source cluster and re-started on the destination cluster. Virtual platform migration can be performed by the user or by her application controller when some predefined condition is broken, i.e., the allocation the application gets on a cluster is too small at the current price. When the virtual platform is migrating, the application controller is marked in a *MIGRATE* state. If the process fails, the application controller is marked in a *FAILED* state.

6.1.5.5 Shutdown

There are several cases when the application controller shuts down: (i) when the application finishes its execution; (ii) when the user requests a shutdown; (iii) when there is not enough budget to keep the application running at the current price and the user doesn't want to suspend the application. In the first case the controller asks the Applications Manager to release all provisioned VMs. In the other cases, the controller first shuts down the application components and then it asks the applications manager to release the virtual machines.

6.2 Implementation

We have implemented Merkat in Python. Merkat depends on the Twisted [170], paramiko [133] and ZeroMq [196] libraries. The Twisted framework is used to develop the XML-RPC services. Paramiko is used for the VM connection: the application management system needs to test and configure the VMs in which the application runs and it does so through SSH. ZeroMq is used for the internal communication between various components of Merkat (e.g., the Applications Manager and application controllers, or the application controllers and virtual cluster monitors). Merkat's services store their state in a database storage for which we have used a MySQL server. The VM connections are done in parallel, by using a thread pool with a size defined in the configuration file. As a IaaS Cloud Manager we use OpenNebula [158]. Merkat's services communicate with OpenNebula through its XML-RPC API. We also shut down OpenNebula's default scheduler, and replaced it with our VM Scheduler. We summarize next some implementation details for the Merkat components and describe how Merkat can be configured.

6.2.1 Component Implementation

The Merkat's services are materialized through a set of daemons. Let us describe them next:

Application Management System The application management system is materialized by the following daemons: (i) Merkat_frontend, which runs the Applications Manager service; (ii) Merkat_backend, which runs the application controller; (iii) Merkat_monitor, which runs the virtual cluster monitor. Each application controller

and virtual cluster monitor has a ZeroMq channel. The application controller listens on this channel for commands and forwards its own commands and heartbeat messages to the Applications Manager. The virtual cluster monitor uses its channel to send performance metrics and the application state to the application controller. Users have access to functionalities regarding VM provisioning and configuration and they only have to write their scaling policies.

VM Scheduler The VM Scheduler is implemented in two daemons called *Merkat_scheduler* and *Merkat_local_scheduler*. *Merkat_scheduler* runs on the cloud frontend and performs the VM allocation. *Merkat_local_scheduler* runs on each node and performs the VM resource allocation. The VM Scheduler retrieves periodically information from the database regarding pending VMs, VMs to be deleted and updates regarding their bids. Based on this information it computes the VM allocations, migration and deployment plans. Each *Merkat_local_scheduler* daemon has a ZeroMq channel on which it receives allocation information from the VM scheduler, and optionally, it sends back utilization information. Whenever it receives new allocations, the daemon enforces them through the *cgroups* interface. To improve the application performance, the VM VCPU cores are pined to the physical cores. KVM [95] is used as hypervisor on each node together with *cgroups* for resource capping.

Virtual Currency Manager The Virtual Currency Manager is implemented in a daemon called *Merkat_vbank*. The IaaS Cloud administrator uses this service to set user "weights", which are used afterwards by the currency distribution policy to give credits to users.

6.2.2 Information Management

Each system component stores information regarding its state in its data storage. We implemented an API to store this information in a storage backend and a large variety of storage backends (e..g, MySQL, MongoDB) can be integrated. The stored information is the following:

- the Applications Manager stores information regarding submitted requests (running and suspended) and their current virtual clusters and state;
- the VM Scheduler stores information regarding virtual cluster identifiers, their associated application controller, their VM ids and their bids;
- the Virtual Currency Manager stores information regarding application controller and user accounts.

6.2.3 Merkat Configuration

The service configuration is done through a Python file. We mention here some important parameters: (i) addresses (hostname and port) for all the Merkat and additional services, i.e., the MySQL server and IaaS cloud manager, and ZeroMq ports; (ii) additional configuration parameters, infrastructure budget, type of VM allocation policy, thresholds; (iii) SSH configuration parameters: each VM connection is retried several times after waiting

for a predefined timeout interval; (iv) path to a shared file system, if any [2]; (v) application controller heartbeat interval.

6.3 Supported Applications

Merkat currently supports two types of applications: (i) static MPI applications; (ii) and malleable applications (e.,g., task processing frameworks). To illustrate the use of Merkat for each of these applications, we implemented a controller to scale the resource demand according to an user objective.

6.3.1 Static MPI applications

To illustrate Merkat's support for static applications, we designed an application controller for MPI applications. We tested this controller with Zephyr [195], an open source software used for fluid dynamics simulations. Zephyr is a typical application used at EDF R&D.

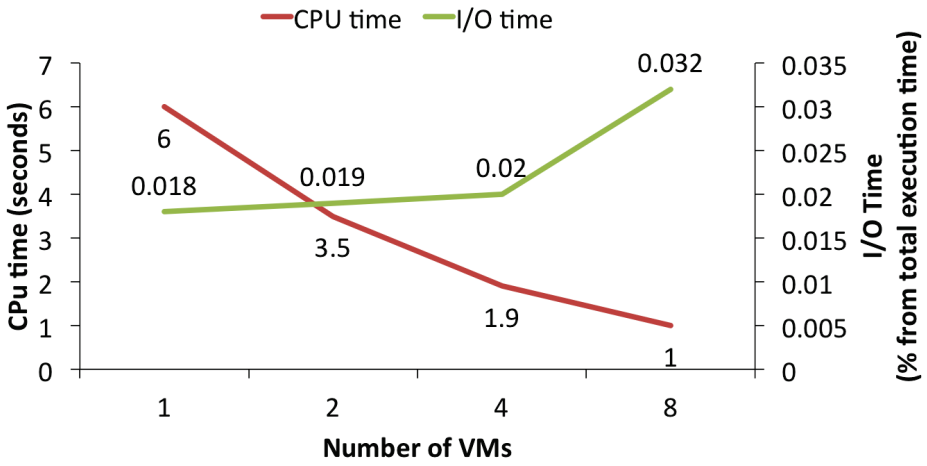


Figure 6.7: Zephyr profile.

6.3.1.1 Zephyr

The implementation of Zephyr is done by using MPI. Zephyr receives as input a configuration file and it simulates a volume filled by fluid for a specified simulated time, $t_{physical}$. The computation is done iteratively: at each iteration a set of equations is solved, e.g., Navier–Stokes equations, using a discretization step τ. The simulated time and the discretization step can be specified as input in Zephyr's configuration file. Zephyr provides the CPU time for each computed simulation step together with the total elapsed time, i.e., the time from the execution start, and the total amount of time spent in I/O.

As the execution time for each iteration remains quite the same, the total execution time of the application can be computed as a product of the total number of iterations and the average iteration execution time. The total number of iterations performed by the application depends on the discretization step, τ, used in solving the equations:

$$N_{iterations} = \frac{t_{physical}}{\tau} \tag{6.1}$$

[2]This file system is mounted in each started VM and any application files are not transferred to the VM, but instead are copied from the file system

We use the simulated time and the discretization step to model the execution time of Zephyr. This gives us the opportunity to run a workload of applications with different execution times, starting from the same application and the same input data. We model the Zephyr's execution time based on an execution profile of the application. Figure 6.7 shows this profile. To obtain this profile we ran Zephyr with a different number of VMs, i.e., one Zephyr process per VM, on the *genepi* cluster from Grenoble and we measured the computation time for each iteration together with the percentage of time spent in I/O. We performed the experiments 10 times and we computed an average over the obtained values.

To make Zephyr run for a given execution time, t_{exec} on a specified number of processors, by knowing the discretization step we compute the simulated time and feed it as an input to Zephyr. The simulated time is computed as follows:

$$t_{physical} = \tau \cdot \frac{t_{exec} - t_{io}}{t_{timestep}} \tag{6.2}$$

where $t_{timestep}$ is the iteration execution time and t_{io} is the I/O time for the specified number of processors. First, we need to compute the total number of iterations that the application needs to perform. For this, we substract from the given execution time, the time spent in I/O, and divide the result with the execution time per time step. Now, to obtain the simulated time, it is enough to multiply the total number of iterations with the discretization step.

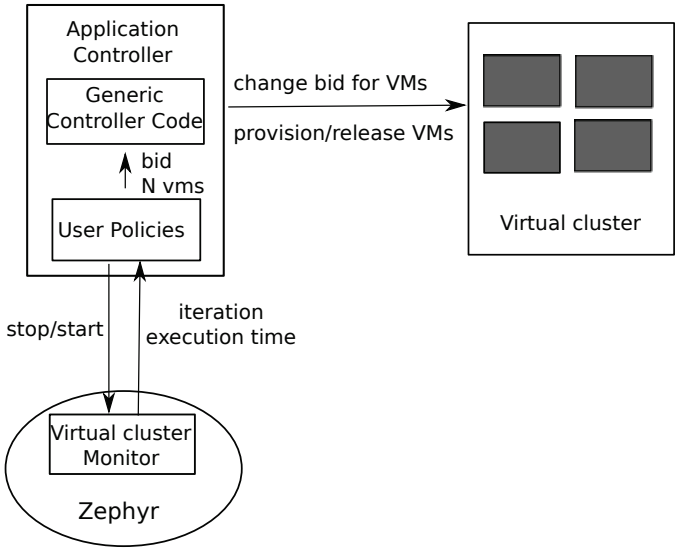


Figure 6.8: Merkat controller - Zephyr interaction.

6.3.1.2 Application Controller

Figure 6.8 describes the interaction between an application controller and a Zephyr application. the application controller is generic enough to support other MPI applications in addition to Zephyr. In fact we have also written plugins for Code_Saturne [48], another fluid dynamics simulator, and Gromacs [82], a molecular dynamics simulator.

The application controller can apply three resource demand adaptation policies:

best-effort To run the application in best-effort mode we implemented a "dummy" policy.

the application controller starts the application when the price is low enough and then doesn't react at all to changes in the application's performance or infrastructure price.

SLO-driven vertical scaling To run the application in SLO-driven mode, the application controller uses uses Algorithm 4.6, described in Section 4.4.5. This ensures that in contention periods the application consumes just as much resource to meet its SLO. This fact has two advantages: (i) first it keeps a low price leaving resources for other applications; (ii) it optimizes the budget spent by the user for its application.

This algorithm tries to keep the application performance between two thresholds: v_{low} and v_{high}. In our experiments we set v_{low} at $0.75 \cdot v_{ref}$ and v_{high} at $0.95 \cdot v_{ref}$, where v_{ref} is the application performance reference. We have also experimented with different values for these thresholds. However, we found that setting a small v_{low} leads to an aggressive bid increase, as the "gap" between the reference performance value and the actual one is also large. Also, when setting a high v_{high} might lead to more deadline misses, as the controller doesn't have time to adapt.

The performance metrics are fed to the application controller by a virtual cluster monitor which runs in the same VM as the MPI master process. Zephyr dumps periodically information about its current iteration step, past iteration execution time and total number of steps in a log file. The virtual cluster monitor uses a plugin to read this log file.

SLO-driven hybrid scaling We also wanted to be able to reconfigure the applications with more processors, dynamically during its execution, when its deadline cannot be met. In this case, the application controller reads a *profile* file, specified in the virtual platform template, containing the number of processors and the application execution time, and selects the smallest number of processors with which the application can meet its deadline. The application runs initially on this number of processors. Then, when the deadline is not met, the application controller adds more VMs and reconfigures the application to run with more processors.

However, in the case of Zephyr is not possible to change the number of processors during its runtime, as the processed data needs to be redistributed between the new processors too. Thus, when the number of processors needs to be changed, two steps are performed. First, the application controller informs the application to save its data in a file and to stop its execution, i.e., it suspends the application. Then, when the application stopped executing, the application controller provisions/releases VMs, configures them, i.e., it adds their IPs in the hostname file needed by MPI, and restarts the application.

In all cases the application controller uses a monitoring period of *60* seconds. This period needs to be larger than the VM Scheduler scheduling period due to two reasons: (i) to give time to the market scheduler to change the application allocation; (ii) to give time to the application to reflect this change.

6.3.2 Malleable Applications

To illustrate the support for malleable applications, we implemented controllers that scale two frameworks: Condor [168] and Torque [160]. We implemented a scaling policy for each framework. Each framework has its own scheduler that manages the framework's resources. Users can submit their applications to these frameworks instead of Merkat. In this case,

Merkat is capable of allocating to frameworks shares of the infrastructure proportional to the framework-assigned budget and current workload.

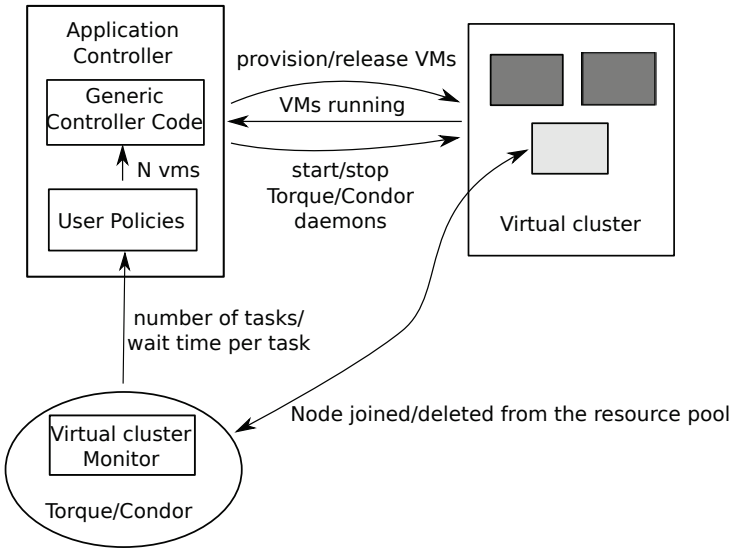


Figure 6.9: Merkat controller - framework interaction.

Figure 6.9 illustrates the interaction of a Merkat controller with the framework. A virtual cluster monitors uses the framework provided commands to get the number of running and queued jobs, i.e., for Torque we use the *qstat* command while for Condor we use *condor_q*. For the Torque framework we also compute the wait time for each queued task. The application controller uses a modified version of Algorithm 4.7 from Section 4.4.6 to provision VMs as long as the total wait time of Torque jobs or the number of queued jobs in Condor is above a threshold.

We use a modified version of Algorithm 4.7 as provisioning one VM at each time interval would have made the scaling process too slow. For example, if the application controller would have used a monitoring period of 60 seconds, it would have required 160 minutes to reach 160 VMs. Thus, the application controller provisions a random number of VMs between a minimum and a maximum limit, set by the user according to the infrastructure size. If the infrastructure is large it makes more sense to provision more VMs at once, otherwise, a smaller number would suffice.

In both scaling policies, the application controller copes with fluctuations in price in two ways. First, it avoids requesting a large number of VMs per time period, as provisioning them is wasteful if the price increases in the next period. Second it avoids starting new VMs if some VMs were released due to a price increase in the previous period.

