Scheduling and deployment of large-scale applications on Cloud platforms

Adrian Muresan

N° d'ordre : XXX
N° attribué par la bibliothèque : XXXXXXXXXXXXXX

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur
de l'Université de Lyon – École Normale Supérieure de Lyon

Spécialité : Informatique

au titre de l'École Doctorale de Mathématiques et d'Informatique Fondamentale

présentée et soutenue publiquement le 10 décembre 2012 par

M. Adrian Mureșan

Scheduling and deployment of large-scale applications on Cloud platforms

Directeurs de thèse : Frédéric Desprez
Eddy Caron

Après avis de : Stéphane Genaud
Jean-François Méhaut

Devant la commission d'examen formée de :

Eddy Caron	Membre
Frédéric Desprez	Membre
Stéphane Genaud	Membre/Rapporteur
Jean-François Méhaut	Membre/Rapporteur
Pierre Sens	Membre
Matthew Wood	Membre

Abstract

Infrastructure as a service (IaaS) Cloud platforms are increasingly used in the IT industry. IaaS platforms are providers of virtual resources from a catalog of predefined types. Improvements in virtualization technology make it possible to create and destroy virtual machines on the fly, with a low overhead. As a result, the great benefit of IaaS platforms is the ability to scale a virtual platform on the fly, while only paying for the used resources.

From a research point of view, IaaS platforms raise new questions in terms of making efficient virtual platform scaling decisions and then efficiently scheduling applications on dynamic platforms. The current thesis is a step forward towards exploring and answering these questions.

The first contribution of the current work is focused on resource management. We have worked on the topic of automatically scaling cloud client applications to meet changing platform usage. There have been various studies showing self-similarities in web platform traffic which implies the existence of usage patterns that may or may not be periodical. We have developed an automatic platform scaling strategy that predicted platform usage by identifying non-periodic usage patterns and extrapolating future platform usage based on them.

Next we have focused on extending an existing grid platform with on-demand resources from an IaaS platform. We have developed an extension to the DIET (Distributed Interactive Engineering Toolkit) middleware, that uses a virtual market based approach to perform resource allocation. Each user is given a sum of virtual currency that he will use for running his tasks. This mechanism help in ensuring fair platform sharing between users.

The third and final contribution targets application management for IaaS platforms. We have studied and developed an allocation strategy for budget-constrained workflow applications that target IaaS Cloud platforms. The workflow abstraction is very common amongst scientific applications. It is easy to find examples in any field from bioinformatics to geology. In this work we

have considered a general model of workflow applications that comprise parallel tasks and permit non-deterministic transitions. We have elaborated two budget-constrained allocation strategies for this type of workflow. The problem is a bi-criteria optimization problem as we are optimizing both budget and workflow makespan.

This work has been practically validated by implementing it on top of the Nimbus open source cloud platform and the DIET MADAG workflow engine. This is being tested with a cosmological simulation workflow application called RAMSES. This is a parallel MPI application that, as part of this work, has been ported for execution on dynamic virtual platforms. Both theoretical simulations and practical experiments have shown encouraging results and improvements.

Résumé

L'usage des plateformes de Cloud Computing offrant une Infrastructure en tant que service (IaaS) a augmenté au sein de l'industrie. Les infrastructures IaaS fournissent des ressources virtuelles depuis un catalogue de types prédéfinis. Les avancées dans le domaine de la virtualisation rendent possible la création et la destruction de machines virtuelles au fur et à mesure, avec un faible surcout d'exploitation. En conséquence, le bénéfice offert par les plateformes IaaS est la possibilité de dimensionner une architecture virtuelle au fur et à mesure de l'utilisation, et de payer uniquement les ressources utilisées.

D'un point de vue scientifique, les plateformes IaaS soulèvent de nouvelles questions concernant l'efficacité des décisions prises en terme de passage à l'échelle, et également l'ordonnancement des applications sur les plateformes dynamiques. Les travaux de cette thèse explorent ce thème et proposent des solutions à ces deux problématiques.

La première contribution décrite dans cette thèse concerne la gestion des ressources. Nous avons travaillé sur le redimensionnement automatique des applications clientes de Cloud afin de modéliser les variations d'utilisation de la plateforme. De nombreuses études ont montré des autosimilarités dans le trafic web des plateformes, ce qui implique l'existence de motifs répétitifs pouvant être périodiques ou non. Nous avons développé une stratégie automatique de dimensionnement, capable de prédire le temps d'utilisation de la plateforme en identifiant les motifs répétitifs non périodiques.

Dans un second temps, nous avons proposé d'étendre les fonctionnalités d'un intergiciel de grilles, en implémentant une utilisation des ressources à la demandes. Nous avons développé une extension pour l'intergiciel DIET (Distributed Interactive Engineering Toolkit), qui utilise un marché virtuel pour gérer l'allocation des ressources. Chaque utilisateur se voit attribué un montant de monnaie virtuelle qu'il utilisera pour exécuter ses tâches. Le mécanisme d'aide assure un partage équitable des ressources de la plateforme entre les différents utilisateurs.

La troisième et dernière contribution vise la gestion d'applications pour les plateformes IaaS. Nous avons étudié et développé une stratégie d'allocation des ressources pour les applications de type workflow avec des contraintes budgétaires. L'abstraction des applications de type workflow est très fréquente au sein des applications scientifiques, dans des domaines variés allant de la géologie à la bioinformatique. Dans ces travaux, nous avons considéré un modèle général d'applications de type workflow qui contient des tâches parallèles et permet des transitions non déterministes. Nous avons élaboré deux stratégies d'allocations à contraintes budgétaires pour ce type d'applications. Le problème est une optimisation à deux critères dans la mesure où nous optimisons le budget et le temps total du flux d'opérations.

Ces travaux ont été validés de façon expérimentale par leurs implémentations au sein de la plateforme de Cloud libre Nimbus et de moteur de workflow MADAG présent au sein de DIET. Les tests ont été effectuées sur une simulation de cosmologie appelée RAMSES. RAMSES est une application parallèle qui, dans le cadre de ces travaux, a été portée sur des plateformes virtuelles dynamiques. L'ensemble des résultats théoriques et pratiques ont débouché sur des résultats encourageants et des améliorations.

Contents

Chapter 1

Introduction

The work in this Thesis aims at improving resource provisioning and allocation in Cloud platforms. First we consider the point of view of the Cloud provider and in this sense study the state of the art and its current problems. We propose resource provisioning solutions that improve on the state of the art.

Next we concentrate on the point of view of the Cloud client. We propose solutions for automatic virtual platform management in the general case and in a specialized application case.

The current chapter presents the motivation, problematics and objectives of the current Thesis, while offering a first introduction in the context of the current domain.

Motivation of the current work

The idea of computing as a utility is by no means a new one. One of the first references to this is given by John McCarthy in 1961 where he described the possibility of having computational power as a utility in the future, similar to the telephone system.

As technology evolved, this idea became a reality. Realizing this was done in two step: a hardware step and a software step. The first step was the proliferation of large data centers formed of computational clusters, each consisting of a large number of powerful computers interconnected by a network, all of them located in a close proximity to each other. Compute clusters were interconnected by fast networks, thus becoming a bigger structure capable of sharing resources from various clusters, regardless of geographic position. The result is a large, heterogeneous platform called a computational grid, which

is an analogy to the power grid. Computational grids were born in scientific communities and were primarily used as a mechanism for researchers to share their resources. They allow large simulations to be made that break geographic boundaries.

Computational grids are shared environments and, as they come from the scientific communities, internal security is usually low, yet internal security problems rarely occur. Politeness and respect for the other members of the community are usually enough for security enforcement. The users of the grid are known, as well as their identities which means the environment is transparent.

Given that compute grids are formed of heterogeneous resources, developing applications for them is a challenging task. The application developers have to take into consideration topics as: ways in which resources can be accessed, job scheduling, job and resource monitoring, data management, security and others. All of these features can be abstracted away by a *grid middleware*, which permits a more straightforward approach to application development. As examples of toolkits that permit development of grid services we can consider: gLite [1] developed as part of the EGEE project, the Globus Toolkit [2] developed by the Globus Alliance, UNICORE [3] sponsored by the German ministry for education and research and later by several European projects.

Another approach of providing transparent access to heterogeneous resources is the *Remote Procedure Call* (RPC) approach which permits invoking a method of an object regardless where the object is located. The RPC paradigm was extended into the context of grid computing by the Open Grid Forum with the GridRPC [4] standard. The standard insures interoperability of grid middleware, meaning that any applications that are built on the GridRPC paradigm are compatible with any GridRPC compliant middleware. Examples of GridRPC middleware are GridSolve [5] which is one of the first GridRPC compliant middleware, Ninf [6] with a similar functionality and Ninf-G which is developed with the Globus Toolkit. Another example is DIET [7, 8] (Distributed Interactive Engineering Toolkit). Throughout this Thesis, DIET is used for practical validation either as an experimental testbed or by extending it with prototypes that underline the work in this Thesis. Whereas Ninf and NetSolve use a single agent, DIET uses a hierarchy of agents. This allows DIET installations to follow the grid topology closely and distributes the scheduling decisions across the hierarchy of agents. This architecture makes DIET a highly scalable middleware. More details about the DIET architecture and its extensions will be presented throughout this Thesis.

In contrast to scientific research grids, a commercial computational grid has different requirements. One of the most important being guaranteeing isolation and security of the environment since commercial grids cannot be transparent to the same level as research grids are. Fair resource sharing is also a problem, as commercial users will want a guarantee that they will have the resources that they are paying for. There are successful commercial

applications of the computational grid, usually based on isolation techniques either by using Operating System mechanisms like processes or more advanced mechanisms as virtualization like the Java Virtual Machine (JVM) or the Java Multi-user Virtual Machine (MVM). A major step forward in this direction was the development of virtualization technology that allowed the virtualization of an entire compute machine inside a host physical machine, effectively allowing the installation of multiple operating systems on the same physical host and thus providing a high level of isolation between the virtual machines themselves and the underlying host machine. This technology has evolved over time from full software virtualization to paravirtualization, a technique through which slow operations running inside the Virtual Machine (VM) can be sped up with special instructions implemented by the CPU. The implications of this approach are that the difference in performance between an application running inside a VM and outside, as a first class application, have become considerably small.

This was the second step towards computation power as a utility: the software step. It opened the possibility of migrating computational needs to the big data centers while guaranteeing the same level of quality of service. Obtaining more computational power or releasing unneeded one became as simple as flipping a switch. The providers of this type of service are called Cloud providers, the term "Cloud" being an analogy to the fact that the end users do not have physical access to the resources they use, they use them through the Internet.