Based on this newly computed VM number, the application controller provisions new VMs, releases them or changes the bid for them. If VMs are provisioned, the application controller also configures them using user-defined scripts, i.e., for the Torque framework it starts a *pbs_mom* daemon which connects to the Torque master node, while for the Condor framework it starts a *condor* daemon which connects to the Condor master node. After VMs are configured, they join the framework's resource pool. If VMs are to be released, the application controller runs a shut-down script on them, stopping the framework daemons.

6.4 Experimental Setup

All the experiments presented in this chapter were run on the Grenoble site of the Grid'5000 testbed [23]. We used 20 nodes from the *genepi* cluster to set up a cloud. Each cluster node has 2 Intel Xeon E5420 QC processors (4 cores) and 8GB of memory. The VM Scheduler assigns for each node an available capacity of $< c = 800units, m = 7000units >$ and it distributes it among the VMs running on the node.

All nodes are connected through a Gigabit Ethernet link. Unless otherwise specified, all VMs have a maximum capacity of 1 core and 900MB of RAM, and thus, 100 CPU units and 900 memory units, each. Each VM is configured with a Debian Squeeze 6.0.1 OS and KVM as a hypervisor on each node. One node is dedicated to the OpenNebula frontend and the Merkat services. A second node is configured to act as an NFS server for storing the VM images. To speed-up the VM deployment we use copy-on-write VM images.

In all experiments, the VM scheduler uses a scheduling period of 40 seconds. We choose this period to be large enough for a small infrastructure in order to allow VM operations like migration and boot to finish before the start of the next scheduling period. In the same time, we choose the period to be small enough to give time to controllers to adapt to changes.

The reserve price is set at 1 credit for CPU unit and memory unit.

6.5 Experiments

We evaluate Merkat in two ways. First we analyze the behavior of the designed controllers. We illustrate the controller behavior through five examples: three examples for the static MPI application controller and two examples for the malleable application controller. For the static MPI applications we show how:

- Merkat can provide vertical scaling to applications and prioritize urgent applications;
- Merkat can react to changes in the application SLO according to user-defined policies.

For the malleable applications we show how:

- Merkat can provide horizontal scaling to applications.
- Merkat can adapt the application resource demand by considering budget changes. In this way, budgets act like priorities which can be tuned dynamically during the application execution.

Finally, we measure the capacity of Merkat to manage multiple SLO-driven applications.

6.5.1 Highly Urgent Application Execution

To show how Merkat can tune vertically the application allocation, we run a straightforward scenario. We start 13 best effort Zephyr applications on a single physical node, and, later during the experiment, we start an urgent Zephyr application. We say the application is urgent as it needs to finish fast its execution and the user assigns a high budget to it. The best-effort applications have 1 task each and run in a VM with 1 core and 900 MB RAM. The urgent application has 4 tasks and runs in a VM with 4 cores and 900 MB RAM. The best-effort applications are started at the beginning of the experiment, while the urgent application is started after 1300 seconds. Each best-effort application submits a bid of 100 CPU credits and 900 memory credits for its VM, expecting to receive the

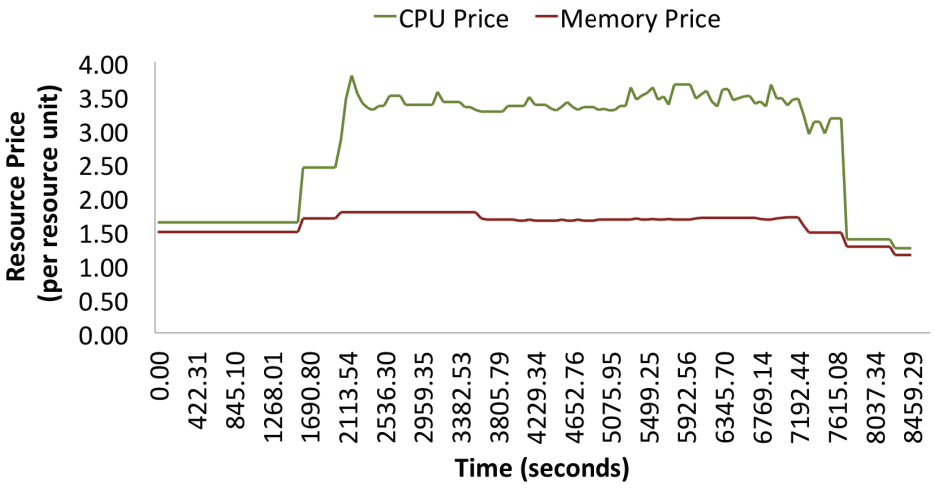


Figure 6.10: Price variation when a highly urgent application is submitted.

maximum resource allocation at the reserve price, while the urgent application, although it starts with a budget of 60000 credits, it adapts its bid to the resource availability.

Figure 6.10 shows the variation in price for each allocated resource. The increase in price marks the moment when the urgent application runs on the node while the decrease in price marks the moment when the urgent application finishes its execution. The variation in price is due to the variation in performance metrics received by the application controller.

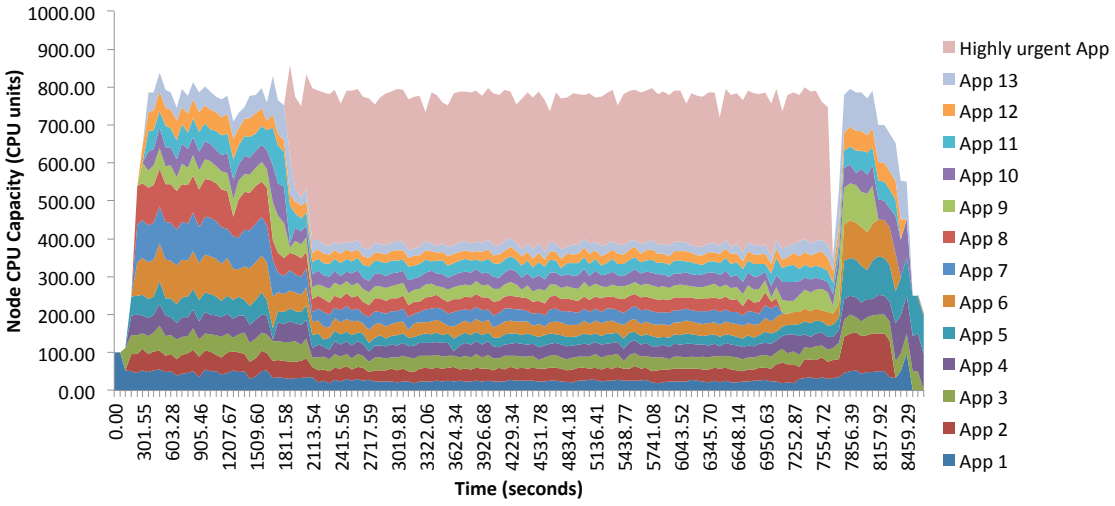


Figure 6.11: CPU utilization variation when a highly urgent application is submitted.

Figure 6.11 and Figure 6.12 show the variation in resource allocation for each application running on the infrastructure. Figure 6.11 shows the CPU utilization variation. At the begining of the experiment, although all applications submit the same bid for CPU, three of them receive twice as much as the others. This comes from the fact that virtual machine CPUs are pinned to the physical cores. Thus, three applications receive one core for themselves while the other applications receive half of a core. As the scheduling algorithms employed by the hypervisor do not support capping, all the core capacity is given to the application running on it. Finally, it can be easily noticed that the best-effort applications

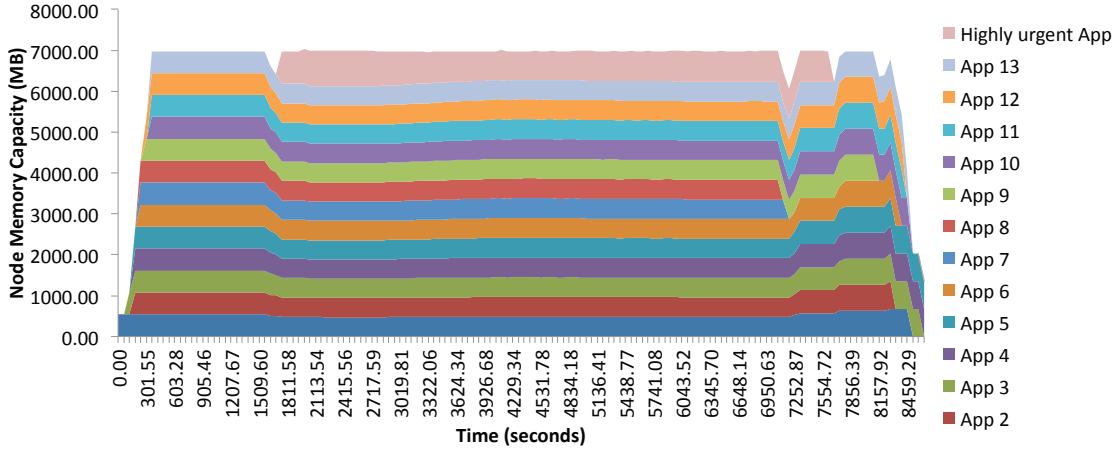


Figure 6.12: Memory allocation variation when a highly urgent application is submitted.

have their allocation reduced, to make space for the urgent application. In this case, the urgent application needs the maximum VM allocation, i.e., 400 CPU units, to finish fast. Figure 6.12 show the memory allocation variation. As the urgent application requires less memory than CPU, the memory allocations for the best effort applications are reduced less.

6.5.2 Controller Behavior for MPI Static Applications

In this section we analyze the application controller behavior for MPI applications. We first test the application controller behavior when additional applications are submitted to the infrastructure and then when the user changes its SLO during its runtime.

We imagine a scenario in which a user would want to run its application on the proportional-share market implemented by Merkat. The user would want some guarantees that the execution of his application finishes before his deadline and her given budget. He expects that these guarantees should be met as long as the infrastructure is not too overloaded.

6.5.2.1 Controller Adaptation at Infrastructure Resource Demand Fluctuations

To show how an application controller can adapt to changes in application resource allocation due to price variation, we ran a micro-benchmark using Zephyr applications: we started an SLO-driven application and four best-effort applications on a node, each in a VM with 4 cores. The SLO-driven application has a budget of 60000 credits, from which it can spend as much as it wants, while the other best-effort applications start with a bid of 100 CPU credits and 900 memory credits which remains unchanged during their execution. For this setup we used a node from our cloud. The SLO-driven application was started at the beginning of the experiment, while the other four applications were started during its execution. The last best-effort application was submitted after 20 minutes from the experiment start. The SLO-driven application has an ideal execution time of 77.5 minutes.

For clarity, we first run the SLO-driven application with a best-effort controller instead of a SLO-driven one. Figure 6.13 shows the progress the application makes over time, i.e., its iteration execution time, in two cases: (i) when it runs alone on the node; (ii) and after the other applications are submitted. The performance difference in this case is highly

noticeable: after all applications started executing, the iteration execution time increased almost 10 times. This degradation is not only due to its reduced resource allocation but also due to the other applications.

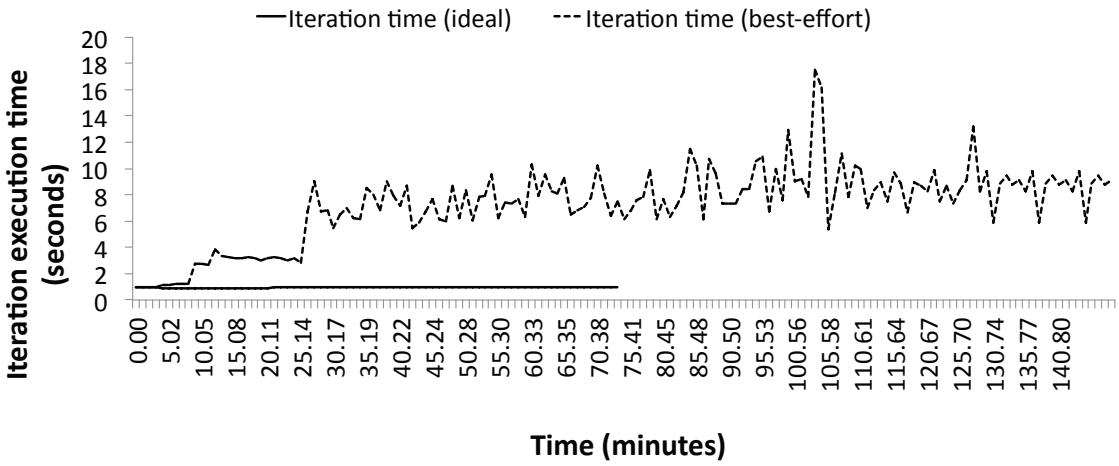


Figure 6.13: Application execution time variation with a best-effort controller.

Then we ran the application with three different deadlines: 12000 seconds, 9000 seconds and 6000 seconds. We repeated each experiment three times and computed the average of the obtained values.

Figure 6.14 shows the bid of the SLO-driven controller for CPU resource for the different application deadlines. The bid stabilizes after all the applications were submitted. The application with the smallest deadline demands a maximum allocation and thus, the submitted bid is also much higher than in the other two cases.

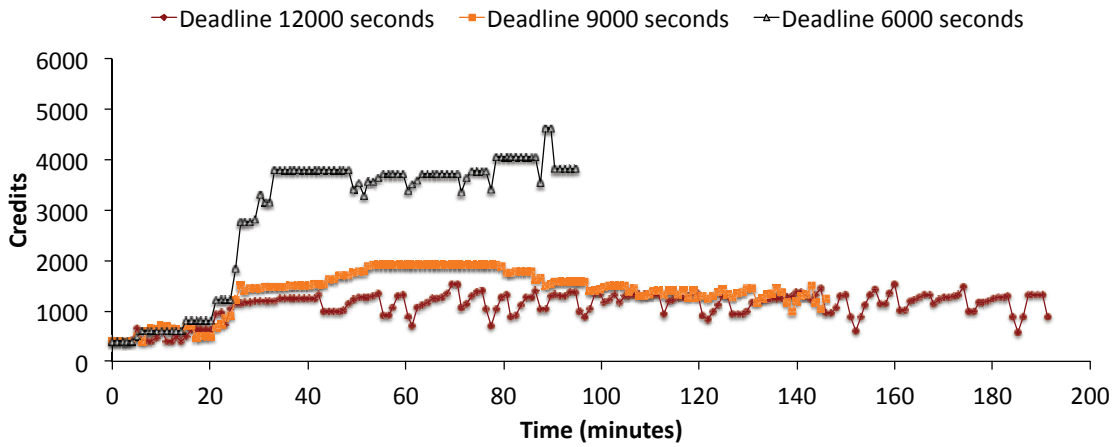


Figure 6.14: CPU bid variation for three different deadlines.

Figure 6.15 shows the variation in the VM resource share and utilization. Remember that the VM resource share is the proportional-share the VM gets according to its bid. The resource utilization is how much the application inside the VM consumes. The left axis shows the CPU resource, i.e., percentage of total CPU time, while the right axis shows the memory resource, i.e., in MBs. The best-effort application arrivals can be noticed by looking at the changes in the SLO-driven VM resource share. In this case, the memory share

reflects the best these arrivals. When the application started alone on the node the VM's share is the entire node capacity. After each application arrival, this share decreases until it equalizes the VM allocation. After all the best-effort applications started running, the SLO-driven controller keeps a reduced resource allocation, depending on the application deadline. In the case of the 6000-deadline application, the application controller keeps maximum allocation.

Figure 6.16 describes the SLO-driven MPI application iteration execution time. To see the behavior of the vertical scaling algorithm, Figure 6.16 also describes the lower and upper scaling thresholds, v_{low} and v_{high}. We notice that after all the best-effort applications started running, the application controller manages fairly well to keep the iteration execution time between these two thresholds. However, there are cases in which the iteration execution time oscillates. We think that one cause for this variation is the sharing of physical cores between more VM processes. The SLO-driven application receives more CPU than the best-effort applications, and thus less VMs get scheduled on the same physical cores as its own.

A case in which the deadline cannot be met by our system is the 6000-deadline application. In this case, obtaining the maximum allocation does not help the application to meet its deadline, due to performance interference of other applications.

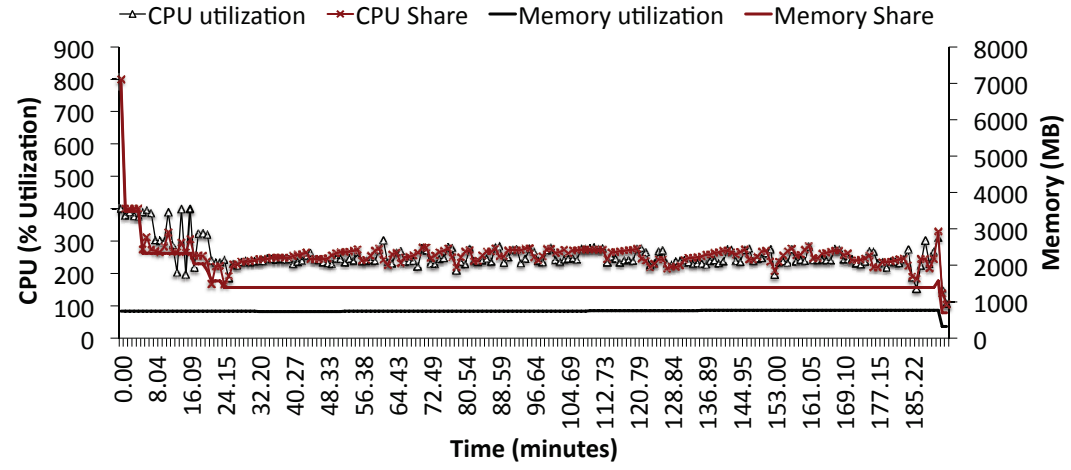
6.5.2.2 Controller Adaptation at SLO Modification

We also analyze the capacity of the application controller to react to a user imposed condition. For this, we ran the same experiment but changed the deadline of the SLO-driven application during its execution. The application is started with an initial deadline of 12000 seconds. After 64 minutes from the application start the deadline is changed to 9000 seconds.

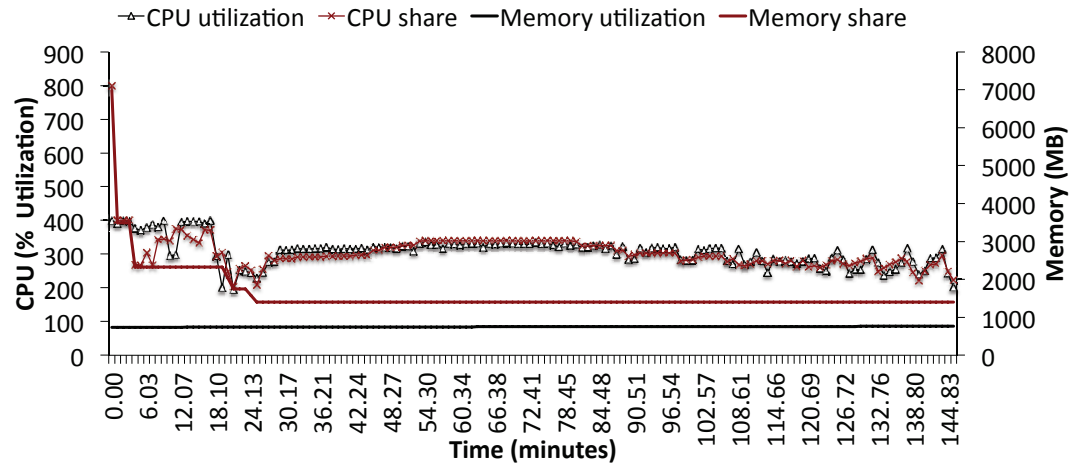
Figure 6.17 shows the application controller behavior and the variations in the estimated application execution time due to allocation and bid fluctuations. the application controller's bid adaptation and the allocation changes can be noticed in Figure 6.18. The additional submitted workload leads to a decrease in application allocation and thus an increase in its execution time. However, the application controller doesn't react aggressively as its allocation, in this case 266 CPU units, is enough to meet the application deadline. When we decrease the application's deadline, the application controller also adjusts the bid aggressively, leading to an increased resource allocation, in this case close to 400 CPU units, which is the maximum VM allocation. This increase allows the application to meet its deadline.

6.5.2.3 Discussion

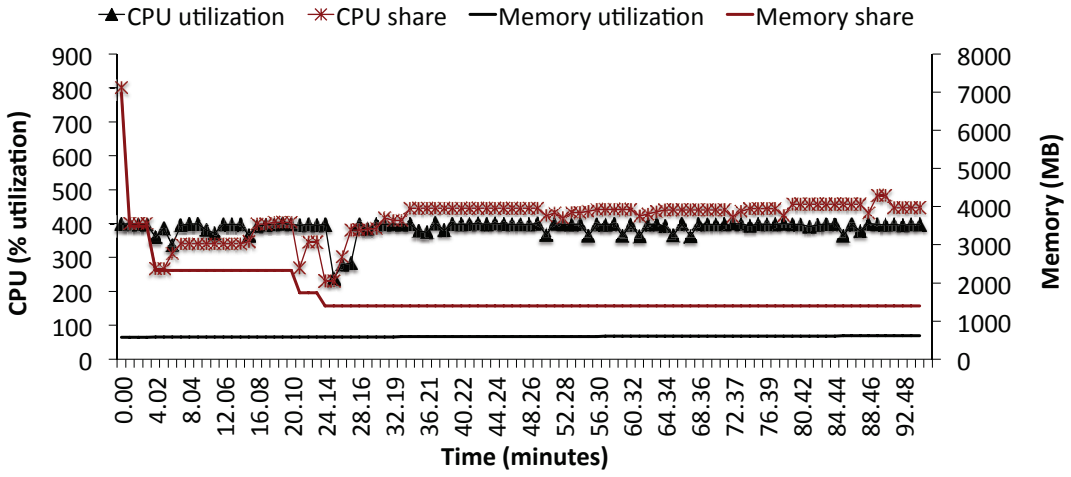
We have seen from these experiments that the application controller reacts fairly well to both changes in system workload and user requirements. Given that the system is not highly dynamic, and the application controller has time to adapt, the application can run with a smaller resource allocation and optimize its budget. While the application controller optimizes the application execution cost, it allows other applications with less budget to use resources. However, our policy does not cover all the possible cases. For example, the user's deadline might be too strict for the current price. In this case, the application controller might need to predict the future infrastructure load and send feedback to the user regarding the minimum deadline it can meet. Starting from the simple mechanisms presented here, to improve the SLO support more complex policies can be developed, based on application profiling and price prediction.



(a)

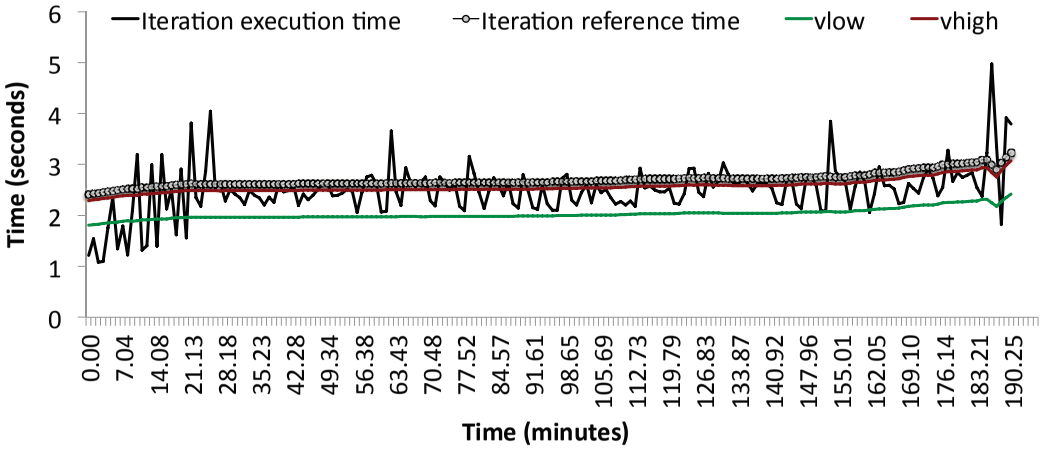


(b)

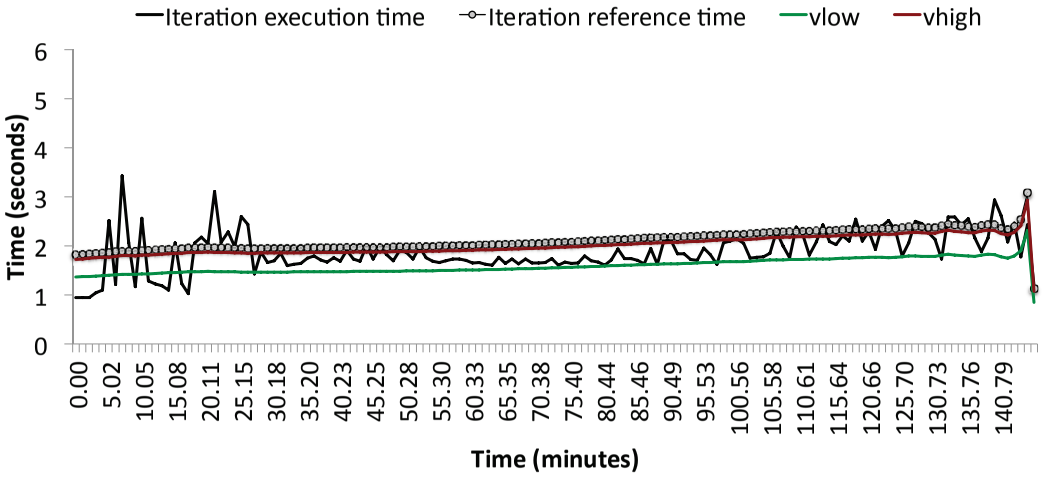


(c)

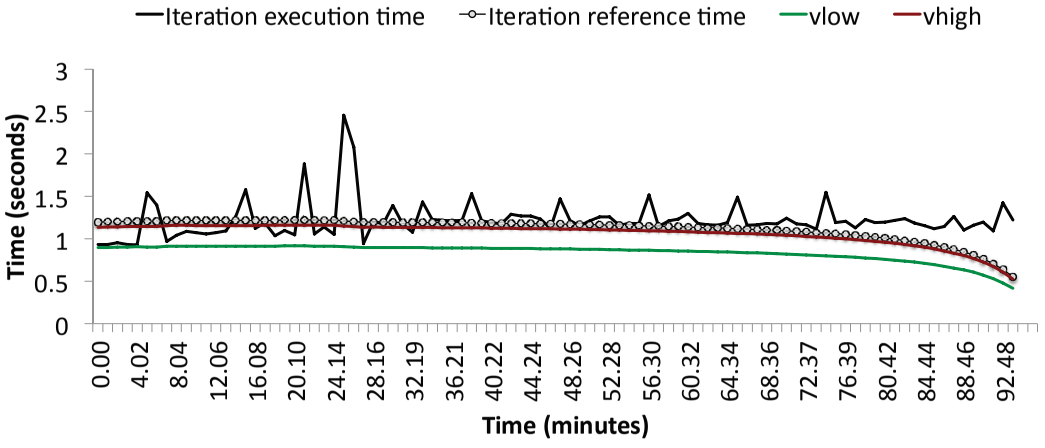
Figure 6.15: Application CPU and memory allocation for: (a) a deadline of 12000 seconds; (b) a deadline of 9000 seconds; (c) a deadline of 6000 seconds.



(a)



(b)



(c)

Figure 6.16: Application iteration execution time for: (a) a deadline of 12000 seconds; (b) a deadline of 9000 seconds; (c) a deadline of 6000 seconds.

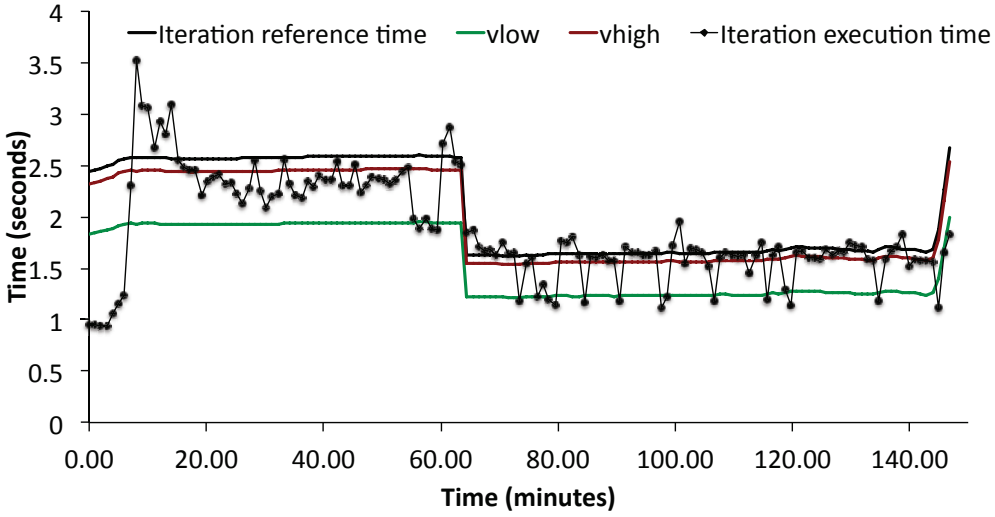


Figure 6.17: Application iteration execution time.

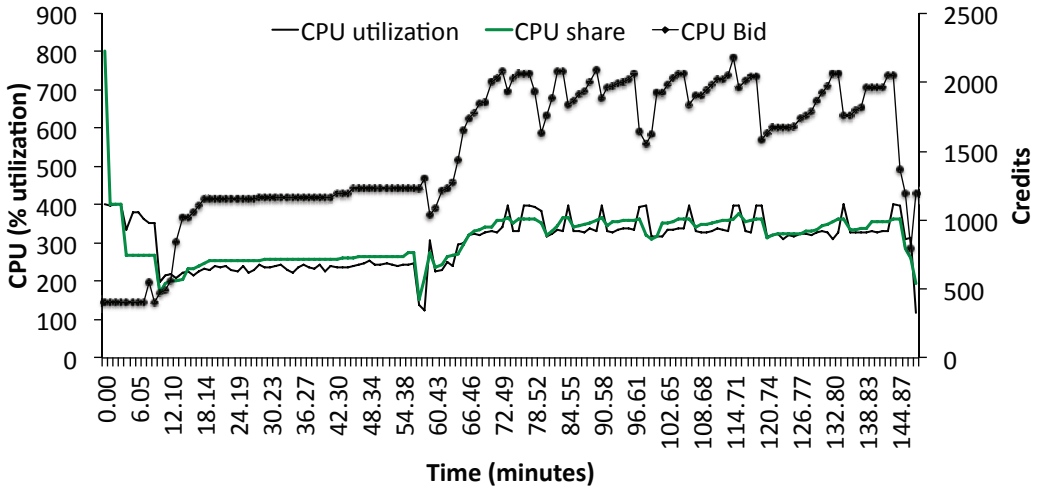


Figure 6.18: Application allocation.