Cloud computing is one of the fields where practice developed before theory and as a result there was no standard way of working with a Cloud provider, which made vendor lock-in a possibility for Cloud clients. Consequently, there are efforts oriented towards the possibility of treating all Cloud platforms in the same way and abstracting away their interface. Interconnecting several Cloud providers together gives birth to Cloud federations or "Sky Computing".

The most typical use-case for Cloud platforms is to automatically scale a virtual platform based on its usage. This leads to lower costs for the platform owner, as he only pays for resources that he uses and the resources he uses follow closely the usage of the platform. When contrasted to the traditional approach of allocating a fixed number of resources, this is clearly a better approach. Until the appearance of Cloud-like platforms this was not possible in the general application case. Although automatic scaling comes with great advantages, it has its share of questions and problems. The most important of these are related to automatic platform scaling decisions in the general case as well as in the case of a specific application type. Making a good scaling decision is not trivial.

Objectives and contributions of the current Thesis

The context of the current work is the field of virtual resource provisioning using Cloud platforms. Resource provisioning for Cloud platforms can be seen from two different points of view: the Cloud provider's and the Cloud client's point of view. The current work has contributions to both.

From the Cloud client's point of view, scaling his virtual platform is the main concern. We have proposed a new automatic scaling algorithm for Cloud client applications that tries to identify usage patterns in the platform's usage history. The identified patterns help in making the scaling decision. The algorithm that we propose is able to identify nonperiodic repetitive behavior, which has been documented in web traffic and is inherent to some application types and some usage scenarios. As such, the proposed algorithm can prove effective for some applications and scenarios, but not for others.

We have considered a specialized application pattern, called workflow, that is very common in scientific communities. We have proposed two algorithms for resource allocation on dynamic platforms. Both algorithms determine allocations that minimize the total running time of the application while keeping a fixed budget limit.

From the point of view of Cloud providers we have studied the problem of resource provisioning and have contributed with the design of a mechanism that extends a grid platform with Cloud resources from a resource catalog. This mechanism was designed for the DIET toolkit (Distributed Interactive Engineering Toolkit), an open-source grid middleware. In order to achieve resource sharing we have used a mechanism based on commodity markets where each user has a finite amount of virtual currency that is recirculated into the system and computational power is the commodity that is being traded. In this system there are computational resource providers and resource users. The virtual currency flows naturally from user to producer and back again by a recirculation mechanism. Resource prices are left to fluctuate freely in this closed system and fair resource sharing comes as a side effect.

Organization of the current Thesis

This Thesis is organized as follows. Chapter 2 presents an introduction into the field of Cloud computing and a study of the state of the art. We focus on three important Cloud features: automatic scaling, load balancing and platform monitoring.

In Chapter 3 we present an automatic scaling approach for Cloud clients. This approach is based on identifying similar existing load patterns in the history of the platform's usage and using them to extrapolate what the future platform usage can be. This approach is highly effective for applications that have a nonperiodic repetitive behavior. To validate our approach we have

tested the algorithm against platform traces from one Cloud client application and several grid workload traces.

Chapter 4 presents our proposal for extending a grid middleware with Cloud resources based on economic mechanisms. Users are given a finite quantity of virtual currency that they spend to execute their tasks on virtual resources. Virtual resource prices are left to fluctuate and tend to self-stabilize, thus guaranteeing fair resource sharing amongst users.

In Chapter 5 we explore the problem of determining resource allocations for a specialized class of applications, the workflow class as this is a very common application class in the scientific world. Since Cloud platforms are advertised as having an unbound number of resources in their pool, the traditional scheduling problem becomes a bi-criteria optimization problem where we must consider both running time and cost as criteria. We propose two algorithms that determine budget-constrained allocations. For validation we check and compare their performance against synthetic application traces.

Finally in Chapter 6 we conclude this Thesis while presenting a summary of the contributions and future perspectives.

Part 1: Context

Chapter 2

Resource management in Cloud platforms

The current chapter has the purpose of introducing the reader into the topic of the current Thesis. It presents an introduction of the Cloud computing paradigm and then focuses on the current state of the art in resource management for Cloud platforms, highlighting auto-scaling, load balancing and monitoring as they appear in commercial and open-source Cloud platforms. At the end it also presents further directions of improvement of these topics.

2.1 Steps towards the Cloud

Grid Computing

As Internet connection bandwidth increased and more types of distributed architectures became possible. In order to facilitate resource sharing and provide access to large computational power, a distributed and heterogeneous architecture is required. Foster and Kesselman invented the term *Computational Grid* [9] to refer to such systems. The term is an analogy to the power grid, but instead of electricity, computational grids provide computational power.

In their definition, a computational grid is an entity, owned by multiple organizations and so it does not have a single controlling entity. In Foster's vision, the grid is decentralized hardware and software infrastructure, based on open protocols that delivers nontrivial services and qualities of service. Each of the owning organizations has transparent access to the grid's resource, regardless of geographic positioning.

In scientific communities involved in *High Performance Computing* (HPC)

a research grid is typically a large network that connects multiple compute clusters [10], giving shared resource access to membering institutions. Resource access is done in a transparent way, even if the membering clusters have different software installations

Cloud Computing

The idea of computational power served as a commodity has continued to evolve beyond the initial computational grid concept. The development of virtualization technology has opened new doors when it comes to computational resource hosting. An organization can give its clients virtual resources which are a very flexible in terms of virtual hardware and software specifications. This leads to a step forward in computational resource sharing.

The term *Cloud computing* has a wide range of definitions [11] and types. In what follows we will explore the most relevant of them.

IaaS (Infrastructure as a Service)

IaaS Cloud platforms offer a virtualized infrastructure to their clients by means of *virtual machines* (VM), an abstraction of a physical machine, that have predefined characteristics from a resource catalog. The Cloud platform deploys new virtual machines when its clients ask for them and gives the client complete control over them. The client is charged based on the type and number of virtual machines that he has running, each virtual machine having a fixed per-hour cost. Typically when calculating costs, Cloud platforms round up the running time of each running virtual machine to the nearest higher hour.

In the traditional grid environment, a user requests physical resources over period of time called a *resource lease*. Once his request is accepted, the user can access the physical resources and use them as he wills. The resource lease is usually fixed in number of resources and in the lease time. In contrast, in a traditional Cloud environment, a user requests virtual resources over an unbound time period and uses them as he pleases. The user can always acquire new resources if needed an release existing ones if they are unnecessary. The contrast between the two scenario leads to more flexible and efficient ways of using resources. Given that Cloud computing offers virtual resources, the Cloud user can sometimes change the virtual hardware specifications of his running resources.

There are many commercial and open-source providers of this type. For the commercial platforms we can enumerate: Amazon EC2 [12], Rackspace [13], GoGrid [14], Microsoft Azure [15] and others. From the open-source communities, there are several platforms that have been developed, most notably: Eucalyptus [16], Nimbus [17], OpenNebula [18], OpenStack [19] and others.

In the rest of this document the term *Cloud computing* or *Cloud* is used with the meaning of an IaaS Cloud platform, unless otherwise specified.

PaaS (Platform as a Service)

In contrast to IaaS providers, in the case of PaaS, the users receive a development platform that they can access and develop applications for. The PaaS platform handles resource access, client application deployment and client application scaling automatically, with no need of client intervention.

Examples of commercial PaaS platforms are: Google AppEngine [20], Microsoft Azure [15], Force.com [21], Heroku [22] and others. From the open-source communities there are a number of PaaS platforms being developed, most notably: AppScale [23], TyphoonAE [24], OpenShift [25] and others.

SaaS (Software as a Service)

In the case of SaaS, the end users are usually also the Cloud's users. No applications are being developed and deployed on the SaaS Cloud by the Cloud's clients, rather the clients use the services themselves, paying for what they use.

It is obvious that any type of online service fits into this category thus greatly widening the scope of the term "Cloud".

2.2 State of the art in Cloud computing

Over the past years, the Cloud phenomenon had an impressive increase in popularity in both the software industry and research worlds. In the industry this increase is due to the benefits of on-demand provisioning: Cloud clients can ask for resources when their platforms are under heavy load and pay for those resources by the hours. Later, when load decreases, Cloud clients can release unused resources and stop paying for them, therefore following more closely their platform usage, which on the long term means saving money when compared to a traditional static resource allocation that is unchangeable over time, or changeable with a big latency. This led to a wide adoption of Cloud computing in the industry.

In the academic world, the interest in Clouds comes from the new research problems that it brings. Two such problems are determining a good strategy for automatically allocating and deallocating resources to a Cloud client and determining new virtual machine aware scheduling strategies.

Achieving the above mentioned is not trivial and is done by leveraging more direct functionalities that Clouds provide. Three of these key functionalities are automatic scaling, load balancing and monitoring.

The increasing relevance of Cloud computing in the IT world is undeniable. Cloud providers have focused a lot of attention on providing facilities for Cloud clients, that make using the Cloud an easy task. These facilities range from automatic and configurable platform scaling and load balancing services to platform monitoring at different levels of granularity and configurable alert

services. Given that there are no formal standards related to this topic, each Cloud provider has their own interpretation of the problem and their own way of addressing it.

In what follows, we will detail the topics of auto-scaling, load-balancing and monitoring. We will explore both Cloud provider and Cloud client points of view and examine what approaches are taken by the commercial providers and their open-source counterparts. Where an implementation is not available for one of the discussed Clouds, we will present alternative solutions by turning to available commercial services and open-source software. We will also present research work that has been done around the topic of interest.

Among the commercial Cloud platform providers we have focused on Amazon EC2, Microsoft Azure, GoGrid and RackSpace. From the open-source initiatives we have focused on Nimbus, Eucalyptus and OpenNebula. We have also detailed endeavors in the research world that are relevant to the topics of auto-scaling, monitoring and load balancing.

At the end of this chapter, we have also presented current standardization efforts around Cloud computing.

Auto-Scaling - a Cloud feature

Elasticity, or on-demand resource provisioning, is regarded as one of the differentiating features of clouds. In fact, for some authors, it is considered the characteristic that makes clouds something other than "*an outsourced service with a prettier face*" [26]. Cloud users can quickly deploy or release resources as they need them, thus taking benefit of the typical pay-per-use billing model. They avoid potential over-provisioning of resources which implies investment in resources that are not needed. Also, increases on demand can be quickly attended to by asking the cloud for more resources, thus preventing a possible degradation of the perceived service quality.