Regarding the sharing of the node with other applications, we make the following observations:

- We also notice that there is a difference between the allocated share that the application should receive and the real utilization (e.g., like in the case of the memory allocation). As the other applications don't use all their allocated shares, resources are redistributed. Thus, the application's VMs use more resources at the same cost for the user.

- There are cases in which the system cannot ensure allocations ideally proportional to the bids, especially when pinning VCPUs to CPU cores. Thus, the application receives more resources than its share from the hypervisor. This leads in performance changes which cannot be adjusted properly by changing the bids.

- There are cases in which the performance interference from the incoming workloads could lead to deadline misses, e.g., like in the 6000-deadline application case, even if the application receives its full resource allocation. For this case, the VM maximum allocation might be set to the node capacity. The application controller could then increase the bid at a high enough value, leading to the migration of the other workload to different nodes. Then, to further increase the chances of the application to finish on time, more VMs can be added and the application can be reconfigured to take advantage of them.

6.5.3 Controller Behavior for Malleable Applications

In this section we analyze the application controller behavior for malleable applications. We consider as malleable applications two task processing frameworks: Condor and Torque. The SLO for each framework is to minimize the number of queued tasks. We first show how resources are shared fairly and dynamically between these frameworks and then how the changes in application budget are reflected in its allocation.

The use of these frameworks allows us to model a scenario in which Merkat can be used to set and change the limits on resources consumed by different departments sharing the same infrastructure. Each department wants to use its own framework. Merkat acts as a dynamic priority system: the frameworks can be started by an administrator with a pre-defined budget representing the initial framework priority. Then, if needed, the priority of one framework can be boosted by increasing its budget during its runtime.

6.5.3.1 Infrastructure Fair-sharing

To show how Merkat is used to share the infrastructure between these two frameworks we deployed them on a cloud of 20 physical nodes. We deployed the Condor framework to process parameter sweep applications and the Torque framework to process MPI applications. To each framework we assign a budget which allows it to provision as many VMs as to fill the infrastructure at a reserve price, i.e., 160 VMs. Thus, when the infrastructure is free a framework can use 160 VMs. When both frameworks have maximum a resource demand, they get half of the infrastructure. We submitted 33 Zephyr applications to Torque with execution parameters taken from a trace generated using a Lublin model [106]: the number of processors was generated between 1 and 8 and the execution time had an average of 2479 seconds with a standard deviation of 1243.5. We submitted 5000 jobs to Condor, simulating a parameter sweep application. As we did not have access to a real parameter sweep application, we used the *stress* benchmark, which ran a CPU intensive worker for different

time intervals, generated with a Gaussian distribution. The average task execution time was 478 seconds, with a standard deviation of 363.

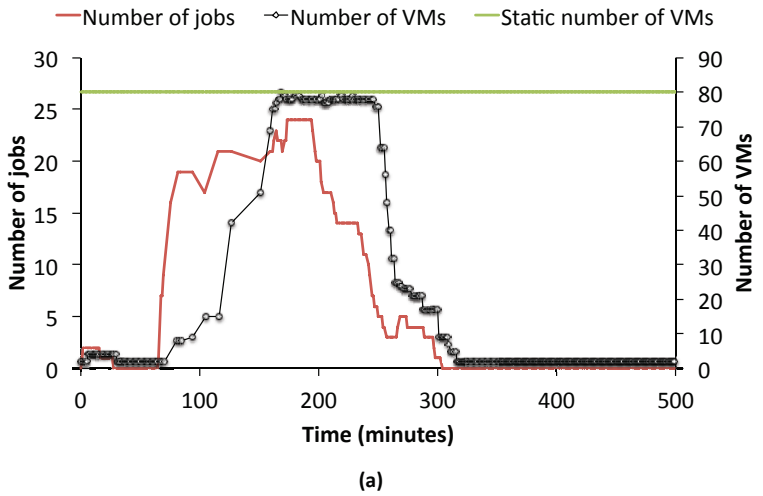


Figure 6.19: The provisioned number of VMs for the Torque framework.

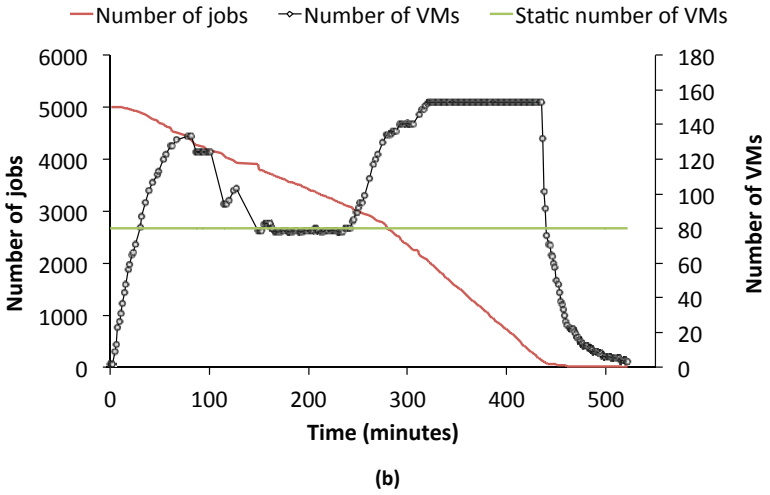


Figure 6.20: The provisioned number of VMs for the Condor framework.

Figure 6.19 and Figure 6.20 show the VM allocation variation in time. Figure 6.19 shows how the Torque controller provisions VMs while Figure 6.20 shows the provisioning made by Condor. For clarification, both figures include the number of queued jobs in the framework's queue, i.e., on the right axis (for Torque we measured the number of jobs and not the total number of demanded processors). It can be noticed how both controllers adapt the framework's resource demand to the variations in the number of queued jobs. If each framework would be assigned an equal share of the infrastructure, it would be able to provision a maximum of 80 VMs. However, in our case, the Condor framework is capable to take advantage of the under-utilization periods of the infrastructure.

We noticed a significant delay in provisioning VMs, as seen in Figure 6.19. During the experiment runtime, we had issues with many VMs which were booting slow, or the sshd daemon failed to start, making the SSH connection to them to fail, after a timeout period.

Our controller does not shut down the VM when the SSH connection fails, but it retries it. This retry period, although it proved to be useful, it also slows down the provisioning process.

6.5.3.2 Dynamic Priority Management

We show how Merkat can also be used to boost dynamically the priority of a framework in case of urgent requests. For this scenario we keep the previous setup. After 268 minutes from the experiment start we double the budget allocated to the Torque framework. Figure 6.21 shows the relation between the number of provisioned VMs and the number of jobs in queue. Initially, the Torque controller can provision a maximum of 80 VMs. When its budget is increased the application controller provisions 107 VMs and processes more jobs.

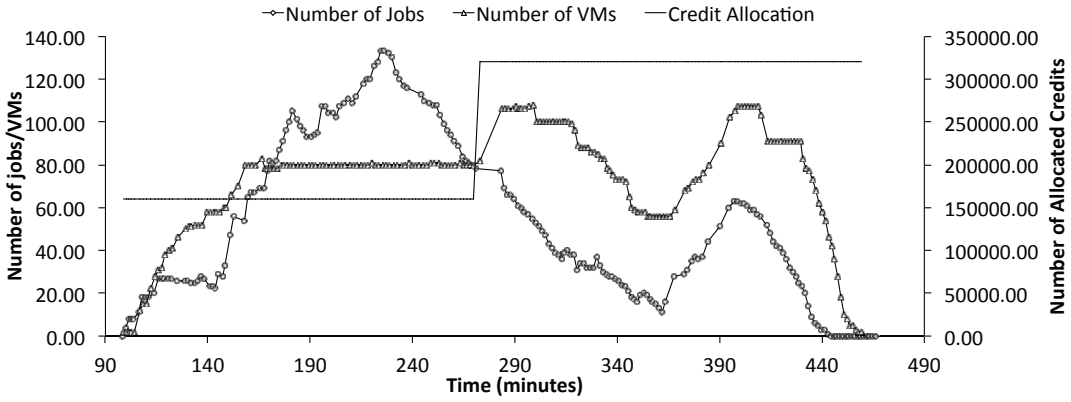


Figure 6.21: The influence of the budget change on the Torque processing performance.

Figure 6.22 shows the total CPU cluster share acquired by both Torque and Condor frameworks over time. The priority boost given by the budget increase can be noticed after 268 minutes from the experiment start, when the cluster share allocated to the Torque framework increases to up to 61%.

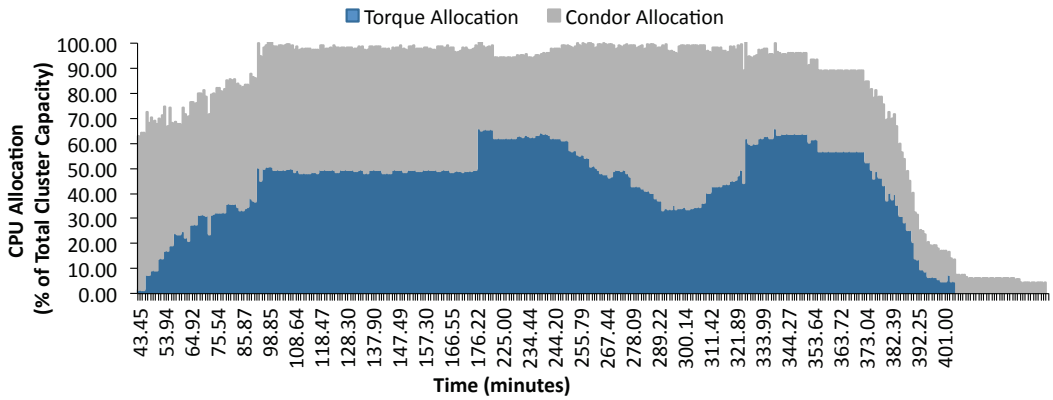


Figure 6.22: The influence of the budget change on the framework CPU allocation.

6.5.3.3 Discussion

We have seen from this experiment how an application controller can be designed to respond in application workload variations by adjusting dynamically the number of VMs. In our experiments we assigned an equal share of the cluster to each framework. In cases in which the framework's share needed to be increased during its runtime, the application controllers adjusted their demands to the priority change. In cases in which the frameworks receive different amounts of budgets, the allocation for each framework might oscillate, leading to many task resubmissions. However, the framework's controller can select which VMs to release to minimize the execution time loss for its jobs. Moreover, in cases in which the frameworks support suspend/resume mechanisms for their jobs, this cost reduces to the time to suspend/resume the jobs.

6.5.4 System Performance

To analyze the dynamic market behavior, we compare Merkat with a traditional batch scheduler, i.e., Maui [112]. We wanted to measure the satisfaction our system can provide to users. For this, we considered users which have the *full-deadline* utility functions detailed in Chapter 5: these users value the execution of their application at the assigned budget if the application finishes before deadline and at the negative budget value if the application doesn't meet its deadline. The satisfaction provided by the system is computed as the sum of the utilities.

We deployed a cloud on 4 nodes and we run 30 Zephyr applications, with 1 to 8 processes, each process in one VM, in two situations: (i) when the cloud is managed by Merkat; (ii) and then when it is managed by Torque and Maui. In the later case, we deploy VMs on each node and add them to Torque as physical nodes. As using a real workload trace would have lead to longer experiment execution time, which would not have been possible on Grid'5000, we used a Lublin model to generate the workload. We chose this model as it is realistic and easy to tune. The application execution time had an average of 2601 seconds with a standard deviation of 1133. All applications were submitted in an interval of 1311 seconds.

In Merkat each application ran with a SLO-driven controller, and had a budget inversely proportional to its deadline, reflecting the user's valuation for her application. The minimum assigned budget was 1022 credits (the minimum bid of a VM is 1000 credits) while the maximum assigned budget was 31910 credits. When executing applications, we counted 18 suspend and 15 resume commands. From the 30 submitted applications, 2 of them did not start their execution. Regarding the number of migrations, we counted in average 7 migrations per scheduling interval.

In Maui applications were submitted to the scheduler's queue. Maui used FCFS and backfilling algorithms to schedule applications to nodes. However, Maui is not aware of user budgets and application deadlines.

We counted 7 applications which did not finish successfully both in Maui and in Merkat. However, when we measured the total satisfaction we obtained that Merkat provides 2.79 times more value to users than Maui. The reason is that Maui does not know about user valuations, i.e., budgets in our case, and thus some of the most important applications failed to reach their time target, due to queue wait times. Nevertheless, in Merkat, applications which missed their deadlines were less valuable to their users, i.e., had a budget less than 1301 credits.

6.6 Conclusion

In this chapter we presented the implementation and evaluation of Merkat, our market-based PaaS prototype. We implemented in Merkat controllers for two application types: static MPI and malleable applications. Currently, these controllers can scale both application types vertically and horizontally, according to the application performance and infrastructure resource availability. We ran Merkat on the Grid'5000 testbed and evaluated the application controller behavior. We submitted these results to an international conference [54].

Our evaluation shows that Merkat meets our three requirements stated in Introduction:

SLO support flexibility: Merkat is a decentralized system that runs applications in virtual environments capable of adapting automatically to users' SLOs. This opens up the possibility to support different application types and user SLOs. Users, or developers, can take advantage of the mechanisms provided by Merkat to implement their own resource demand adaptation policies. The core of Merkat can be applied in different scenarios, from sharing resources between isolated applications to sharing resources between frameworks. We have shown through several examples that Merkat application controllers can apply our policies to adapt to dynamic changes in application requirements and resource availability. Nevertheless, an open issue remains the design of mechanisms that can be applied to provide stronger SLO guarantees and give users the posiblity to negociate the execution cost and time with the system.

Maximum resource utilization: Merkat currently uses a proportional-share policy to allocate fine-grained amounts of CPU and memory to VMs. This mechanism ensures maximum resource utilization of the infrastructure by automatically distributing all available resources among the running VMs. Nevertheless, in contention periods stronger guarantees might be needed, thus combining the proportional-share mechanism with other auction types could be desirable.

Fair resource utilization: Merkat implements a market. In contrast to a weight/priority-based systems, markets introduce the notion of resource cost and lead to a fair resource utilization. Resources have a dynamic price that, together with currency distribution policies, regulate the user behavior. To that extent, we have shown that, compared to traditional systems, the performance of Merkat is good, even in a context in which applications compete for resources and adapt in an uncoordinated fashion.