However, to benefit from elasticity in typical Infrastructure-as-a-Service (IaaS) settings, the cloud user is forced to constantly control the state of the deployed system. This must be done in order to check whether any resource scaling action has to be performed. To avoid this, several auto-scaling solutions have been proposed by academia [27, 28] and by different cloud vendors. All these solutions allow users to define a set of scaling rules regarding the service hosted in the clouds. Each rule is composed by one or more conditions and a set of actions to be performed when those conditions are met. Conditions are typically defined using a set of metrics which are be monitored by the cloud platform, as for example CPU usage, and some threshold. When the threshold is traversed the condition is met and an action is executed. Actions are typically acquisition of new VMs or release of running VMs.

Most auto-scaling proposals are all based on the "conditions and actions" approach, yet they vary substantially in several aspects: which metrics are monitored (and so included in the rules definition); expressiveness of the con-

ditions defining mechanism; and which actions can be taken. Many of them focus on horizontal scaling, i.e., deploying or releasing VMs, while vertical scaling (like for example increasing physical resources of an overloaded server) is not considered, possibly due to the impossibility of changing the available CPU, memory, etc., on-the-fly in general purpose OSs.

Here we analyze the different auto-scaling solutions used by several cloud proposals, commercial ones such as Amazon EC2 and open source solutions such as Open Nebula. Also, we have examined solutions developed by third parties.

Auto-Scaling in Commercial Clouds

Amazon EC2

Amazon provides auto-scaling as part of the service offered by their IaaS EC2 public cloud. This service can be accessed by a web services API or through the command line. Auto-scaling in EC2 is based on the concept of *Auto Scaling Group* (ASG). A group is defined by:

- A *launch configuration* that will be part of the group. The launch configuration is given by the virtual image (that contains the OS and software stack of the VM) and virtual hardware characteristics. As there can only be one unique configuration per auto-scaling group, then all machines must by force have the same virtual hardware specifications. The lack of heterogeneity in launch configurations means that a user wanting a set of heterogeneous VMs has to have at least one different launch configuration, and implicitly different auto-scaling group, per VM type. This is a limitation that makes certain usage scenarios more difficult to implement. For example, some users might benefit by replacing several "small" machines with one single "powerful" machine for cost reasons. Such replacement cannot be done automatically by EC2 auto-scaling service.
- Certain parameters such as the zone where VMs of the group will be deployed (among EC2's available regions, i.e. EU, US East and others) and the minimum and maximum amount of VM instances allowed for the group. When setting a minimum size on the group, the user implicitly configures EC2 to automatically create a new VM whenever some of the running instances are shut down (e.g. because of a failure) and the minimum limit is exceeded.

Finally, the user can define a set of rules for each ASG. In EC2 jargon, the possible actions to be run are denoted *policies*. Each policy defines the amount of capacity (in absolute or relative values) to be deployed or released in a certain group. The platform will create or shut down VM instances in the ASG to meet that capacity demand. Triggers are denoted *metric alarms* and are based on the metrics served by EC2's monitoring service *CloudWatch* (see Section 2.2). Each metric alarm is defined by the metric and related statistic to

be observed (like the average value), the evaluation period, and the threshold that will trigger the alarm. When the alarm is triggered, the action defined by the corresponding policy is run. Load balancers are automatically notified to start/stop sending requests to the created/stopped VMs.

Overall, the auto-scaling functionality that Amazon EC2 offers is considerably flexible. Being one of the first IaaS Cloud providers, Amazon has considerable experience which translates into maturity when it comes to the services it offers.

Microsoft Azure

Microsoft Azure is considered to be a Cloud of type PaaS as the Google App Engine platform or salesforce.com. PaaS clouds offer a runtime environment system (e.g. a servlets container) where users' components can be deployed and executed in a straightforward manner. Thus, PaaS clouds are said to offer an *additional abstraction level* when compared to IaaS clouds [11], so users do not have to handle virtual resources such as machines or networks to start running their systems. In such settings is the cloud system who must scale resources as needed by the container platform in a transparent manner, without any user intervention. Users are not required to monitor its service to scale resources, nor to define scalability rules.

Azure, nonetheless, is an exception. In Azure it is the user who must configure the scalability settings, i.e. the platform does not handle resources scaling on behalf of the user. It is worth noting that the user is give access to instantiating and releasing VMs through an API, so Azure does not isolate users from resources, in contrast with other PaaS platforms. Azure does not have an implicit auto-scaling mechanism, but there is an auto-scaling service for Azure as part of Microsoft's *Enterprise Library Integration Pack for Azure* [29]. This features a customizable rule-based VM auto-scaling mechanism. A user can use two type of rules:

1. **constraint rules** that are independent of the current state of the application, as for instance: minimum and maximum number of VMs for an hourly interval
2. **reactive rules** that reflect changes in the current state of the application and can be configured on user-defined metrics.

Auto-scaling in Azure lacks the maturity of the equivalent service that can be found in Amazon EC2. Given that in the first releases of Azure there was no implicit auto-scaling service some commercial offers such as Paraleap [30] have emerged that try to address this severe limitation. Paraleap automatically scales resources in Azure to respond to changes on demand and is possibly a more mature technology than the equivalent service from the Enterprise Library Integration Pack for Azure.

GoGrid

By default, GoGrid does not offer any auto-scaling functionality. Similarly to Azure, it does provide an API to remotely command the addition or re-

moval of VMs (servers), but it is up to the user to use this API method when required. Auto-scaling is possible by using third party services either as part of the *GoGrid Exchange* [31] program or outside of GoGrid, as discussed in Section 2.2.

The market-based approach that GoGrid has related to Cloud services that are not present in the platform has both advantages and disadvantages. On the plus side of things, the presence of a market of similar services stimulates market-driven service price. The services market also makes the whole Cloud platform extensible, anyone can provide a new service for an uncovered need of the platform's users. On the negative side of things, these services will never be first class citizens of the Cloud platforms and as a consequence they will be priced separately and introduce platform management complexity.

RackSpace

As in the case of GoGrid, RackSpace has not built in auto-scaling capabilities, although it does provide an API for remote control of the hosted VMs. Thus, the user is responsible for monitoring the service and taking the scaling decisions. The creation and removal of resources is done through calls to the remote API.

Third party tools for auto-scaling can be found as part of Rackspaces' *Cloud Tools Market*, a catalog of third party tools developed for Rackspace. This approach is similar to GoGrid's exchange program and shares the same advantages and disadvantages.

Implementations of Auto-Scaling in Open-Source Clouds

In what follows, we will examine the presence of auto-scaling services in the open source counterparts of the commercial providers.

Nimbus

The Nimbus *Phantom protocol* is an ongoing project that provides a partial implementation of the Amazon Auto-scaling service, focusing on preserving a fixed number of healthy VMs running in one or more Nimbus deployments. However, the Phantom protocol does not provide any functionality similar to Amazon's triggers and metric alarms, therefore limiting the flexibility of the service.

Given that Nimbus implements a subset of the Amazon EC2 interface, auto-scaling can be done by using third party services.

Eucalyptus

In Eucalyptus there is no out-of-the-box auto-scaling functionality. Eucalyptus is focused on the management at virtual resource level, and does not control the services running on it. Hence it cannot be aware of their state and so it cannot decide when to add or release resources to the service. However, it does implement the Amazon EC2 API and, as such, can take advantage of auto-scaling services of third party providers.

OpenNebula

OpenNebula does not provide auto-scaling. It promotes a clear distinction of the roles at each layer of the cloud stack. The Cloud is at the bottom of this stack. It aims to ease the management of the virtual infrastructure demanded, and it assumes that scaling actions should be controlled and directed at a higher level. Therefore OpenNebula is not "aware" of the services it hosts, or of their state and is not in charge of supervising the deployed systems.

The work in the "*OpenNebula Service Management Project*" [32] tries to develop a component to be run on top of OpenNebula that handles services (understood as a cluster of related VMs) instead of raw hardware resources. Support for auto-scaling could be added in the future.

Third party services for auto-scaling

RightScale

RightScale [33] is a *cloud management platform* that offers control functionality over the VMs deployed in different clouds. It provides auto-scaling functionality on top of GoGrid, EC2, Rackspace and others based on *alerts* and associated *actions* to be run (one or more) each time an alarm is triggered. This is similar to the auto-scaling services of EC2. But there are some differences. First, alerts can be defined based not only on hardware metrics. Metrics regarding the state of software applications such as the Apache web server and MySQL engine are also available. Also, several actions can be associated to the alert, like for example sending emails to administrators. Besides, these actions can be run periodically for defined intervals of time, not just once. Finally, alerts and actions are usually defined at the server level. But scaling actions are an exception: they are performed only if a certain user-defined percentage of servers "vote" for the action to be run.

Enstratus

Enstratus [34] is a *cloud management platform* that offers control functionality over the VMs deployed in different clouds, as RightScale does. It provides auto-scaling functionality on top of all those clouds, including RackSpace, again very similar to the one provided by the auto-scaling services of EC2.

Scalr

Scalr [35] is an open source project that handles scaling of cloud applications on EC2, RackSpace, Eucalyptus and others. It manages web applications based on Apache and MySQL, and the scaling actions are decided by a built-in logic. Users cannot configure how their applications must scale in response to changes on load.

Cloud Client Load Balancing

This concept of load balancing is not typical to Cloud platforms and has been around for a long time in the field of distributed systems. In its most abstract form, the problem of load balancing is defined by considering a num-

ber of parallel machines and a number of independent tasks, each having its own load and duration [36]. The goal is to assign the tasks to the machines, therefore increasing their load, in such a way as to optimize an objective function. Traditionally, this function is the maximum of the machine loads and the goal is to minimize it. Depending on the source of the tasks, the load balancing problem can be classified as: *offline load balancing* where the set of tasks is known in advance and cannot be modified and *online load balancing* in the situation that the task set is not known in advance and tasks arrive in the system at arbitrary moments of time.

In the case of Cloud computing we can consider load balancing at two different levels: Cloud provider level and Cloud client level. From the point of view of the Cloud provider, the load balancing problem is of type online and is mapped in the following way:

- The parallel machines are represented by the physical machines of the Cloud's clusters
- The tasks are represented by client requests for virtual resources
- Cloud client requests can arrive at arbitrary moments of time

In the case of Cloud client's virtual platform, the load balancing problem is mapped in the following way:

- The parallel machines translate into the virtual resources that the Cloud client has currently running
- The tasks translate into client requests to the Cloud client's platform
- End user requests can arrive at arbitrary moments of time

As a consequence, in Cloud platforms, load balancing is an online problem where end user requests that enter the Cloud client's application need to be distributed across the Cloud client's instantiated virtual resources with the goal of balancing virtual machine load or minimizing the number of used virtual machines.