Conclusion

Organizations owning HPC infrastructures are facing a difficulty in managing their resources. This difficulty comes from the need to provide concurrent resource access to different application types considering that users might have different performance objectives for them. These organizations usually use batch schedulers to manage their resources. As until recently applications were having mostly static resource needs and did not require complex environments to run, these systems became popular. Nevertheless, as applications have become more dynamic and users have become more demanding regarding their execution environment or application performance objectives, such systems also proved to be inefficient in satisfying all these demands.

The emergence of the cloud computing paradigm, which promotes an "on-demand access" model to virtual resources, decreases the resource management complexity through its ability to manage elastic customized software environments for individual users. The organization can efficiently share its physical resources between different application types (e.g. MapReduce, MPI, or other legacy applications) by allowing each application to run in its own virtual cluster with limited interference from the infrastructure's administrator. The on-demand resource access is useful for applications which have time-varying resource demands as it ensures the right amount of resources with minimum delays.

Nevertheless, even in a context in which the infrastructure capacity is limited, cloud resource management models are simplistic and rely on the "infinite capacity" illusion. First, at the IaaS level, cloud providers offer virtual machines with static resource configurations, forcing applications to wait in queues while resources remain idle. Then, they provide poor resource regulation among users and their applications. The implemented resource quotas mechanisms are not sufficient to ensure a rational user behavior. More advanced PaaS solutions, which ensure automatic application management, use the interfaces provided by the IaaS cloud providers and fall in the same trap. Thus, a remaining challenge is to multiplex the limited infrastructure capacity between different application types while satisfying the users who execute them.

Contributions

As a step towards solving the previously mentioned challenge we defined a set of requirements for designing a resource management system: flexible application adaptation, efficient resource allocation and fair resource utilization. To provide flexible SLO support, a resource management system should meet four characteristics: (i) it should allow applications to run in isolated environments customized according to their required software configuration (isolation); (ii) it should allow applications to change their resource allocations dynamically over time (dynamic resource allocation); (iii) it should give applications the possibility to take decisions regarding the amount of allocated resources (decentralized resource control); (iv) it should also provide support for monitoring applications during

their runtime, as simply running applications on the acquired resources does not guarantee that they will meet their SLOs, due to unpredictable factors, such as performance interference, slow nodes or failures. To provide maximum resource utilization, resources should be allocated in a fine-grained manner and to provide fair resource utilization regulation mechanisms that guide users to value resources should be enforced. We analyzed the current state of the art in resource management systems and found a variety of solutions, which have the potential to address these stated requirements. Nevertheless, solutions that address at least one of the two requirements fail to meet the last one. In the same time, solutions that target fair resource utilization do not provide SLO support, dynamic resource allocation or isolation, and, thus, they don't meet the first requirement.

To meet all these previously stated requirements this dissertation proposes the design of a platform for application and resource management in private clouds. Our system applies an unique approach to multiplex the limited infrastructure capacity between different application types while maximizing the infrastructure utilization and providing support to meet different SLOs. By virtualizing resources, we give control to users over their software requirements. Our platform provides per-application isolation: each application runs in an environment customized to its needs. This environment, which we call virtual platform, can autonomously adapt its resource demand to meet the application SLO. The resource management is decentralized: each virtual platform can take resource demand adaptation decisions independently from the others. Thus, the platform becomes flexible and can support multiple application types and SLOs. To regulate user's access to resources we use a market mechanism. This market relies on a proportional-share policy, providing efficient resource allocation: resources in terms of CPU and memory are allocated in a fine-grained manner to the provisioned VMs. A unique aspect of this approach comes from the resource management decentralization mechanism. Basically, the acquired resources have a price that acts as a control mechanism to ensure that resources are distributed to applications according to the user's value for them. The market's currency distribution policy ensures that users receive limited budgets and value truthfully the execution of their applications. The combination of currency distribution and dynamic resource pricing ensures fair resource utilization. The proportional-share market approach was validated in a simulation-based prototype, called Themis, and with a real prototype, called Merkat, deployed on the Grid'5000 testbed.

This dissertation brings the following contributions:

A proportional-share model for dynamic VM resource allocation We implemented an algorithm which allocates proportional shares of CPU and memory to the running virtual machines and performs load balancing between physical nodes. This proportional-share model ensures fine-grained resource allocation among applications.

Two predefined resource demand adaptation policies We propose two resource demand adaptation policies to be used by applications running on the proportional-share market: (i) a policy that adapts the application resource demand per VM to the current resource availability and resource price; (ii) a policy that adapts the application resource demand in terms of number of VMs to the current resource availability and resource price. These policies can also be combined for better application execution.

A generic platform for application and resource management We designed a platform which uses the previously mentioned approach to provide users with

generic automatic application management mechanisms, allowing them to implement their own resource demand policies. The platform is extensible, allowing new scheduling algorithms to be developed and new application types to be integrated on the infrastructure.

An evaluation of the proposed approach We evaluated the proposed approach by simulation and in a real environment. We implemented the proportional-share market in CloudSim, in a system called Themis. We evaluated the performance of the proportional-share market in terms of total user satisfaction when applications adapt their resource demands per VM to track a user-given SLO [53, 55]. Our results show that, despite its decentralized nature, compared to a centralized system, the system performs well in light load periods, while its performance degrades in high load periods. The performance degradation, also known as the Price of Anarchy, is caused by the decentralized nature and the application selfish behavior. We also implemented our approach in a prototype, called Merkat [54]. We have tested Merkat with two application types: static MPI applications and malleable task processing frameworks. We have shown in both cases that resources can be allocated dynamically among applications which are SLO-driven. Our results show that: (i) Merkat is flexible, allowing the co-habitation of different applications and policies on the infrastructure; (ii) despite application continuous adaptation, application co-location and VM operations, Merkat can achieve better user satisfaction than a traditional resource management system.

Perspectives

The obtained results make us believe that the platform we propose, materialized through Merkat, is easy to adapt to a variety of scenarios. We plan to release Merkat as an Open Source Software for the community. Merkat could be useful for commodity clusters, used by users with various software configuration requirements. Infrastructure administrators could benefit from Merkat due to its flexible resource allocation mechanisms. Users could also benefit from its automatic resource demand capabilities. We envision that Merkat can be applied not only by an organization to manage its internal resources, but also by a cloud provider. In the last case, Merkat provides the building blocks for further research in the area of resource demand adaptation policies for providing stronger SLO guarantees to different application types.

We present next some concrete perspectives to improve and extend our approach.

Scalability

The scalability of Merkat should be evaluated and improved. As the current scale on which Merkat was tested was quite limited, also due to limitations in the underlying third party software stack, we could not identify the upper limit on the performance of Merkat when the scale increases. It would be interesting to see the efficiency limitations of the VM Scheduler in performing allocations when it has to cope with adaptation requests from a large number of applications. To allow Merkat to be deployed on a larger scale we plan to use the scalable and autonomic cloud manager developed in the Myriads team [67].

High Availability

To provide users with a production system, Merkat services require self-healing capabilities to make the platform highly available. This characteristic is important, as users might be dissatisfied if the system is not reliable. For example, if the Applications Management service would become unavailable, users could not deploy their applications. Or, if the VM Scheduler would become unavailable, VMs could not be deployed on the nodes and the application controllers could not adapt their resource demands. Finally, in the case of the Virtual Currency Manager, applications could not run as budget management operations would be unaccesible. We are currently investigating the possibility to use replication techniques [87] to achieve these goals.

Support for Evolving Applications

A future work direction is to add support in Merkat for evolving applications. As a concrete example, in the context of the collaboration with EDF, we are working with two researchers from EDF to reconfigure dynamically a scientific application [48], according to its computational needs. This application performs a computational fluid dynamics simulation used to analyze the fluid flow inside the reactor of a nuclear plant. When the temperature increases, turbulence appears in the fluid and additional computational modules are started to model this phenomenon. With traditional resource management systems, to run this application, a scientist needs to estimate the maximum amount of resources and start the application with this amount. When the additional computational modules are not used, however, the application performance is poor, as the communication cost is higher than the computation amount. In this case, the challenge is to run the application as efficiently as possible. As a first step towards solving this challenge, we want to develop an application controller that acquires and releases the needed resources dynamically during the application runtime. First, we want to create a performance profile of the application on its given data set. Then, we can change the resource demand adaptation policies currently implemented in our controller for MPI static applications as follows. When the application starts, the application controller selects the maximum number of resources on which the application will run efficiently. Then, when the turbulence starts, i.e., the execution time per iteration increases significantly, the application controller will use the application profile to select the new number of VMs and will reconfigure the application to use them. We have already implemented a simple policy to reconfigure the application. However, we lacked a valid data input to validate the proposed policy.

Sharing Virtual Platforms among Trusted User Groups

Currently, Merkat provides isolation between applications: each application runs in its own virtual platform. In the case of task processing frameworks, users submit applications to the framework's scheduler directly, with no knowledge of the Merkat's existence. After receiving feedback from EDF researchers, however, we discovered that users might want different isolation levels for their applications. Some users might want to keep a virtual platform idle and re-use it for the execution of the same application. Others might want to share a running virtual platform with users belonging to a trusted group. To support these cases Merkat needs to start virtual platforms with different sharing options. Depending on the sharing option, users can be allowed or not to submit multiple applications to the same virtual platform by interacting with Merkat. Application controllers can be designed to manage requests coming from different users and enforce access restrictions.

Improving the Current Resource Management Policies

We envision several research directions to improve the resource management algorithms.

Placement and migration preferences. A first direction would be to provide the capability to express placement and migration preferences for the applications. The advantage in this case would be the use of the price to take into account application preferences. Some applications might want to have their virtual machines kept continuously on the same nodes instead of having them migrated while paying a higher resource price. Other applications might have placement constraints: their tasks could run better on specific nodes as these nodes might store the data they need to process. Critical applications might have preferences regarding the node availability: they could pay a higher price to run on the most reliable nodes driving away poor applications which were already running on them. Extending the current API to allow expressing these preferences and modifying the current algorithms is an open challenge.

SLO guarantees. Currently, users still have to cope with a certain level of uncertainty when executing their applications on our market. When facing the need to specify a cost for their application execution, users consider it a difficult task as the resource price is volatile. Given a large scale of the system, price prediction algorithms [148] can be implemented in the application controller code to provide better guarantees. These algorithms can be used to answer questions like: *what is the maximum amount I need to pay to meet a certain deadline?* or *what is the deadline that can be met with this budget?*.

Multi-cloud VM provisioning. Merkat can also be extended to provision VMs from public clouds, if specific users beloging to the organization request it. Leaving up to the user the choice to get VMs from the public cloud is beneficial, as some users might have critical applications and might require to run them specifically on the organization's cluster.

I/O resources. The current resource allocation algorithms could be extended to consider network or storage resources. Such resources can become a bottleneck when network or data intensive applications are executed. Thus, it is normal to make them available at a price that reflects the total resource demand. Regarding the network resource, software tools can be used to limit the bandwith available to each virtual machines and ensure a proportional share. We envision that similar mechanisms can be applied for storage too.

Virtual Currency Management. The virtual currency distribution policies should be further investigated. Currently, there is no mean of setting the total currency amount that the system has available and we rely on the knowledge of the administrator to set the user budget limits. The administrator has to cope with difficult decissions in this case. If she sets the total currency amount too high, prices will become inflated and users will not meet their SLOs. If she sets the total currency amount too low, users will starve as they could not buy resources. An automatic system to help the administrator take these decissions could be designed. This system could use as feedback past price fluctuations and reports from users regarding their SLO violations.

A Double Auction-based Resource Allocation Protocol

A direction parallel with the previous ones would be to make the application controllers more aware of the other infrastructure occupants. In our system each application controller

changes its bid independently and makes decisions to suspend and resume the application based only on the current price. As these decisions are not coordinated with each other, they might lead to non necessary price fluctuations. A double auction can be implemented to allow applications to communicate their value for resources and decide the resource allocations. This auction would be applied when the infrastructure is full. Running application controllers would have the chance to sell their resources at a price which reflects the value of not finishing their application on time, by posting a resource offer. In this case, users might gain credits by giving up resources to more urgent applications. Receiving currency for their decision to relinquish resources provides users with incentives to collaborate. Starting application controllers will then decide if it's worth buying their resources and running their applications based on the importance users assign to them. Developing price computation policies for the application controllers and algorithms to answer to the SLO guarantee questions is an open research direction. Merkat can support this model with few modifications, as the VM Scheduler can act as an auctioneer between the sellers and the buyers. It would be interesting to compare such a model with the currently implemented one, in terms of scalability and application performance.

Evaluation of Merkat in a Real-world Environment

It would be interesting to run our system in a real-world environment over long-term periods. The obtained results will help answering the question: *How much satisfaction will users get from running their applications on a market-based cloud, compared to the previously-used batch schedulers?* Currently, to our knowledge, although market-based solutions were thoroughly investigated, only a few of them were deployed in a real setting. Given a choice between a market-based system and a traditional one, users sharing distributed resources preferred using the latter, due to its ease of use. This was the case of Bellagio, a market-based system deployed on PlanetLab [17]. We hope that getting real data from users running applications on our system will help in establishing the practical benefits, or disadvantages, in using a market.

To this end, EDF can provide a large user-base. Efforts were successfully made to unify the access to resources through a software, similar to a PaaS solution, called VISHNU [99]. This middleware consists in deploying a set of services on the top of the HPC infrastructure. A user or an application can easily interact with the services through an API or a web portal. Nevertheless, on the long term, a migration to a private cloud infrastructure is envisioned. This migration will provide internal EDF researchers and developers, thus hundreds of users, with even more flexibility by allowing them to run their applications in controlled software environments and scale them "on-demand". Connecting VISHNU with Merkat's capabilities will give users a choice of running their applications on the Merkat managed cloud and will provide us with real feedback regarding the satisfaction users can get when using a market-based system.

Bibliography

[1] T. 500. http://www.top500.org/.

[2] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[3] T. G. Alliance. www.globus.org.

[4] A. Amar, R. Bolze, A. Bouteiller, A. Chis, Y. Caniou, E. Caron, P. Kaur Chouhan, G. Le Mahec, H. Dail, B. Depardon, F. Desprez, J.-S. Gay, and A. Su. DIET: New Developments and Recent Results. In *Euro-Par 2006 Workshops - Parallel Processing*, pages 150–170, 2006.

[5] AmazonAutoScaling. http://aws.amazon.com/autoscaling/.

[6] AmazonEBS. http://aws.amazon.com/.

[7] AmazonEC2. http://aws.amazon.com/s3/.

[8] AmazonElasticBeanstalk. http://aws.amazon.com/elasticbeanstalk.

[9] AmazonS3. http://aws.amazon.com/ebs/.

[10] AmazonVPC. http://aws.amazon.com/vpc.

[11] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[12] P. Anedda, S. Leo, S. Manca, M. Gaggero, and G. Zanetti. Suspending, migrating and resuming hpc virtual clusters. *Future Generation Computer Systems*, 26(8):1063–1072, 2010.

[13] M. Annaratone. Mpps, amdahl's law, and comparing computers. In *Fourth Symposium on the Frontiers of Massively Parallel Computation.*, pages 465–470. IEEE, 1992.