Although load balancing is not a unique feature to Cloud platforms, it should not be regarded as independent from auto-scaling. In fact, the two need to work together in order to get the most efficient platform usage and save expenses.

The end goal of load balancing from the Cloud client's point of view is to have a more efficient use of the virtual resources that he has running and thus reduce cost. Since most Cloud providers charge closest whole hour per virtual resource, then the only way that cost saving is achieved is by reducing the number of running virtual resources, while still being able to service client requests. It follows that load balancing should be used in conjunction with auto-scaling in order to reduce cost. As a result we have the following usage scenarios:

1. **high platform load** when the Cloud client's overall platform load is high, as defined by the Cloud client. The platform needs to scale up by adding more virtual resources. The load balancing element automatically

distributes load to the new resources once they are registered as elements of the platform and therefore reduces platform load.

2. **low platform load** when the Cloud client's overall platform load is low, as defined by the Cloud client. In this situation, the platform needs to scale down by terminating virtual resources. This is not as trivial as the previous scenario, because the load balancing element typically assigns tasks to all resources and therefore prevents resources from reaching a state where their load is zero and can be terminated. In this situation, the load balancing element needs to stop distributing load to the part of the platform that will be released and, even more, the currently-running tasks of this part of the platform need to be migrated to ensure that a part of the platform will have zero load and therefore can be released.

Load balancing also brings some issues as side effects along with it. One of these is session affinity. Because load balancers distribute load evenly among available nodes, there is no guarantee that all the requests coming from one user will be handled by the same node from the pooled resources. This has the implication that all context related to the client session is lost from one request to another. This is usually an undesired effect. In the great majority of situations, it is desired that requests from the same client be handled by the same node throughout the duration of the client's session. In modern clouds this is referred to as session stickiness.

Mapping of virtual resources to physical resources also has an impact on Cloud clients. There is usually a compromise between the following two opposite use cases:

- The Cloud provider achieves a more efficient resource usage by trying to minimize the number of physical hosts that are running the virtual resources. The downside for the Cloud client is the fact that his platform is at a greater risk in case of hardware failure because the user's virtual resources are deployed on a small number of physical machines.
- The virtual resources are distributed across the physical resources. Thus the risk of failure is less for Cloud clients in case of hardware failure. On the downside, there is a greater number of physical machines running and thus more power usage.

Load Balancing in Commercial Clouds

The problem of load balancing in all Cloud platforms may be the same, but each Cloud provider has its own approach to it, which is reflected in the services they offer and their differences with respect to other providers.

Amazon EC2

Amazon EC2 offers load balancing through their *Amazon Elastic Load Balancing* service [37]. The Cloud client can create any number of load balancers and each will distribute all incoming traffic that it receives for its configured protocol to the EC2 instances that are sitting behind the load balancer. One

single load balancer can be used to distribute traffic for multiple applications and across multiple *Availability Zone*s, but limited to the same Amazon EC2 *Region*.

If an instance that is behind the load balancer reaches an unhealthy state, as defined by the load balancer's health check, then it will not receive any new load, until its state is restored so that it passes the health check. This feature increases the fault tolerance of Cloud client applications by isolating unhealthy components and giving the platform notice to react.

Amazon's *Elastic Load Balancing* service has two ways of achieving stickiness:

1. A duration-based sticky session in which case the load balancers themselves emit a cookie of configurable lifespan, which determines the duration of the sticky session.
2. An application-controlled sticky session in which case the load balancers are configured to use an existing session cookie that is completely controlled by the Cloud client's application.

As for pricing, the Cloud user is charged for the running time of each load balancer, rounded up to an integer number of hours, and also for the traffic that goes through the load balancer. Pricing for load balancers is calculated identically to pricing for any other instance type, given that the balancers are not hardware load balancers, but regular instances configured to balancer incoming network load.

The *Elastic Load Balancing* service is a useful service for any large scale platform, especially after Amazon added HTTPS traffic decryption at load balancer level. Its integration with other EC2 services is also a plus of the service. The one disadvantage that comes to mind is the fact that the load balancing service does not use dedicated hardware load balancers.

Microsoft Azure

In Windows Azure, Microsoft has taken an automatic approach to the load balancing problem, the *Windows Azure Load Balancers*[38] work with VM endpoint connections, which can be Windows or Linux instances. An endpoint is a tuple of a port number and a protocol, which can be either TCP (including HTTP and HTTPS traffic) or UDP. Endpoints need to be connected under the same Cloud service. Once this is done, the load balancer will automatically use a round robin algorithm to balance load on all the public ports of the Cloud service.

The Azure load balancing service is well integrated into the platform and its ability to load balance even HTTPS traffic is highly useful. The only negative point that comes to mind when examining the Azure load balancing service is its lack of maturity when compared to similar services in other platforms.

GoGrid

With respect to load balancing, GoGrid uses redundant hardware load balancers [39]. Each account has free usage of the load balancers.

The load balancers can be configured in terms of what load balancing algorithm to use. The user has a choice between two available approaches:

1. Round robin: with this configuration, traffic is balanced evenly among available pooled nodes.

2. Least connect: this configuration makes the load balancers send new traffic to the pooled node with the least number of currently active concurrent sessions.

Load balancing can disturb client sessions if traffic for the same session is not routed to the same server node that initiated the session throughout the whole duration of the session. To prevent this, load balancers can be configured with a persistency option. The user can choose one of the following three:

1. None: in which situation, traffic is distributed as according to the balancing algorithm selected, ignoring possible session problems.

2. SSL Sticky: in which situation, all SSL traffic is routed to the same destination host that initiated the session. When a new SSL session is initiated, the destination node for handling the first request is chosen based on the balancing algorithm selected for the load balancer.

3. Source address: in which situation, all traffic from a source address is routed to the same destination node after the initial connection has been made. The destination node for handling the first connection of a new source is chosen based on the algorithm that the load balancer is configured to use.

The load balancers also check the availability of nodes in the balancing pool. If one node becomes unavailable, the load balancer removes it from the pool automatically.

The load balancing service offered by GoGrid has advanced features related to load distribution and session affinity. In combination with the fact that load balancing is done by dedicated hardware, their service is one of the most interesting ones from the commercial providers that we have examined.

Rackspace

Rackspace Cloud offers two types of Cloud services: *Cloud Servers* and *Cloud Sites*. The *Cloud Servers* service is of type IaaS in which all auto-scaling, load balancing and backup related issues are left in the hands of the Cloud client. A solution proposed by Rackspace is its *Cloud Load Balancers* [40] service which are dedicated load balancers. These services are widely configurable and offer a wide range of load distribution algorithms, including round robin, weighted round robin, sticky, random and others.

The possible pluses and minuses of a market-based approach to extend cloud services has been discussed in the previous section.

Implementations of Load Balancing in Open-Source Clouds

As the open source IaaS providers evolved later then their commercial counterparts, it is expected (and recurring) that find a lot more features in the commercial side of things.

Nimbus

From the Cloud provider's point of view, there is a feature still under development in Nimbus that allows back-filling of partially used physical nodes. This will also allow preemptable virtual machines, an identical concept to Amazon EC2's spot instances.

From the virtual platform level, there is ongoing work for a high-level tool that monitors virtual machine deployment and allows for compensation of stressed workloads based on policies and sensor information.

For current use, the Nimbus user can set up his own load balancing VM. This approach has the disadvantage of being tedious when compared to using an automated load balancing service that is present in other providers.

Eucalyptus

Eucalyptus does not contain an implicit load balancing service for low-level virtual machine load balancing or high-level end-user request load balancing. Nor does Eucalyptus have a partnership program similar to RackSpace's Cloud Tools or GoGrid's Exchange programs.

As alternatives, one can opt for a complete managed load balancing solution offered by third party providers. Given that Eucalyptus implements the same management interface as Amazon EC2 does, it is relatively easy to find such commercial services. Alternatively, the Cloud user can set up his own load balancing service, as in the case of Nimbus.

OpenNebula

OpenNebula is service agnostic. This means that the service being deployed on OpenNebula needs to take care of load balancing on its own.

From a virtual resource balancing point of view, OpenNebula's virtual resource manager [41] is highly configurable. Each virtual machine has its own placement policy and the virtual resource manager places a pending virtual machine into the physical machine that best fits the policy. This is done through the following configuration groups:

- The *Requirements* group is a set of boolean expressions that provide filtering of physical machines based on their characteristics.
- The *Rank expression* group is a set of arithmetic statements that use characteristics of the physical machines and evaluate to an integer value that is used for discriminate between the physical machines that have not been filtered out. The physical host with the highest rank is the one that is chosen for deploying the virtual machine.

To choose the best physical machine is done by first filtering based on the requirements of the virtual machine and then choosing the physical machine with the highest rank for deployment.

It is trivial to obtain a policy that minimizes the number of used physical resources. It is also possible to obtain a policy that achieves a good distribution of virtual machines among the physical machines with the goal of minimizing the impact that a hardware failure would have on a Cloud client's platform.

The virtual machine placement policies can be configured per virtual machine instance; however, when using a Cloud interface, this cannot be specified by the user and so they are defined by the Cloud administrator per virtual machine type.

The extra control that the OpenNebula administrator has related to VM placement is a plus to the general platform, but it is transparent to the Cloud user. To achieve load balancing, the user has to take a manual do-it-yourself approach, similar the Eucalyptus and Nimbus. This process is clearly inferior to using an automated and integrated approach that the commercial providers offer.

Cloud Client Resource Monitoring

Load balancing is an important feature for an efficient large-scale Cloud client application, yet just as important is ensuring that the platform's resources are working according to specifications.

Keeping track of the platform health is crucial for both the platform provider and the platform user. This can be achieved by using platform monitoring systems. Monitoring can be done on two different levels, depending on the beneficiary of the monitoring information:

1. Low-level platform monitoring is interesting from the point of view of the platform provider. Its purpose is to retrieve information that reflects the physical infrastructure of the whole Cloud platform. This is relevant to the Cloud provider and is typically hidden from the Cloud clients, as their communication to the underlying hardware goes through a layer of virtualization. In general, it is the responsibility of the Cloud provider to ensure that the underlying hardware causes no visible problems to the Cloud clients. For commercial Cloud providers, the low-level monitoring service is usually kept confidential.
2. High-level monitoring information is typically interesting for Cloud clients. This information is focused on the health of the virtual platform that each individual Cloud client has deployed. It follows that the Cloud providers have little interest in this information, as it is the up to the client to manage his own virtual platform as he sees fit. Due to privacy constraints, platform monitoring information is only available to the virtual platform owner and is hidden from the other Cloud clients.