[14] AppFog. http://www.appfog.com/.

[15] AppScale. http://www.appscale.com/.

[16] P. W. Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[17] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure*, volume 9, 2004.

[18] A. I. Avetisyan, R. Campbell, I. Gupta, M. T. Heath, S. Y. Ko, G. R. Ganger, M. A. Kozuch, D. O'Hallaron, M. Kunze, T. T. Kwan, et al. Open cirrus: A global cloud computing testbed. *Computer*, 43(4):35–43, 2010.

[19] M. Azure. http://www.windowsazure.com/.

[20] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX distributed operating system: load balancing for UNIX*. Springer-Verlag New York, Inc., 1993.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[22] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[23] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 2006.

[24] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions of Computing Systems*, 15(4):412–447, Nov. 1997.

[25] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1507–1542, 2002.

[26] R. Buyya, R. Ranjan, and R. N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010.

[27] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software Practice and Experience*, 41(1):23–50, 2011.

[28] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya. The aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds. *Future Generation Computer Systems*, 28(6):861–870, 2012.

[29] E. Carlini, M. Coppola, P. Dazzi, L. Ricci, and G. Righetti. Cloud federations in contrail. In *Euro-Par 2011: Parallel Processing Workshops*, pages 159–168. Springer, 2012.

[30] A. Casajus, R. Graciani, S. Paterson, and A. Tsaregorodtsev. Dirac pilot framework and the dirac workload management system. In *Journal of Physics: Conference Series*, volume 219, page 062049. IOP Publishing, 2010.

[31] H. Casanova, F. Berman, G. Obertelli, and R. Wolski. The apples parameter sweep template: User-level middleware for the grid. In *Proceedings of the ACM/IEEE 2000 Supercomputing Conference*, pages 60–60. IEEE, 2000.

[32] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings of 12th IEEE International Symposium on High Performance Distributed Computing, 2003.*, pages 90–100. IEEE, 2003.

[33] C. Chaubal. The architecture of vmware esxi. *VMware White Paper*, 2008.

[34] J. Q. Cheng and M. P. Wellman. The walras algorithm: A convergent distributed implementation of general equilibrium outcomes. *Computational Economics*, 12(1):1–24, 1998.

[35] K. Chodorow and M. Dirolf. *MongoDB: the definitive guide*. O'Reilly Media, 2010.

[36] chroot. http://linux.die.net/man/1/chroot.

[37] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. Institute of Electrical and Electronics Engineers, 2005.

[38] B. N. Chun and D. E. Culler. Rexec: A decentralized, secure remote execution environment for clusters. In *Proceedings of the 4th International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, CANPC '00, pages 1–14, London, UK, UK, 2000. Springer-Verlag.

[39] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *Proceedinfs of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 30–30. IEEE, 2002.

[40] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, and J. Sauvé. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of the 2003 International Conference on Parallel Processing*, pages 407–416, 2003.

[41] B. Claudel, G. Huard, and O. Richard. Taktuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, pages 91–100. ACM, 2009.

[42] H. Cloud. https://www.hpcloud.com/.

[43] P. Cloud. http://www.picloud.com/.

[44] CloudControl. http://www.cloudcontrol.com/.

[45] CloudFoundry. http://www.cloudfoundry.com/.

[46] Cloudify. www.cloudifysource.org/.

[47] CloudStack. http://cloudstack.apache.org/.

[48] CodeSaturne. Codesaturne : a finite volume code for the computation of turbulent incompressible flows - industrial applications. *International Journal on Finite Volumes*, 2004.

[49] G. Compute. https://cloud.google.com/products/compute-engine.

[50] B. F. Cooper and H. Garcia-Molina. Bidding for storage space in a peer-to-peer data preservation system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems.*, pages 372–381. IEEE, 2002.

[51] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto, and A. Reinefeld. XtreemOS: a Vision for a Grid Operating System. Technical report, XtreemOS, 2008.

[52] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. An economic approach for application qos management in clouds. In *Proceedings of the 6th Workshop on Virtualization in High-Performance Cloud Computing*, Europar Workshops, 2011.

[53] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. Themis: Economy-based automatic resource scaling for cloud systems. In *Proceedings of IEEE International Conference on High Performance Computing and Communications*, HPCC '12, 2012.

[54] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. Merkat: A market-based slo-driven cloud platform - under submission. 2013.

[55] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. On the use of a proportional-share market for application slo support in clouds. In *Proceedings of the 19th International European Conference on Parallel and Distributed Computing*, Europar'13, 2013.

[56] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[57] S. Delamare, G. Fedak, D. Kondo, and O. Lodygensky. Spequlos: a qos service for bot applications using best effort distributed computing infrastructures. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'12, pages 173–186. ACM, 2012.

[58] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, and M. Loichate. Building a mosaic of clouds. In *Euro-Par 2010 Parallel Processing Workshops*, pages 571–578. Springer, 2011.

[59] P. DuBois. *MySQL*. Addison-Wesley Professional, 2009.

[60] W. Emeneker and D. Stanzione. Dynamic virtual clustering. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 84–90, Washington, DC, USA, 2007. IEEE Computer Society.

[61] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott. Proactive fault tolerance using preemptive migration. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing.*, pages 252–257. IEEE, 2009.

[62] S. G. Engine. http://www.sun.com/software/sge.

[63] EnStratus. www.enstratius.com/.

[64] A. C. Enterprises. Inc.: Moab workload manager administrator guide, version 6.0. 2.

[65] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[66] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[67] E. Feller, L. Rilling, and C. Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 482–489. IEEE Computer Society, 2012.

[68] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic algorithms for load balancing in distributed computer systems. In *International Conference on Distributed Computer Systems*, volume 499, 1988.

[69] J. R. Figueiredo, P. A. Dinda, and J. A. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[70] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Morgan Kaufmann, 2003.

[71] F. Glover, M. Laguna, et al. *Tabu search*, volume 22. Springer, 1997.

[72] D. Goldenberg. Infiniband device virtualization in xen. *Xen summit, January*, 19, 2006.

[73] GoogleApps. http://www.google.com/apps.

[74] F. Grid. http://www.windowsazure.com/.

[75] L. Grit, D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht. Harnessing virtual machine resource control for job management. In *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing*, 2006.

[76] L. E. Grit. *Extensible resource management for networked virtual computing*. ProQuest, 2007.

[77] N. Grozev and R. Buyya. Inter-cloud architectures and application brokering: taxonomy and survey. *Software Practice and Experience*, 2012.

[78] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[79] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID'12, pages 644–651. IEEE, 2012.

[80] F. Hermenier, X. Lorca, and J.-M. Menaud. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, 2009.

[81] Heroku. http://www.heroku.com/.

[82] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, 2008.

[83] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of USENIX conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[84] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, pages 51–60. ACM, 2009.

[85] W. Huang, J. Liu, B. Abali, and K. D. Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international Conference on Supercomputing*, 2006.

[86] A. C. Hume, Y. Al-Hazmi, B. Belter, K. Campowsky, L. M. Carril, G. Carrozzo, V. Engen, D. García-Pérez, J. J. Ponsatí, R. Kűbert, et al. Bonfire: A multi-cloud test facility for internet of services experimentation. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 81–96. Springer, 2012.

[87] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[88] E. G. Infrastructure. www.egi.eu.

[89] A. S. Instances. http://aws.amazon.com/ec2/spot-instances/.

[90] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-recharging virtual currency. In *Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 93–98. ACM, 2005.

[91] KaVLAN. https://www.grid5000.fr/mediawiki/index.php/kavlan.

[92] K. Keahey, P. Armstrong, J. Bresnahan, D. LaBissoniere, and P. Riteau. Infrastructure outsourcing in multi-cloud environment. In *Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit*, 2012.

[93] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual workspaces: Achieving quality of service and quality of life in the grid. *Scientific Programming*, 13(4):265–275, 2005.

[94] K. Keahey and T. Freeman. Contextualization: Providing one-click virtual clusters. In *Proceedings of the 4th IEEE International Conference on eScience.*, pages 301–308. IEEE, 2008.

[95] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[96] N. Kiyanclar, G. A. Koenig, and W. Yurcik. Maestro-vc: A paravirtualized execution environment for secure on-demand cluster computing. In *Proceedings of the 6th IEE International Symposium on Cluster Computing and the Grid.*, volume 2, pages 12–pp. IEEE, 2006.

[97] C. Klein and C. Perez. An rms architecture for efficiently supporting complex-moldable applications. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, HPCC '11, 2011.

[98] C. Klein and C. Pérez. An rms for non-predictably evolving applications. In *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 326–334. IEEE, 2011.

[99] S. Kortas and B. Depardon. Vishnu/sysfera-ds : Un portail d'accès unifié aux ressources des centres de calcul. mise en application à edf r&d. *Journées scientifiques mésocentres et France Grilles, supercalculateurs, grilles et clouds : des outils complémentaires pour la science*, 2012.

[100] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[101] E. Laure, S. Fisher, A. Frohner, C. Grandi, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, et al. Programming the grid with glite. *Computational Methods in Science and Technology*, 12(1):33–45, 2006.

[102] A. A. Lazar and N. Semret. Design and analysis of the progressive second price auction for network bandwidth sharing. *Telecommunication Systems-Special issue on Network Economics*, 20:255–263, 1999.

[103] C. B. Lee and A. E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, pages 107–116. ACM, 2007.

[104] libvirt. libvirt.org/.

[105] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, 1988.

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

[107] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[108] LXC. http://lxc.sourceforge.net/.

[109] C. E. Manager. http://opennebula.org/software:ecosystem:carina.

[110] P. Marshall, K. Keahey, and T. Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52. IEEE Computer Society, 2010.

[111] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai. The design and evolution of live storage migration in vmware esx. In *USENIX ATC*, volume 11, pages 1–14, 2011.

[112] Maui. http://www.nsc.liu.se/systems/retiredsystems/grendel/maui.html.

[113] M. Maurer, I. Brandic, and R. Sakellariou. Enacting slas in clouds using rules. In *Euro-Par 2011 Parallel Processing*, pages 455–466. Springer, 2011.

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

[115] P. B. Menage. Adding generic process containers to the linux kernel. In *Proceedings of the Linux Symposium*, volume 2, pages 45–57. Citeseer, 2007.

[116] Mendix. http://www.mendix.com/.

[117] T. Metsch, A. Edmonds, R. Nyrén, and A. Papaspyrou. Open cloud computing interface–core. In *Open Grid Forum, OCCI-WG, Specification Document. Available at: http://forge. gridforum. org/sf/go/doc16161*, 2010.

[118] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[119] Moab. http://www.clusterresources.com/products/moab-cluster-suite.php.

[120] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, and L. Rilling. Kerrighed: a single system image cluster operating system for high performance computing. In *Euro-Par 2003 Parallel Processing*, pages 1291–1294. Springer, 2003.

[121] M. A. Murphy, M. Fenn, and S. Goasguen. Virtual organization clusters. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 401–408, Washington, DC, USA, 2009. IEEE Computer Society.

[122] A. C. Murthy, C. Douglas, M. Konar, O. O'Malley, S. Radia, S. Agarwal, and V. KV. Architecture of next generation apache hadoop mapreduce framework. Technical report, Technical report, Apache Hadoop community, 2011.

[123] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the 2nd International Conference on Autonomic Computing*, ICAC '05, pages 229–240, Washington, DC, USA, 2005. IEEE Computer Society.

[124] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for hpc with xen virtualization. *Proceedings of the 21st International Conference on Supercomputing (ICS)*, page 23, 2007.

[125] H. Nguyen Van, F. Dang Tran, and J.-M. Menaud. Autonomic virtual resource management for service hosting platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 1–8. IEEE Computer Society, 2009.

[126] Nimbus. www.nimbusproject.org.

[127] J. Norris, K. Coleman, A. Fox, and G. Candea. Oncall: Defeating spikes with a free-market application cluster. In *Proceedings of the 2004 International Conference on Autonomic Computing*, ICAC'04, pages 198–205. IEEE, 2004.

[128] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[129] OAR. oar.imag.fr/.

[130] Openshift. http://www.openshift.com/.

[131] OpenStack. http://www.openstack.org/.

[132] OpenVZ. http://wiki.openvz.org.

[133] paramiko. www.lag.net/paramiko/.

[134] S.-M. Park and M. Humphrey. Self-tuning virtual machines for predictable escience. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:356–363, 2009.

[135] N. Phantom. www.nimbusproject.org/phantom.

[136] G. Pierre, I. Helw, C. Stratan, A. Oprescu, T. Kielmann, T. Schutt, J. Stender, M. Artac, and A. Cernivec. ConPaaS: an integrated runtime environment for elastic cloud applications. In *Proceedings of the Workshop on Posters and Demos Track (PDT '11)*, 2011.

[137] G. Pierre and C. Stratan. Conpaas: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 2012.

[138] RabbitMQ. http://www.rabbitmq.com/.

[139] Rackspace. http://www.rackspace.com/cloud/.

[140] L. Ramakrishnan, D. Irwin, L. Grit, A. Yumerefendi, A. Iamnitchi, and J. Chase. Toward a doctrine of containment: grid hosting with adaptive resource control. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 101. ACM, 2006.

[141] Redis. redis.io/.

[142] O. Regev and N. Nisan. The popcorn market. online markets for computational resources. *Decision Support Systems*, 28(1):177–189, 2000.

[143] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[144] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *Computer*, 38(5):63–69, 2005.

[145] P. Ruth, P. McGachey, and D. Xu. Viocluster: virtualization for dynamic computational domains. In *Proceedings of IEEE International Conference on Cluster Computing*, 2005.

[146] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *IEEE International Conference on Autonomic Computing, 2006*, pages 5–14. IEEE, 2006.

[147] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation or the sun network filesystem, 1985.

[148] T. Sandholm and K. Lai. A statistical approach to risk mitigation in computational markets. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 85–96, New York, NY, USA, 2007. ACM.

[149] T. Sandholm and K. Lai. Dynamic proportional share scheduling in hadoop. In *15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.

[150] T. Sandholm, K. Lai, and S. Clearwater. Admission control in a computational market. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID'08, pages 277–286, Washington, DC, USA, 2008. IEEE Computer Society.

[151] Scalr. scalr.com/.

[152] T. W. Scheduler. www-142.ibm.com/software/products/us/en/tivoworksche/.

[153] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European conference on Computer systems*, Eurosys'13, 2013.

[154] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[155] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: a computational economy-based job scheduling system for clusters. *Software Practice and Experience*, 34:573–590, 2004.

[156] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[157] B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, pages 87–96. ACM, 2008.

[158] B. Sotomayor, R. Montero, I. Llorente, and I. Foster. An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[159] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings of Parallel Processing Workshops*, pages 514–519. IEEE, 2002.

[160] G. Staples. Torque resource manager. In *Proceedings of ACM/IEEE conference on Supercomputing*, 2006.

[161] Starcluster. http://star.mit.edu/cluster/.

[162] M. Stillwell, F. Vivien, and H. Casanova. Dynamic fractional resource scheduling for HPC workloads. In *In Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'10, pages 1–12. IEEE, 2010.

[163] I. Stoica, H. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *Job Scheduling Strategies for Parallel Processing*, pages 200–218. Springer, 1995.

[164] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, 1996.

[165] R. Sudarsan and C. J. Ribbens. Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In *Proceedings of the 2007 International Conference on Parallel Processing*, pages 44–44. IEEE, 2007.

[166] I. E. Sutherland. A futures market in computer time. *ACM Communications*, 11(6), June 1968.

[167] P. B. System. http://www.pbsworks.com/.

[168] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency Practice and Experience*, 17(2-4), 2005.

[169] S. Tsunami. code.google.com/p/scp-tsunami/.

[170] Twisted. twistedmatrix.com/.

[171] S. S. Vadhiyar and J. J. Dongarra. A metascheduler for the grid. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing*, pages 343–351. IEEE, 2002.

[172] G. Vallee, T. Naughton, C. Engelmann, H. Ong, and S. Scott. System-level virtualization for high performance computing. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2008.

[173] H. N. Van, F. Tran, and J.-M. Menaud. Performance and power management for cloud infrastructures. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 329–336, 2010.

[174] L. van Doorn. Hardware virtualization trends. In *Proceedings of the 2nd ACM/Usenix International Conference On Virtual Execution Environments*, volume 14, pages 45–45, 2006.

[175] A. Velte and T. Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[176] VirtualBox. http://www.virtualbox.org/.

[177] L. VServer. http://linux-vserver.org/.

[178] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.

[179] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[180] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of the 2004 International Conference on Autonomic Computing.*, pages 70–77. IEEE, 2004.

[181] Z. Wang, Y. Chen, D. Gmach, S. Singhal, B. Watson, W. Rivera, X. Zhu, and C. Hyser. Appraise: application-level performance management in virtualized server environments. *IEEE Transactions on Network and Service Management*, 6(4):240 –254, 2009.

[182] Westgrid. www.westgrid.ca/.

[183] J. Wilkes. Utility functions, prices, and negotiation. *Wiley Series on Parallel and Distributed Computing - Market Oriented Grid and Utility Computing.*, pages 67–88, 2008.

[184] WMware. http://www.wmware.com.

[185] R. Wolski, J. S. Plank, T. Bryan, and J. Brevik. G-commerce: Market formulations controlling resource allocation on the computational grid. In *Proceedings of hte 15th International Parallel and Distributed Processing Symposium.*, pages 8–pp. IEEE, 2001.

[186] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

[187] V. Workstation. http://www.vmware.com/products/workstation/.

[188] XSDE. www.xsede.org.

[189] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, 2008.

[190] C. S. Yeo and R. Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Software: Practice and Experience*, 36(13):1381–1419, 2006.

[191] C. S. Yeo and R. Buyya. Pricing for utility-driven resource management and allocation in clusters. *International Journal of High Performance Computing Applications*, 21(4):405–418, 2007.

[192] C. S. Yeo, S. Venugopal, X. Chu, and R. Buyya. Autonomic metered pricing for a utility computing service. *Future Generation Computer Systems*, 26(8):1368–1380, 2010.

[193] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[194] S. Zaman and D. Grosu. Combinatorial auction-based allocation of virtual machine instances in clouds. *Journal of Parallel and Distributed Computing*, 2012.

[195] Zephyr. https://github.com/kortas/zephyr.

[196] ZeroMq. http://www.zeromq.org/.

[197] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 2–12, New York, NY, USA, 2005. ACM.

[198] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1):45–57, 2009.

Appendix A

Merkat API

This chapter describes how users can interact with the Merkat services by using their provided APIs and CLI tools.

$< price >$ getPrice()
$vclusterId$ createVCluster(controllerId, nvms, template, b, extra)
$(success, < vmsIds >)$ editVCluster(controllerId, vclusterId, nvms, <vmsIds>, b, extra)
$success$ suspendVCluster(vclusterId)
$success$ resumeVCluster(vclusterId)
$success$ deleteVCluster(vclusterId)
$cost$ getCost(allocation)
$< vclusters >$ listVClusters()

Table A.1: VM Scheduler API.

The VM Scheduler Table A.1 details the VM Scheduler's API. The functions composing the API are:

- *createVCluster*: this function creates *nvms* VMs from an IaaS cloud template *template* at a bid *b*. The time duration for which the virtual cluster should run is specified by the parameter *extra*. The VMs are not deployed on the nodes but put in the VM scheduler's queue. The VM scheduler calls a periodic auction to compute the resource allocations and deploy the VMs on the nodes. For resource charging, the VM scheduler keeps a mapping of virtual cluster-to-application controller.
- *editVCluster*: this function allows changing virtual cluster parameters. The virtual cluster can be shrinked/expanded, its time duration changed, or the value of its bid modified. The *nvms* parameter gives the new number of VMs. If the virtual cluster is shrinked, the $< vmIds >$ vector specifies the VMs to be shutdown. When the virtual cluster is expanded, this function returns the VMs that were created. If *nvms* is not changed and VMs are to be deleted, new VMs are created to replace them.
- *suspendVCluster*: this function suspends all the VMs from the virtual cluster. The suspend process is not syncronized, i.e., the VMs might not be suspended exactly at the same time.
- *resumeVCluster*: this function resumes all the VMs from the virtual cluster. Similar to suspendVCluster, the resume process is not syncronized.

- *deleteVCluster*: this function destroys all the running or suspended VMs.
- *listVClusters*: this function lists all the existing virtual clusters.
- *getPrice* and *getCost*: these two functions provide information about the current resource price and the estimated cost of getting a virtual cluster with a resource allocation $< allocation >$.

Users can interact directly with the scheduler through a command line interface client, called *Merkat_ lease*.

success register(controllerId, amount)
success charge(<controllerId, amount>)
success renew(controllerId, amount)
success delete(controllerId)
success edit(controllerId, amount)
success edit(controllerId, amount)
success editUser(user, weight)

Table A.2: Virtual Currency Manager API

The Virtual Currency Manager provides six main functions, as illustrated in Table A.2:

- *charge*: this function substracts from the application controller account the specified *amount* of credits. This function is actually called by the VM Scheduler to charge the application controller accounts.
- *register*: this function is called by the Applications Manager to register the application controller with the Virtual Currency Manager. The Virtual Currency Manager creates an account for it and transfers the specified *amount* of credits from the user's account to it.
- *renew*: this function, called by the Applications Manager on behalf of the user, transfers an *amount* of credits from the user's account to the application controller account. The transfer is done if the accumulated amount of credits in the account is not higher than the value specified at submision time.
- *edit*: this function, called by the Applications Manager on behalf of the user, increases the value of the application controller account to the new specified *amount*. Any renew call made after this function will transfer credits up to this new amount.
- *delete*: the Applications Manager calls this function to delete the application controller account.
- *editUser*: this function changes the current weight of the user. This weight is set by the administrator and is used to distribute budget among users. Initially, all users in the system receive the same weight. The infrastructure administrator calls this function to change a weight and give more/less importance to a specific user.

Users can interact with the Virtual Currency Manager through the *Merkat_ users* client.

controllerId createPlatform(template)
controllerId submitApplication(template)
success editPlatform(controllerId, <parameter, value>)
success suspendPlatform(controllerId)
success resumePlatform(controllerId)
platformState getPlatformInfo(controllerId)
success deletePlatform(controllerId)

Table A.3: Applications Manager API

The Applications Manager has the following interface, as illustrated in Table A.3:

- *createPlatform*: this function creates a new virtual platform from the specified template. The Applications Manager initiates the deployment of an application controller and delegates the virtual cluster and application management to it.
- *submitApplication*: this function creates a new virtual platform from an existing virtual platform template. For example, a virtual platform template for static MPI applications can be re-used for different MPI applications by changing the application input files, execution commands and the SLO value. This function receives a higher-level template, containing the previously mentioned information together with an existing virtual platform template. It basically perfoms the same steps as the previous call but it also fills the existing virtual platform template with the high level application description information.
- *editPlatform*: this function allows editing of user defined SLOs, policy parameters or budget. These parameters are sent to the application controller that will perform the actions specified in the user-defined policies.
- *suspendPlatform* and *resumePlatform*: these functions suspend and, respectively resume all the virtual clusters composing this virtual platform.
- *deletePlatform*: this function deletes the virtual platform. Basically the Applications Manager sends a message to the application controller to shut down the application.
- *getPlatformInfo*: this function lists all existing virtual platforms from the infrastructure.

Users interact with the application management system through the *Merkat_platform* client, which provides user access to all the previously mentioned functionalities.

Appendix B

Résumé en français

Les organisations qui possèdent des infrastructures de calcul à haute performance (HPC) font souvent face à certaines difficultés dans la gestion de leurs ressources. En particulier, ces difficultés peuvent provenir du fait que des applications de différents types doivent pouvoir accéder concurremment aux ressources tandis que les utilisateurs peuvent avoir des objectifs de performance variés pour leurs applications. Les nuages informatiques (*Clouds*) apportent plus de flexibilité et un meilleur contrôle des ressources qui laissent espérer une amélioration de la satisfaction des utilisateurs en terme de qualité de service perçue. Cependant, les solutions de nuage informatique actuelles fournissent un support limité aux utilisateurs pour l'expression ou l'utilisation de politiques de gestion de ressources et elles n'offrent aucun support pour atteindre les objectifs de performance et de temps de restitution des applications. Dans cette thèse nous proposons de relever ce défi par la conception d'une plate-forme de nuage informatique souple, efficace et équitable.

B.1 Motivation

La motivation principale de cette thèse s'illustre par le cas d'une organisation effectant du calcul à haute performance, telle que Electricité de France (EDF). En tant que premier producteur mondial d'électricité, EDF mise sur la simulation numérique sur des ressources de calcul à haute performance depuis plus de 30 ans. Les résultats de simulation sont, notamment, utilisés pour optimiser la production au jour le jour, ou pour déterminer les configurations les plus sûres et les plus efficaces pour le rechargement des combustibles nucléaires. A plus long terme, d'autres résultats s'avèrent très précieux pour expliquer des phénomènes physiques complexes, afin de mieux planifier les opérations de maintenance ou pour évaluer l'impact des modifications éventuelles ou des nouvelles technologies de fournisseurs. Ces simulations numériques sont réalisées dans une grande variété de domaines. Entre autres, des études sont menées en neutronique, en dynamique des fluides, en mécanique ou en thermodynamique. Une partie se compose d'applications MPI qui sont constituées d'un ensemble fixe de processus, tandis que d'autres sont composées de milliers de tâches indépendantes, exécutant le même code avec des paramètres différents. En outre, pour prévoir la demande en énergie des utilisateurs pour la période suivante, un grand nombre de données doit être analysé. Ces études sont réalisées avec des environments spécifiques (par exemple, Hadoop) qui utilisent leurs propres mécanismes d'ordonnancement. Dans ce contexte, comme ces études doivent être réalisées dans un contexte sécurisé, EDF s'astreint à n'utiliser que ses propres grappes de calcul.

Au fur et à mesure de l'évolution des modèles de programmation et des besoins des utilisateurs, les applications développées ont également des demandes de ressources de plus

en plus dynamiques. Prenons l'exemple d'une simulation menée en dynamique des fluides, souvent utilisée pour analyser l'écoulement d'un fluide dans certaines configurations. Lorsque certaines conditions sont satisfaites, un écoulement turbulent fait son apparition. Pour le simuler, cela demande la résolution d'un ensemble supplémentaire d'équations que l'on ajoute au modèle initial. Dès cet instant, le calcul est plus consommateur en temps CPU et l'application bénéficierait de la mobilisation de plus de ressources pour améliorer sa vitesse de calcul. D'autres applications peuvent également profiter de la disponibilité variable des ressources. Par exemple, les applications de type groupe de tâches [40] peuvent utiliser l'ensemble des ressources disponibles lorsque l'infrastructure est sous-utilisée, et restituer une partie de ces ressources quand la demande des autres applications augmente.

En dehors du caractère dynamique de la demande de ressources par les applications, un autre problème tient à l'exigence des utilisateurs en termes d'objectifs de performance ou de facilité de configuration, de déploiement tout en conservant toute la disponibilité possible de leurs environnements logiciels. Certains d'entre eux désirent obtenir les résultats de l'application en un délai très précis (par exemple, un utilisateur doit envoyer a son responsable hierarchique les résultats d'une simulation a sept heures du matin le lendemain) ou dès que possible (par exemple, un développeur veut tester un nouvel algorithme). D'autres veulent exprimer des politiques d'exécution simples, comme: "*Je désire arrêter mon application lorsqu'une certaine condition de performance est satisfaite.*", ou "*J'ai besoin d'ajouter un module de visualisation a mon application et j'ai besoin de plus de ressources maintenant*".

En outre, les études réalisées par EDF peuvent nécessiter l'installation de divers logiciels. Incombe alors à l'administrateur de l'infrastructure la difficile tâche de gérer la configuration logicielle pour chacun sur l'ensemble des nœuds physiques.

Même si la virtualisation de l'infrastructure et la transformation en un nuage privé peuvent améliorer la gestion des ressources de l'infrastructure, elles ne couvrent pas entièrement les exigences mentionnées précédemment. Bien que les plates-formes logicielles développées récemment offrent aux utilisateurs une interface pour développer et exécuter leurs applications sans aucune préoccupation concernant la complexité de gestion des ressources, elles sont encore limitées. Plus précisément, elles offrent un support limité d'expression ou d'utilisation de politiques de gestion des ressources et elles ne fournissent aucun outil pour atteindre des objectifs de performance.

Enfin, comme la capacité de l'infrastructure est limitée, le véritable défi réside dans la façon de partager les ressources entre les applications des utilisateurs. Cela ne serait pas un problème si la capacité de l'infrastructure était assez importante pour couvrir l'ensemble des demandes des utilisateurs dans les périodes de plus forte demande. Cependant, c'est rarement le cas, car l'élargissement de la grappe de calcul est coûteux. Ainsi, il est préférable de résoudre les périodes de contention au cas par cas, lorsqu'elles apparaissent à des périodes ponctuelles, (par exemple, une conférence ou un livrable de projet), plutôt que d'acquérir plus de ressources pour satisfaire cette augmentation de la demande. Cependant, il n'y a pas de mécanismes sociaux pour que les utilisateurs adaptent la demande de ressources de leurs applications dans des périodes de contention, d'une manière qui profite à tous les utilisateurs de l'infrastructure.

B.2 Objectifs

Étant donné la complexité croissante des applications, la dynamicité des demande de ressources et la variété des besoins des utilisateurs, les objectifs de cette thèse sont de concevoir, de mettre en œuvre et d'évaluer une plate-forme qui alloue des ressources en-

tre les applications d'une manière flexible, efficace et équitable. Cette plate-forme doit satisfaire trois besoins:

Mécanismes flexibles d'adaptation de l'application: Les utilisateurs d'une organisation executant des applications scientifiques ont des exigences variées en terme de types d'applications qui doivent être exécutées sur l'infrastructure et de garantie de performance ou de droit d'accès qui doivent leur être associés. Pour répondre à toutes ces exigences avec un effort minimum de l'utilisateur, il faut concevoir des mécanismes flexibles qui adaptent automatiquement les applications et leurs demandes de ressources à des objectifs définis par l'utilisateur. Ces mécanismes flexibles doivent avoir quatre caractéristiques: (i) ils doivent permettre aux applications de s'exécuter dans des environnements isolés et personnalisés en fonction de la configuration logicielle requise; (ii) ils doivent permettre aux applications de modifier leurs allocations de ressources dynamiquement; (iii) ils doivent donner aux applications la possibilité de prendre des décisions concernant la quantite des ressources allouées; (iv) ils doivent également offrir des garanties de performance aux applications, simplement parce qu'exécuter des applications sur les ressources acquises ne garantit pas qu'elles vont atteindre leurs objectifs de performance, en raison de facteurs imprévisibles, comme par exemple, des interférences impactant les performances, des nœuds lents ou des défaillances.