Although this separation is intuitive, there is no clear separation between the two. Each Cloud provider comes with its own interpretation and implementation of resource monitoring.

In what follows we will examine how resource monitoring is achieved in commercial and open-source Cloud platforms.

Monitoring in Commercial Clouds

Amazon EC2

As a commercial Cloud, the low-level monitoring system that Amazon uses for acquiring information on its physical clusters is kept confidential.

The approach that Amazon EC2 has taken with respect to high-level resource monitoring is to provide a service called *CloudWatch* [42] that allows monitoring of other Amazon services like EC2, Elastic Load Balancing and Amazon's Relational Database Service. The monitoring information provided to a Cloud client by the *CloudWatch* service is strictly related to the Cloud client's virtual platform.

The *CloudWatch* service collects the values of different configurable measurement types from its targets and stores them implicitly for a period of two weeks. This period of two weeks represents the expiration period for all available measures and is, in essence, a history of the measure that allows viewing of the evolution in the measurements. *CloudWatch* is actually a generic mechanism for measurement, aggregation and querying of historic data.

In association with the Elastic Load Balancer service and the Auto-scaling feature, *CloudWatch* can be configured to automatically replace platform instances that have been considered unhealthy, in an automatic manner.

CloudWatch comes with an alarm feature. An alarm has a number of actions that are triggered when a measure acquired by the monitoring service increases over a threshold or decreases under a threshold. The measures are configurable and the thresholds correspond to configurable limits for these measures. The possible actions are either a platform scaling action or a notification action. In the case of notification, there are a number of possible channels for doing this. They include Amazon's SNS and SQS services, HTTP, HTTPS or email. The actions are executed only when a measure transitions from one state to another and will not be continuously triggered if a measure persists on being outside the normal specified working interval.

Related to pricing, the *CloudWatch* service is charged separately with a single price per hour, regardless of the resource that is being monitored. Recently, Amazon changed the basic monitoring plan to be free of charge. This includes collection of values every five minutes and storage of these values for a period of two weeks. A detailed monitoring plan is also available that offers value collection at a rate of once per minute and is charged per hour of instance whose resource values are collected.

The *CloudWatch* service is well integrated into the Cloud platform (it was designed to work well with the auto-scaling services for example). It also has advanced features and a level of maturity that few other providers can rival with.

Microsoft Azure

Information about the monitoring system used for low-level platform monitoring of the whole Azure platform has not been given. However, the approaches that the Azure Cloud client has to application monitoring have been documented [43].

For monitoring applications deployed on Microsoft Windows Azure, the application developer is given a software library that facilitates application diagnostics and monitoring for Azure applications. This library is integrated into the Azure SDK. It features performance counters, logging, and log monitoring.

Performance counters are user-defined and can be any value related to the Cloud application that is quantifiable.

The logging facilities of the library allow tapping into:

- Application logs dumped by the application. This can be anything that the application developer wants to log.
- Diagnostics and running logs
- Windows event logs that are generated on the machine that is running a worker role
- IIS logs and failed request traces that are generated on the machine that is running a web role
- Application crash dumps that are automatically generated upon an application crash

The storage location for the log files is configurable. Usually one of two storage environments is used: local storage or Azure storage service. The former is a volatile storage that is included in the virtual machine's configuration, while the latter is a storage service offered by Azure and has no connection to the virtual machine's storage. Usually the latter is preferred for what the Cloud user considers to be permanent logs while the former is used as a volatile storage.

There is no automatic monitoring mechanism for web roles and worker roles running on Microsoft Azure.

The cost implications of using the diagnostics and monitoring libraries are only indirect. There is no fee associated to using them, but there is a fee for storing information in a non-volatile persistence storage service and also in querying that storage service.

In association to the monitoring library, Windows Azure's load balancing mechanisms also offer the possibility of probing the load balanced endpoints every 15 seconds and if an endpoint does not reply it will be taken out of the round robin rotation. Probes can be customized by means of *PowerShell*, a console scripting language.

Azure's approach to monitoring is do-it-yourself, but in contrast to other providers with the same approach, it does give users a set of libraries that facilitate this. The result is still more time consuming than using an automated service.

GoGrid

There is currently no public information related to how GoGrid achieves low-level monitoring on their platform.

The *GoGrid Exchange* program includes third-party packages that target monitoring features ranging from platform security monitoring to resource usage monitoring and database monitoring. These services also include the possibility of configurable alerts based on the values of the monitored measures.

The do-it-yourself approach is present in GoGrid, but its service market helps in finding automated solutions.

RackSpace

As in the case of the other commercial Cloud providers, the approach used by RackSpace for low-level platform monitoring is not public. In what follows we will detail how Cloud clients can monitor their platform on RackSpace.

The Rackspace *Cloud Sites* service offers monitoring capabilities at the whole application level for fixed parameters that include used compute cycle count, used bandwidth and storage. This fits well into the usage scenario that *Cloud Sites* offer: that of a PaaS service; but lack of finer-grained sub-application level monitoring can be a downside for some Cloud clients. Again at an application level, logging for applications deployed on *Cloud Sites* is offered, but in a per-request manner.

On the other hand, the *Cloud Servers* service, which is an IaaS-type of service, does also have monitoring capabilities through the use of third-party partner software, especially tailored for Rackspace's *Cloud Servers* service. These partner solutions are aggregated by Rackspace under the name of *Cloud Tools* [44]. Among these partner services, one can find complete monitoring solutions ranging from general virtual machine monitoring to specialized database monitoring. The services that are specialized on monitoring also feature configurable alert systems.

Recently, RackSpace has acquired CloudKick [45], a multi-cloud virtual platform management tool. CloudKick has a broad range of monitoring features for virtual machines. These include different monitoring metrics from low-level metrics like CPU / RAM / disk utilization to high-level metrics like database statistics, HTTP / HTTPS and others. The monitoring metrics can be extended by custom plugins that are able to monitor anything that the user defines. Measured data can be presented in raw form or aggregated by user-defined means. For data visualization, a real-time performance visualization tool is also provided.

CloudKick also features alerts that have a configurable trigger and repeat interval. The alert prompt can be sent by SMS, email or HTTP. These features make the monitoring capabilities of CloudKick one of the most complete that we have examined.

Implementations of Monitoring in Open-Source Clouds

Nimbus

Nimbus features a system of Nagios[46] plugins that can give information on the status and availability of the Nimbus head node and worker nodes, including changes of the virtual machines running on the worker node.

Monitoring on a Nimbus deployment can also be done via the *cloudinit.d* [47] service, or by using third party distributed system monitoring software as the ones discussed in Section 2.2.

Eucalyptus

Since version 2.0, Eucalyptus has introduced monitoring capabilities[48] for the running components, instantiated virtual machines and storage service. This is done by integrating Eucalyptus monitoring into an existing and running monitoring service. Currently, monitoring has been integrated with Ganglia[49] and Nagios. In Eucalyptus this is done by means of scripts that update the configuration of the running monitoring service to also monitor Eucalyptus components and virtual machines.

As alternative solutions to achieving monitoring at a hardware level, one can employ one of the monitoring systems that have been designed and used in grid environments. Some such systems have been detailed in Section 2.2.

We can also opt for a completely managed monitoring solution offered by third party providers. Given that Eucalyptus implements the same management interface as Amazon EC2 does, it is relatively easy to find such commercial services.

OpenNebula

The built-in monitoring capabilities of OpenNebula focus on the Cloud provider's interest in the physical resources. This functionality is found in the OpenNebula module called the *Information Manager* [50].

The Information Manager works by using *probes* to retrieve information from the cluster's nodes. The probes are actually custom scripts that are executed on the physical nodes and output pairs of Attribute=Value on their standard output. The pairs are collected and centralized. As a requirement, the physical nodes should be reachable by SSH without a password.

Currently, the probes are focused on retrieving only information that underlines the state of the physical nodes and not its running virtual machines (CPU load, memory usage, host name, hypervisor information, etc.). It is advised that this information not be mixed with information of interest to the Cloud client. For such a task, the OpenNebula community recommends using a service manager tool that is a separate entity from OpenNebula. As possible solutions, we can consider commercial services that are specialized in Cloud platform management, including monitoring. Such solutions have been described in the previous sections. Alternatively, we can also turn to cluster monitoring solutions that come from the open-source world, some of which are the result of long research endeavors and have been described in Section 2.2.

Other Research Endeavors That Target Monitoring in Large-Scale Distributed Systems

Over the years as grid computing evolved, so did the need for monitoring large-scale distributed platforms that are built on top of grids. There have been many fruitful research efforts for designing and implementing monitoring systems for large-scale platforms. In the following, we will highlight some of these efforts. The list of research projects that we present is not exhaustive for the field of large-scale platform monitoring.

The Network Weather Service - NWS

NWS [51] has the goal of providing short-term performance forecasts based on historic performance measurements by means of a distributed system. To achieve this, NWS has a distributed architecture with four different types of component processes:

Name Server process is responsible for binding process and data names with low level information necessary when contacting a process

Sensor process is responsible for monitoring a specified resource. The first implementation contained sensors for CPU and network usage. Sensors can be added dynamically to the platform.

Persistent state process is responsible for storing and retrieving monitoring data. By using this type of process, the process of measuring is disconnected from the place where measurements are stored.

Forecaster process is responsible for estimating future values for a measured resource based on the past measure values. The forecaster applies its available forecasting models and chooses the value of the forecaster with the most accurate prediction over the recent set of measurements. This way, the forecaster insures that the accuracy of its outputs is at least as good as the accuracy of the best forecasting model that it implements.

To increase fault tolerance, NWS uses an adaptive and replicated control strategy by an adaptive time-out discovery and a distributed leader election protocol. Sensors are grouped into hierarchical sets called *cliques*. A Sensor can only perform intra-clique measurements, thus limiting contention and increasing scalability of the system.

The implementation uses TCP/IP sockets because they are suited for both local area and wide area networks and they provide robustness and portability.

Ganglia

Ganglia [49] addresses the problem of wide-area multi-cluster monitoring. To achieve this it uses a hierarchy of arbitrary number of levels with components of two types:

Gmon component responsible for local-area monitoring. To gather information from cluster nodes, Gmon uses multicast over UDP, which has proved to be an efficient approach in practice, and it also makes Ganglia immune to cluster node joins and parts.

Gmeta component is responsible for gathering information from one or more clusters that run the Gmon component or from a Gmeta component running in a lower level of the tree hierarchy. Communication between the two components is done by using XML streams over TCP.