Répartition efficace des ressources: En fournissant un accès concurrent aux ressources, il existe deux façons de partager les ressources: (i) à grain grossier, et (ii) à grain fin. Dans le cas d'allocation de ressources à grain grossier, soit les applications ont l'accès exclusif aux nœuds physiques, soit elles s'exécutent dans des machines virtuelles avec une configuration choisie à partir d'une classe spécifique, comme pour Amazon EC2 [7]. Cette méthode d'allocation des ressources conduit à une mauvaise utilisation des ressources : tandis que certaines applications n'utilisent pas toutes leurs ressources provisionnées, d'autres applications peuvent attendre dans les files d'attente. Avec l'allocation des ressources à grain fin, les applications peuvent demander des quantités arbitraires de ressources et l'utilisation de l'infrastructure est amélioré. Ainsi, des mécanismes doivent être conçus pour fournir l'allocation des ressources à grain fin.

Utilisation équitable des ressources: Généralement, une organisation dispose d'une infrastructure qui doit être partagée entre les utilisateurs venant de différents départements et ayant des préférences différentes concernant l'exécution de leurs applications. Pour l'administrateur, la réglementation de l'accès aux ressources de ces utilisateurs peut être une tâche complexe: il doit décider des critères utilisés pour accorder un accès aux ressources le plus équitable possible pour les utilisateurs. Un critère serait d'assigner des priorités aux utilisateurs. Même si l'administrateur décide des priorités des utilisateurs, il ne peut pas décider quelle application est la plus urgente à exécuter à la place de l'utilisateur et quand elle doit être exécutée. Par exemple, dans un centre de calcul, l'infrastructure peut connaître des pics de charge à l'approche de conférences importantes, car les chercheurs peuvent vouloir utiliser les ressources pour obtenir aussi rapidement que possible les résultats de leurs simulations. Dans ce cas, un utilisateur avec une priorité élevée pourrait abuser de ses droits et d'exécuter une simulation moins urgente, en prenant des ressources aux utilisateurs qui en ont vraiment besoin. Nous ne pouvons pas supposer que tous les utilisateurs sont coopératifs et qu'ils libèrent spontanément l'accès aux ressources que demandent d'autres utilisateurs. Ainsi, des mécanismes doivent être conçus pour assurer une répartition

équitable en affectant des ressources aux utilisateurs qui sauront les apprécier à leur juste valeur.

B.3 Contributions

Pour atteindre ces objectifs, cette thèse propose un cadre générique et extensible pour la gestion autonome des applications et l'allocation dynamique des ressources. Nous avons conçu ce cadre pour des infrastructures partagées privées. Il offre un contrôle des ressources entièrement décentralisé en les allouant suivant un grain fin en se fondant sur un marché à pourcentage proportionnel. Le contrôle de ressources décentralisé permet aux applications de s'exécuter dans des environnements virtuels autonomes, capables de faire évoluer la demande en ressources de l'application en fonction des objectifs de performance de l'utilisateur. Le marché fixe dynamiquement un prix aux ressources, ce qui, combiné avec une politique de distribution de monnaie entre les utilisateurs, en garantit une utilisation équitable.

B.3.1 Conception

Notre système utilise un nuage informatique de type IaaS pour gérer les nœuds de l'infrastructure privée de l'organisation. Le nuage informatique de type IaaS fournit aux utilisateurs des environnements logiciels isolés et contrôlés, leur permettant d'installer toutes les dépendances logicielles requises pour leurs applications sans l'intervention de l'administrateur. Du point de vue de l'architecture, nous positionnons notre système au-dessus de la couche de nuages de type IaaS, car il offre une solution PaaS pour les applications scientifiques.

La conception de notre système repose sur quatre principes:

Des plates-formes virtuelles privées et autonomes: Pour supporter différents types d'applications et des objectifs définis par l'utilisateur, chaque application s'exécute dans son propre environnement virtuel, appelé plate-forme virtuelle. Une plate-forme virtuelle est une entité autonome composée d'une ou plusieurs grappes de ressources virtuelles et d'un contrôleur qui gère les ressources au nom de l'application. Une grappe de ressources virtuelles est un groupe de machines virtuelles qui ont la même configuration et sont démarrées à partir du même disque de base. Pour atteindre les objectifs définis par l'utilisateur, le contrôleur interagit avec l'application, adapte dynamiquement la demande de ressources à la charge de l'infrastructure et aux performances de l'application quitte à reconfigurer l'application. Chaque grappe des ressources virtuelles a un moniteur qui envoie les métriques de performance de l'application au contrôleur. Les utilisateurs ont la possibilité d'automatiser et de personnaliser le déploiement et la gestion de leurs applications en écrivant des contrôleurs d'applications. Ces contrôleurs peuvent utiliser des politiques sophistiqués, par exemple, de prédiction, de profilage et des modules d'apprentissage pour déterminer l'allocation de ressources dont l'application a besoin pour son exécution et utiliser cette information pour modifier le nombre de machines virtuelles provisionnées.

Des mécanismes fondés sur un marché pour partager l'accès aux ressources: La principale caractéristique de notre système est l'utilisation d'un mécanisme fondé sur un marché pour fournir des machines virtuelles et réguler la quantité de ressources que les applications peuvent obtenir. Les approches fondées sur un marché ont été appliquées dans une variété de contextes dans les systèmes distribués

tels que grappes, des grilles, des réseaux pairs à pair et des nuages informatiques. Chaque utilisateur reçoit un certain montant en devises, qui reflète la priorité qu'il a dans le système. Le utilisateur peut distribuer ce montant entre ses applications, ce qui traduit la véritable importance qui il accorde a sa exécution.

Pour exécuter ses applications sur l'infrastructure, l'utilisateur dispose de deux possibilités: (i) soumettre un paiement pour chaque machine virtuelle, et démarrer manuellement son application; (ii) estimer le budget dont l'application a besoin pour s'exécuter et démarrer une plate-forme virtuelle. Le contrôleur de la plate-forme virtuelle gère au nom de l'utilisateur le coût total d'exécution et la soumission des paiements pour les machines virtuelles au cours de l'exécution de l'application.

L'utilisation d'un marché présente deux avantages. Premièrement, elle permet aux utilisateurs de prendre de meilleures décisions concernant le volume de ressources que leur application utilise, conduisant à une amélioration de la satisfaction globale des utilisateurs. Deuxièmement, le marché prévoit un mécanisme de régulation décentralisée des ressources qui permet à chaque application de contrôler sa propre allocation de ressources en fonction de la priorité assignée par l'utilisateur. Des politiques d'adaptation de la demande de ressources peuvent être conçues en utilisant le prix comme un signal alors que les politiques de distribution de la monnaie virtuelle assurent une adaptation équitable.

L'allocation des ressources à grain fin: Pour assurer une allocation des ressources à grain fin avec une faible complexité, notre système utilise une politique de type pourcentage proportionnel. Cette politique est également appliquée par l'ordonaceurs du système d'exploitation pour allouer du temps de processeur aux tâches proportionnellement aux priorités données aux utilisateurs. Dans notre système, cette politique est utilisée pour l'allocation de mémoire et du temps de processeur aux machines virtuelles.

En utilisant cette politique, les machines virtuelles peuvent recevoir une partie de la capacité de l'infrastructure proportionnelle à leurs paiements et inversement proportionnelle à la somme de tous les paiements en cours. Ces allocations de ressources sont fractionnaires et dynamiques car elles varient dans le temps en fonction de la demande totale d'infrastructure. La migration à chaud des machines virtuelles est utilisée pour équilibrer la charge entre les nœuds et pour s'assurer que chaque machine virtuelle reçoit une part de ressources proportionnelle au montant payé à cet effet.

L'extensibilité et la facilité d'utilisation: Nous avons conçu notre système pour qu'il soit suffisamment modulaire pour permettre aux développeurs de mettre en œuvre d'autres mécanismes de régulation de la demande de ressources avec un minimum de modifications. Pour faciliter l'execution d'une application et les changements de demande de ressources, notre système offre deux politiques d'adaptation prédéfinies qui tiennent compte d'un objectif de performance et d'un budget défini par l'utilisateur et adaptent la demande de ressources à leur prix observé sur le marché: (i) une politique qui adapte la demande de ressources pour chaque machine virtuelle (élasticité verticale); (ii) une politique qui adapte le nombre de machines virtuelles (élasticité horizontale). Ces politiques peuvent être combinées pour améliorer les performances des applications. Par ailleurs, une API simple et générique permet aux utilisateurs de concevoir leurs propres contrôleurs avec des politiques différentes de celles fournies.

B.3.2 Mise en œuvre et évaluation

Nous avons mis en œuvre et évalué notre proposition dans deux contextes:

Simulation Nous avons simulé notre système en Java, en utilisant CloudSim [27]. Le prototype résultant s'appelle Themis. Nous avons utilisé Themis pour évaluer la performance du marché fondé sur un pourcentage proportionnel et la politique de élasticite verticale vis à vis de divers objectifs de performance des applications.

Nous avons modélisé trois types d'utilisateurs, en fonction des types de résultats qu'ils attendent de leurs applications (résultats potentiellement partiels ou obligatoirement complets), et suivant diverses échéances (le plus tôt possible ou obligatoirement avant une date fixée d'avance). Nous avons attribué à chaque utilisateur une fonction d'utilité exprimant la satisfaction qu il obtient après l'exécution de son application. Nous avons mesuré la satisfaction totale que le système peut fournir à chaque type d'utilisateur, le nombre d'opérations effectuées par les machines virtuelles et nous avons analysé le comportement global des applications.

Nos résultats ont montré que: (i) malgré un comportement égoïste non coordonné, comparé à des politiques centralisées de gestion de ressources, notre système peut atteindre une performance raisonnable, mesurée en termes de satisfaction des utilisateurs; (ii) étant donné l'ampleur du système, le nombre d'opérations effectuées par les machines virtuelles reste dans des limites raisonnables.

Déploiement sur une infrastructure distribuée Nous avons implémenté un prototype, appelé Merkat. Dans ce prototype, nous avons mis en œuvre deux contrôleurs: (i) un contrôleur pour les applications statiques de type MPI; (ii) et un contrôleur des applications malléables. Nous avons déployé Merkat sur l'infrastructure Grid'5000 et nous l'avons testé avec plusieurs applications scientifiques representative pour EDF.

Nos résultats ont montré que: (i) Merkat permet une utilisation efficace des ressources en partageant les ressources entre les applications à grain fin; (ii) Merkat est flexible en permettant aux utilisateurs de spécifier différents objectifs de performance pour différents types d'applications; (iii) Merkat fournit une différenciation équitable entre les utilisateurs, conduisant à une meilleure satisfaction des utilisateurs que la satisfaction obtenue avec les systèmes traditionnels de gestion de ressources.

B.4 Organisation du document

Ce document est organisé de la façon suivante. Le chapitre 1 décrit les modèles principaux d'application et de l'organisation des systèmes distribués. Le chapitre 2 couvre l'état de l'art. Nous présentons les principales approches pour la gestion des ressources qui pourraient répondre à autaines de exigences. Le chapitre 3 donne un bref aperçu de notre contribution. Nous présentons notre approche et expliquons le principle de conception de systéme proposé. Le chapitre 4 présente les mécanismes de gestion des ressources mises en œuvre dans notre plate-forme. Nous décrivons comment les applications peuvent adapter dynamiquement leur demande de ressources et comment les ressources sont gérées sur l'infrastructure. Le chapitre 5 décrit les résultats des simulations concernant la performance dans des systèmes de grande taille. Le chapitre 6 traite l'applicabilité de notre solution dans un environnement réel. Nous concluons par un chapitre dressant un bilan de nos contributions et présentant des directions possibles de travaux de recherche ultérieurs.

Market-based Autonomous Application and Resource Management in the Cloud

Organizations owning HPC infrastructures are facing difficulties in managing their infrastructures. These difficulties come from the need to provide concurrent resource access to applications with different resource requirements while considering that users might have different performance objectives, or Service Level Objectives (SLOs) for executing them. To address these challenges this thesis proposes a market-based SLO-driven cloud platform. This platform relies on a market-based model to allocate resources to applications while taking advantage of cloud flexibility to maximize resource utilization. The combination of currency distribution and dynamic resource pricing ensures fair resource distribution. In the same time, autonomous controllers apply adaptation policies to scale the application resource demand according to user SLOs. The adaptation policies can: (i) dynamically tune the amount of CPU and memory provisioned for the virtual machines in contention periods; (ii) dynamically change the number of virtual machines. We evaluated this proposed platform in simulation and on the Grid'5000 testbed. Results show that: (i) the platform provides flexible support for different application types and different SLOs; (ii) the platform is capable to provide good user satisfaction achieving acceptable performance degradation compared to existing centralized solutions.

Keywords: cloud computing, resource management, autonomous systems

Gestion autonome des ressources et des applications dans un nuage informatique selon une approche fondée sur un marché

Les organisations qui possèdent des infrastructures de calcul à haute performance (HPC) font souvent face à certaines difficultés dans la gestion de leurs ressources. En particulier, ces difficultés peuvent provenir du fait que des applications de différents types doivent pouvoir accéder concurremment aux ressources tandis que les utilisateurs peuvent avoir des objectifs de performance (SLOs) variés. Pour atteindre ces difficultés, cette thèse propose un cadre générique et extensible pour la gestion autonome des applications et l'allocation dynamique des ressources. L'allocation des ressources et l'exécution des applications est régie par une économie de marché observant au mieux des objectifs de niveau de service (SLO) tout en tirant avantage de la flexibilité d'une nuage informatique et en maximisant l'utilisation de des ressources. Le marché fixe dynamiquement un prix aux ressources, ce qui, combiné avec une politique de distribution de monnaie entre les utilisateurs, en garantit une utilisation équitable. Simultanément, des contrôleurs autonomes mettent en œuvre des politiques d'adaptation pour faire évoluer la demande en ressource de leur application en accord avec la SLO requise par l'utilisateur. Les politiques d'adaptation peuvent : (i) adapter dynamiquement leur demande en terme de CPU et de mémoire demandés en période de contention de ressource aux machines virtuelles (ii) et changer dynamiquement le nombre de machines virtuelle. Nous avons évalué cette plateforme au moyen de la simulation et sur l'infrastructure Grid'5000. Nos résultats ont montré que cette solution: (i) offre un support plus flexible aux applications de type différent demandant divers niveaux de service; (ii) conduit à une bonne satisfaction des utilisateurs moyennant une dégradation acceptable des performances comparées aux solutions centralisées existantes.

Mots clés: nuages informatiques, allocation de ressources, systemes autonomes