In order to achieve almost linear scalability with the total number of nodes in the clusters, the root Gmeta component should not be overwhelmed with monitoring data. To do this, an 1-level monitoring hierarchy should be avoided. Instead, an N-level monitoring tree hierarchy should be deployed, where it is dependent on the number of nodes in the cluster. There is a limitation here in the sense that although nodes can be dynamically added and removed from a cluster without needing to manually update the Ganglia hierarchy, the same cannot be said for Gmon and Gmeta components. The Gmeta needs to have *a priori* knowledge of the of its underlying child nodes.

Supermon

Supermon [52] aims at providing a high-speed cluster monitoring tool, focusing on a fine-grained sampling of measurements. To achieve this, Supermon uses three types of components:

Kernel module for monitoring provides measurements at a high sampling rate. Values are represented in the form of s-expressions [53].

Single node data server (mon) is installed for each monitoring kernel module. It parses the s-expressions provided by the kernel module. For each client connected to this server, it presents measurement data filtered by the client's interest. Data is sent by means of TCP connections.

Data concentrator (Supermon) gathers data from one or several mon or Supermon servers. They also implement the same per client filtering capability that mon servers have. Hierarchies of Supermon servers are useful to avoid overloading a single Supermon, especially in situations where a large number of samples is required or there is a large number of nodes that are monitored.

RVision

RVision (Remote Vision) [54] is an open tool for cluster monitoring. It has two basic concepts that make it highly configurable:

Monitoring Sessions are actually self-contained monitoring environments. They have information on what resource to monitor and what acquisition mechanism to use for the monitoring process associated to the resource.

Monitoring Libraries are actually collections of routines that are used for resource measurement information acquisition. These routines are dynamically linked at runtime and thus information acquisition, which is occasionally intimately tied to the resource that is being measured, is disconnected from the general mechanisms of monitoring.

The architecture of RVision is a classical master-slave architecture. The master node is represented by the RVCore. It is responsible for managing all

the active sessions and distributing the monitoring information to the clients. The communication layer is implemented by using TCP and UDP sockets.

The slave part corresponds to the RVSpy component. There is one such component running on each node of the cluster that is to be monitored. The monitoring libraries that are needed for information acquisition are dynamically linked to the RVSpy. The slave component communicates its acquired information to the master by means of UDP, as this is a low-overhead protocol, and cluster nodes are usually connected by means of a LAN, where package loss is usually small.

Efforts of uniformization and standardization in Cloud computing

Cloud computing is one of the fields of Computer Science where practice (and commercial applications) evolved faster than theory and open source. As a possible implication Cloud platforms do not expose the same API, do not have the same component service and do not have the same behavior.

In what follows, we will explore efforts in the direction of standards and uniformity related to Cloud platforms.

The NIST Definition of Cloud Computing

One of the possible causes of the different behaviors of Cloud platforms is the fact that there are no standard definitions for what a Cloud platform is.

In the USA, the National Institute of Standards and Technology (NIST) proposed a definition for what a Cloud platform is [55]. Since its publication, this definition was used as the *de-facto* standard definition of a Cloud platform in the USA and has gained increasing traction worldwide.

The NIST definition, "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."

Their definition encompasses five essential characteristics of Cloud computing: on-demand self-service, broad network access, resource pooling, rapid elasticity or expansion, and measured service.

δ-cloud

The Apache δ-cloud [56] is an open source API project that abstracts away the differences between IaaS Cloud platforms. It works by providing a REST API service that creates a wrapper over existing Cloud APIs.

Currently, the instance management API supports the following Cloud providers (some not completely): Amazon EC2, Eucalyptus, IBM Smart-

Cloud, GoGrid, OpenNebula, Rackspace, RHEV-M, RimuHosting, Terremark, vSphere, OpenStack, FGCP, Aruba cloud.it.

Open Cloud Computing Interface (OCCI)

The Open Grid Forum (OGF) is a community of users, developers and vendors around grid computing. This community focuses on building standards around grid and Cloud computing. In 2010, OGF release the Open Cloud Computing Interface (OCCI) [57]. OCCI delivers "an API specification for remote management of cloud computing infrastructure, allowing for the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring". The specification includes VM life cycle management and elasticity specifications.

OCCI is free to contribute and has contributions from notable commercial providers, but the number of actual implementations of the OCCI interface amongst Cloud providers is still in its infancy. Most notably we can enumerate OpenNebula and OpenStack amongst the fist implementers.

2.3 Conclusions

The current chapter aims to familiarize the reader with the background information that is the basis of the current Thesis. It discusses the evolution of parallel computing from multicore to cluster and Grid, and finally to Cloud platforms and Cloud federations. Then, it presents in-depth information on the state of the art in resource management for Cloud platforms.

The last parts of this chapter tries to familiarize the reader with the importance of auto-scaling, load balancing and monitoring for a Cloud client platform. The elasticity of Cloud platforms is reflected in their auto-scaling feature. This allows Cloud client platforms to scale up and down depending on their usage. Achieving automatic client platform scaling is done in different ways, depending on the Cloud platform itself. One can opt for the Cloud's built-in auto-scaling feature if present in Amazon EC2, Microsoft Azure, or use a third party Cloud client management platform that offers this functionality: RightScale, Enstratus, Scalr, that are usable in most Cloud platforms. GoGrid and RackSpace have partner programs the GoGrid Exchange and the RackSpace Cloud Tools that provide custom-made tools that work on top of the hosting service that they offer.

Load balancing has the goal of uniformly distributing load to all the worker nodes of the Cloud client's platform. This is done by means of an entity called a load balancer. This can be either a dedicated piece of hardware that is able to distribute HTTP/HTTPS requests between a pool of machines the case of Microsoft Azure, GoGrid, or with a Virtual Machine instance that is configured by the platform provider or by the client himself to do the same job the case of Amazon EC2, Rackspace, Nimbus, Eucalyptus and OpenNebula.

In order to make sure that the platform is in a healthy working state, platform monitoring is used. This is done by means of a service that periodically collects state information from working virtual machines. The monitoring service can either be built as a part of the Cloud platform, as is the case of Amazon's CloudWatch, Microsoft Azure's monitoring package, Nimbus' Nagios plugins, Eucalyptus 2.0's monitoring service and OpenNebula's Information Manager. GoGrid and RackSpace offer this feature by means of their partner programs. Alternatively, Cloud clients can always choose a third party monitoring solution. As examples, we can enumerate CloudKick, Makara or any of the third party Cloud client platform management tools presented above.

Ultimately, all the three presented Cloud features are designed to work hand-in-hand and have the high-level goal of ensuring that the Cloud client's platform reaches a desired QoS level in terms of response time and serviced requests, while keeping the cost of running the platform as low as possible.

Cloud computing is a field where practice evolved a lot faster than theory and as a result, the commercial providers tend to have more mature services than their equivalent open source counterparts. This was visible in all three topics that we have examined in this chapter.

Part 2: Resource management in Cloud platforms

Chapter 3

Auto-scaling Cloud applications

The current chapter builds on the topic of automatic application scaling that was introduced in Section 2.2. Here we go in depth by presenting the existing types of auto-scaling approaches. We then introduce our contribution to the topic of predictive approaches. We have chosen to elaborate a new predictive approach because in the case of on-demand resource scaling, knowledge in advance is necessary as the virtual resources that Cloud computing users have a setup time that is not negligible. The approach that we will present to the problem of workload prediction is based on identifying similar past usage patterns to the current short-term workload history. This approach is useful for any signal that has a repetitive, non-periodic behavior.

We present in detail the auto-scaling algorithm that uses the above approach as well as experimental results by using real-world data and an overall evaluation of this approach, its potential and usefulness.

3.1 Introduction

The evolution of IT software services in the direction of Cloud Computing took a step forward in the efficient use of hardware resources through the use of virtualization. In a traditional hosting services the user receives a static amount of hardware resources that he or she makes use of. In contrast to this, the Cloud approach is to offer on-demand virtualized resources to its users. Because virtual resources can be added or removed at any time during the lifetime of the application hosted on a Cloud, the possibility of dynamic scaling arises. Even more, dynamic scaling can be easily automated either at Cloud provider level or at Cloud client level through the use of the Cloud provider's APIs.

To take full advantage of the benefits of dynamic scaling, a Cloud client (user or middleware) needs to be able to make accurate decisions on when to scale up and down.

To achieve good performance, the Cloud client needs to be able to make accurate scaling decisions. These scaling decisions are influenced by several aspects as for example virtual resource setup time or migration of existing processes to free resources, but resource usage has the biggest impact on the decision.

The idea of self-similarity in web traffic is not new [58]. Based on this a new auto-scaling strategy can be elaborated. By identifying usage patterns that have occurred in the past and have a high similarity to the present usage pattern, a decision can be made as to the necessity and/or direction of scaling for the present situation.

This chapter presents a new approach to the resource usage prediction problem based on identifying past patterns that are similar to the present use of the system. We present an algorithm for identifying the patterns by using an approximate matching approach.

In Figure 3.1 we have a generic Cloud system usage model to have a top-level view on the role of the prediction model. As part of a Cloud client's resource management module, the prediction module uses the client's usage history to try and make an intelligent guess on short-term usage demands. This alone does not constitute the client's scaling decision as there are a number of other relevant factors that should be taken into consideration like the migration of currently running tasks from virtual resources that need to be terminated. In the current work we are focusing only on resource usage prediction.

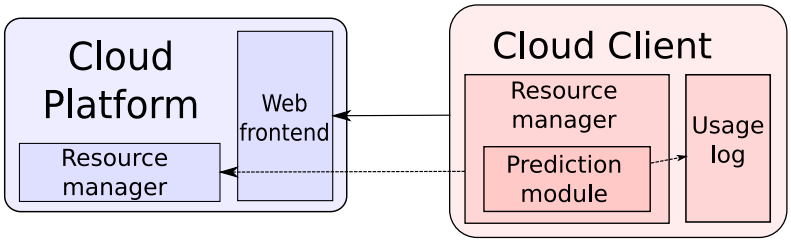


Figure 3.1: The role of the prediction component in a generic model of a Cloud system usage scenario.

The rest of this chapter is organized as follows. The next section present an overview of existing approach given in the literature. Then, Section 5.4 presents our algorithm and its key design principles. Finally, before a conclusion and a description of future work, Section 3.4 presents our experimental results using actual grid traces.

3.2 Related Work

There are currently two main approaches for facilitating the auto-scaling decisions of Cloud client as a result of resource usage. The first approach treats the past server usage as a predictable sequence and constructs a mathematical model around it. As a result, the next value of the request sequence is obtained by evaluating the obtained model at the next time point. In other words, a prediction model is built by considering past resource usage. The second approach is a reactive one, based on the current server load and auto-scaling rules that are set up by a human operator (usually a cloud client). This approach has been often referred to as the "Elasticity rules" approach or the "SLA" approach.

In [59] a description and comparison of three different auto-scaling algorithms is given: auto-regression of order 1 (AR1), Linear Regression, and the RightScale algorithm. The auto-regression of order 1 algorithm is from the first category of auto-scaling algorithms. Its approach consists in using a finite history window and identifying appropriate parameters so that a recurring sequence can be obtained and therefore used to calculate the next values. The obtained parameters are adapted as the window slides along the time axis. The linear regression algorithm is also from the first category and calculates a polynomial approximation of the history of requests. The predicted value is then obtained by evaluating the polynomial at a higher point along the time axis. The RightScale algorithm is from the second category, being a version of threshold-based auto-scaling. Its approach is to use a democratic voting system that is based on the current server load. Each virtual machine owned by the cloud client has a vote based on its current load level and two thresholds: low threshold that corresponds to a "scale down" vote (with a default value of 30% system usage) and a high threshold that corresponds to a "scale up" vote (with a default value of 85% system usage). The votes are collected by a central machine and the majority decides the scaling decision for the whole platform. The three algorithms have been put side-by-side and compared by a metric proposed in the same article. Their performance is considerably high.

A more complex form of SLA-based dynamic provisioning can be described by using elasticity rules that dictate what part of the cloud client needs to scale, in which direction and by how much. In [60], we find such an example with threshold-based rules. This is done by means of an extension to the OVF (Open Virtualization Format), an interoperable, platform and vendor neutral, open format that is used to describe VAs (Virtual Applications). VAs are preconfigured software stacks consisting of one or several Virtual Machines with the purpose of offering self-contained services. The OVF document is actually an XML document containing the description of the OVF package. The elasticity rules come as an extension of this document. They have three components: an associated name, a trigger condition based on the defined key performance indicators and an associated action that represents the concretization of the

rule in the form of instantiating new components of the VA or removing existing component instances. Like the RightScale algorithm, this approach is also a reactive one. Scalability rules have the benefits of combining the high performance of threshold-based algorithms such as RightScale with tune-ability and therefore have been widely used in practice in commercial clouds.

In [61] a decentralized online clustering model is described and proposed for automatic workload provisioning for enterprise grids and clouds and addresses their distributed nature. In this approach a workload prediction algorithm is used and integrated into the system to model the application dynamics. More specifically, a quadratic response surface model is used.

The ideas of workload prediction and workload modeling are by no means new, in fact they have been active areas or research in the field of Grid computing. In [62] we find a fine-detailed study on the topic of Grid performance evaluation by using synthetic workloads obtained from the modeling of grid workloads. The work describes performance metrics useful for evaluating grid environments. These are composed of traditional performance metrics that are time, resource or system related and grid-specific related to workload completion or failure metrics. The article continues by describing the specifics of grid workload modeling. These include user group modeling that underline the importance of taking into consideration statistics for all gobs on one hand and statistics for each user in particular on the other hand, based on his (or her) past actions. The article also describes submission patterns that arise in Grid environments and enumerate some of the current approaches of modeling them that include combining Poisson distributions for daily patterns or by using a polynomial function of degree eight. The authors argue that these pattern modeling approaches may not hold as they are indifferent to workload inter-dependency. The authors continue by presenting the GrenchMark [63] synthetic grid workload generation, execution and analysis framework. They also present extension suggestions to the framework that would make the framework be a better tool for workload generation and analysis.

In [64] we find an integration effort of a grid application development toolkit named Ibis [65], a grid co-scheduler name Koala [66] and the GrenchMark synthetic grid workload generator with the purpose of providing an end-to-end workload generation and testing framework. The authors argue about the benefits that experimental testing of grid systems has over an analytical or simulated test model. The authors also argue in favor of using synthetic grid workloads over real grid workloads or benchmarking approaches. Next the authors describe their integration proposal of building applications with the Ibis toolkit, generating and submitting synthetic workloads with GrenchMark, and then scheduling them with Koala so that the results cal be analyzed with GrenchMark again. As result of experimentation they concluded that workloads generated in GrenchMark can cover a wide range of run characteristics.

A non-linear model for grid workload prediction can be found in [67]. The

authors propose a prediction model as a series of finite known functional components, usually taken from the sigmoid function class, with unknown coefficients. The coefficients are determined by using the least square approximation method on a training set. The training set can be split into a training partition and an evaluation partition. This way an early stop strategy can be applied to avoid data over-fitting. Their model has been tested on a 3D image rendering set of tasks based on the Blue Moon Rendering Tool. The error of their prediction is less than 14% with an average of 7.5%.

In [68] we find a real-time resource provisioning system for massive multiplayer online games based on a predictive usage model. The application is dynamically provisioned on a Grid environment. The authors propose a predictive model based on neural networks as this approach has more predictive power than simpler approaches like exponential smoothing, yet is faster in terms of runtime than more complex approaches like autoregressive models, integrated models or moving average models. The neural network is prepared with two offline phases that include gathering of training samples and using them to train the neural network. As results of experimentation, the neural network approach has proved to have a greater accuracy when compared to the other tested prediction methods: average, moving average, last value and exponential smoothing. The obtained prediction error during the experiments has a maximum value of 33% and a minimum value of 4.94%.

3.3 String Matching based Scaling Algorithm

Idea Description

A Cloud client is provisioned depending on his platform's usage. The usage of a Cloud client can sometimes have a repetitive behavior. This can be caused by the similarities between tasks that the Cloud client is running or the repetitive nature of human behavior. Given the self-similar nature of web traffic it follows that current usage patterns of online services have a probability of having already occurred in the past in a very similar form. Therefore we can infer what the system usage will be for a Cloud client by examining its past usage and extracting similar usages.

The pattern strategy has two inputs: a set of past Cloud client usage traces and the present usage pattern that consists of the last usage measures of the Cloud client. Cloud clients working in the same application domain have a higher similarity in resource usages. Due to this similarity it follows that the most relevant historic resource usage data that can be used comes from Cloud clients working in the same application domain. Therefore it would make sense to isolate historical data based on application domains before usage.

The present usage pattern of the Cloud client is used to identify a number of patterns in the historical set that are close to the present pattern itself. Identified patterns should not be dependent on their scale, just on the relation

between the elements of the identified pattern and the pattern we are looking for. The resulting closest patterns will be interpolated by using a weighted interpolation (the found pattern that is closest to the present pattern will have a greater weight) and will have as result an approximation of the values that will follow after the present pattern. In essence, the usage of the Cloud client is predicted by finding similar usage patterns in the past or in other usage traces.

The problem of finding a pattern inside an array of data that is very similar to a given pattern is close to the problem of string matching. The approximate string matching problem has been widely studied especially in its relation to bioinformatics problems, yet it is considerably different from the problem we are addressing.

One definition for the approximate string matching problem is the following: given a text string $T = t_0 t_1 ... t_n$ and a pattern $P = p_0 p_1 ... p_m$ find a substring of consecutive characters from T call it $T_{i,j}$ that has the smallest edit (or Levenshtein) distance as possible [69].

The edit distance is defined as the number of simple string operations: insert, delete, replace and sometimes exchange, that needs to be performed on the identified text substring to have equality to the pattern. The operations can have the same or different weights, depending on problem needs. The identified match can have any length because of the possible insert and delete operations.

In the problem that we are addressing, the edit distance cannot be applied as we are not comparing string character values, but floating point values. We are interested in identifying sub-arrays of the same over very close length and and whose floating point absolute value difference is as close as possible to zero. An insertion into or deletion from the identified sub-array would have a great impact on the floating-point difference.

We shall now describe the problem of string matching and its relation to the problem that the current chapter addresses, as well as our proposal for the approximate variant that is relevant to our problem.

The String Matching Problem

The string matching problem consists in finding the position of a string (called pattern) inside a larger string. There are several approaches solutions to this well known problem. We have chosen the Knuth-Morris-Pratt (abbreviated KMP) as its performance are good as described in [70]. The KMP algorithm consists of a preprocessing step with a running time of $\Theta(m)$ where m is the length of the matching pattern and a matching step with running time of $\Theta(n)$ where n is the length of the string to match against. The algorithm is also embarrassingly parallel as it is data independent. Therefore the input data can easily be divided into independent blocks on which the algorithm can run in parallel.

1	A A	B B	A A	B B	A C	B	C
2	A	B A	A B	B A	A B	B C	C
3	A	B	A A	B B	A A	B B	C C

Table 3.1: KMP example

P =	A	B	A	B	C
π =	_	_	0	1	_

Table 3.2: Calculating the auxiliary array

The efficiency of the KMP algorithm is due to its approach in saving unnecessary comparisons in case of a mismatch between the pattern and the string to match against. It is able to do this by first identifying repetitive prefixes of the input pattern in the preprocessing step.

Consider the following example: input pattern P = "ABABC" and matching string T ="ABABABC". There are three possible positions for P to be found in T, by using a sliding window approach, until one of the matches succeeds.

In the example given in Table 3.1, when step 1 fails, the pattern slides to the next possible position in the matching string and a new comparison is made in step 2. After step 2 fails, the pattern slides once again and reaches step 3 which makes a full match.

In Table 3.1, step 2 can be skipped altogether if we consider the relation that the pattern has with itself, i.e. its repetitive prefix. Once the first 4 characters of P have been matched against the 4 consecutive characters in T (the following 4 characters starting from position 0) we deduce that there is no need to restart the whole matching from position 1 in T because, from analyzing P we know that the match will fail as the 4 characters of T starting from 0 are the same as the first 4 characters of P starting from 0.

To assist the matching process, an auxiliary array is constructed over P (called π) that contains at position i, the ending position of the largest prefix of P that is a suffix of P[0..i]. For the P in our example, we have the results given in Table 3.2.

The entries in π that have a value of "_" represent entries that are not prefixes of P. For example the second "A" in P is both a prefix and also a suffix of P[0..2] = "ABA". The largest prefix that is also a suffix for P[0..2] is "A" and has the ending position at P[0]. This means that once we have matched P[0..2] = "ABA" in T and P[3] does not match in T, we can continue matching in T from the same index of T, and we can start in P knowing that

we have already matched the first character in P, as it is a prefix of length 1 of P[0..2]. Therefore we resume matching with the P index of $\pi[2] + 1 = 0 + 1 = 1$. Now, resuming our matching example, in step 1 we have matched P[0..3] to T[0..3]. We have P[4] != T[4], but we know that P[0..3] = "ABAB" has "AB" as the largest prefix that is also a suffix. So we can resume by matching T[4] to $P[\pi[3]] = P[1]$, skipping P[0].

The preprocessing step The goal of the preprocessing step is to compute the π array. At each index i, π stores the end position of the longest prefix of P[0..i], that is also a suffix of P[0..i]. The algorithm for this has a runtime of $\Theta(m)$ where m is the length of P (Algorithm 1).

Algorithm 1 Calculate-prefix(P)

```
1: m ← length(P)
2: π[0] ← -1
3: k ← -1
4: for q ← 1 to m - 1 do
5:    while k > -1 and P[k+1] ≠ P[q] do
6:       k ← π[k]
7:    end while
8:    if P[k+1] = P[q] then
9:       k ← k+1
10:   end if
11:   π[q] ← k
12: end for
13: return π
```

The matching step The matching algorithm (Algorithm 2) has a runtime of $\Theta(n)$, where n is the length of T, the string to match against. It is very similar to a naive matching algorithm, but improved to skip redundant comparisons.

Algorithm Description

The KMP algorithm (Algorithm 2) is a good solution to the string matching problem. Despite the great similarities, our own pattern matching problem has some particularities of this own:

- an approximate matching is needed since the odds of finding an identical pattern to the one we are looking for are considerably low;
- matches which are too dissimilar either on small intervals or as a whole need to be discarded;
- when comparing the pattern to the matching data, scale also needs to be taken into consideration. To be more exact, the scale of the pattern and the scale of the possible match should not affect the comparison, therefore it needs to be scale-independent.

Algorithm 2 KMP(T, P)

```
1:  n ← length(T)
2:  m ← length(P)
3:  π ← Calculate-prefix(P)
4:  q ← -1
5:  for i ← 0 to n - 1 do
6:     while q > -1 and P[q+1] ≠ P[i] do
7:        q ← π[q]
8:     end while
9:     if P[q+1] = T[i] then
10:       q ← q+1
11:       if q = m-1 then
12:          write "Found at position" i-m
13:          q ← π[q]
14:       end if
15:    end if
16: end for
```

– the resulting matches are interpolated having different weights on the final result, based on their similarity to the identified pattern.

In order to do an approximate matching, the original KMP algorithm needs to be changed in the content of both functions, therefore they need to be modified accordingly.

Two types of approximation errors are used for the matching:

– an instant error which dictates the amount by which the current match is allowed to differ from the pattern by comparing in smallest possible units;
– a cumulative error that characterizes the amount by which the current match is allowed to differ from the pattern as a whole. This is basically a sum of the instant errors of the whole matching.

Figure 3.2 illustrates graphically the difference between the two types of acceptable errors (instant and cumulative) when comparing two patterns.

Scale-Independent Comparison

The distance between the pattern we are trying to match and a candidate pattern should be computed in a scale-independent manner by first normalizing the two pattern values to a common scale. To decrease floating point approximation errors, one can choose a distance computation that does not use divisions and therefore calculating only on integer values.

As an example, consider the pattern and the candidate from Figure 3.3(a). The pattern is an array containing values: 20 , 38 , 21 and the candidate match contains values: 42 , 81 , 39 . In this form, we cannot compare the two

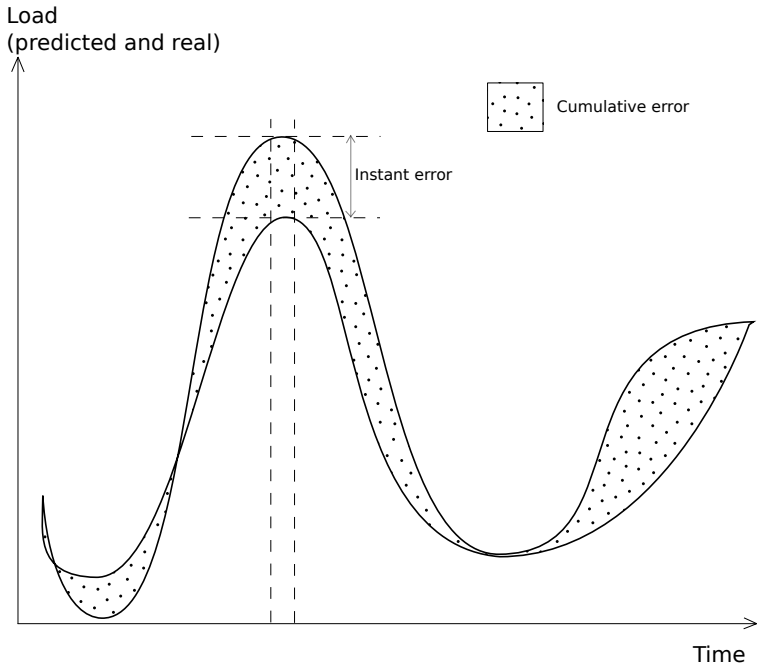


Figure 3.2: Difference between the two types of acceptable errors.

patterns. A first idea would be to normalize both arrays to a floating point [0..1] interval and then compare. Working with floating point numbers can be avoided by working with big integer numbers. To reach a common scale we simply multiply each array by the scale of the other. For the scale of each array we can simply consider the first element. As a result, the pattern array is multiplied by the scale of the candidate (this is 42) and the candidate is multiplied by the scale of the pattern (which is 20). The result is depicted in Figure 3.3(b). In this new situation, comparing two components of each array is done simply by subtraction. The instant error is used here to assure that there is no pair of components of the two arrays whose values differs too much.

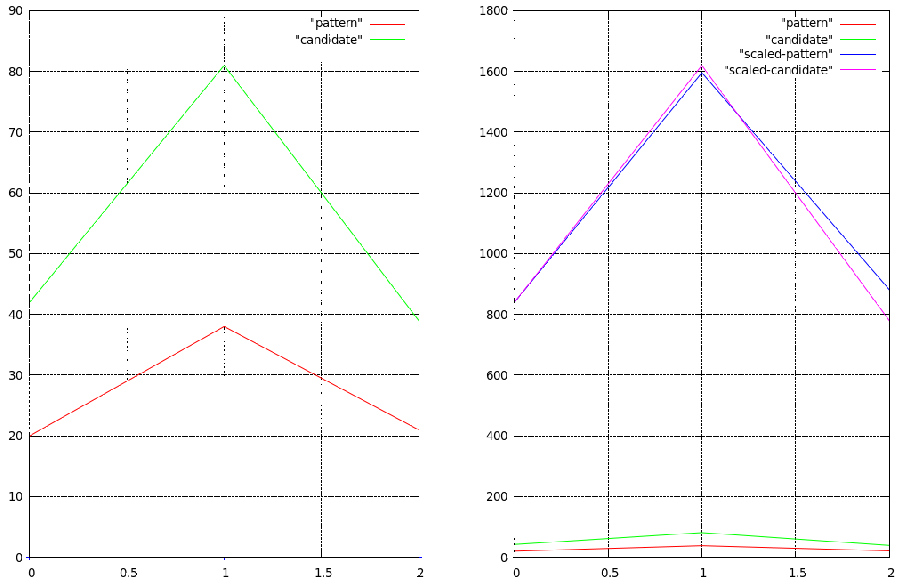
Once the comparison is done, the identified candidate is stored along with its total distance from the pattern. This facilitates the significance of the result, as the candidate that is closest to the pattern has a higher weight in final result. The pseudocode for computing the instant error is illustrated Algorithm 3.3.1 in the Distance function.

Algorithm 3.3.1 Distance(PatternElement, PatternScale, DataElement, DataScale)

---

return PatternElement $\times$ DataScale - DataElement $\times$ PatternScale

---



(a) Initial (b) Normalized to common scale

Figure 3.3: Scale-independent comparison

The cumulative error is obtained by summing up the instant errors from all the elements of the pattern and candidate. This is illustrated in the CumulativeDistance function.

Algorithm 3.3.2 CumulativeDistance(P, T, DataOffset)

```
1: patternScale ← P[0]
2: dataScale ← T[DataOffset]
3: length ← length(P)
4: distance ← 0
5: for index ← 0 to length do
6:     distance ← distance + | dataScale × P[index] - patternScale × T[index
       + DataOffset] |
7: end for
8: return distance
```

KMP Modification

The prefix calculation function is changed as described in Algorithm 3.3.3. The scales of the two components compared are represented by the first value of each component. This is arguable, but in practice we have achieved good results with this approach. In the function, scaleK represents the scale of

Algorithm 3.3.3 Calculate-prefix-approx(P, ACCEPT_INST_ERR)

```
1: m ← length(P)
2: π[0] ← -1
3: k ← -1
4: scaleK = P[0]
5: scaleQ = P[1]
6: for q ← 1 to m - 1 do
7:    dist ← Distance(P[k+1], scaleK, P[q], scaleQ)
8:    maxDistance ← ACCEPT_INST_ERR × scaleQ × P[k+1]
9:    while k > -1 and dist > maxDistance do
10:      k ← π[k]
11:      dist ← Distance(P[k+1], scaleK, P[q], scaleQ)
12:      scaleQ = P[q - (k+1)]
13:   end while
14:   if dist ≤ ACCEPT_INST_ERR × scaleQ × P[k+1] then
15:      k ← k+1
16:   end if
17:   π[q] ← k
18: end for
19: return π
```

the prefix and scaleQ represents the scale of the postfix of the pattern. The Distance function returns an appreciation of the distance between two different pattern instances, each having a different scale which is passed as parameter. The comparisons on lines 9 and 14 assure that the current instant distance does not differ by more then the acceptable error ($ACCEPT\_INST\_ERR$ in percentage) from the actual pattern that we are matching. The scaleQ term, representing the scale of the data, from the comparison is needed for bringing the current term of the pattern to the same scale as the data.

The matching algorithm is changed as described in Algorithm 3.3.4. The main difference when compared to the original KMP algorithm is the use of the instant and cumulative distances as a means of filtering out potential matches that are too different either on small time intervals or as a whole.

On lines 10 and 16 we ensure that the instant distance between the identified candidate and the pattern is no more than what the acceptable error ($ACCEPT\_INST\_ERR$) permits. In order to ensure a correct comparison, the pattern term needs to be scaled to the same size as the data, hence the scaleT term is used in the comparison. Filtering by cumulative distance is done in lines 20 to 24. The CumulativeDistance function returns a sum of instant distances for every instant of the two compared arrays. The running time of this function is $\Theta(m)$ where m is the length of the arrays, which in our case is always equal to the length of P. Line 22 of the algorithm assures that the cumulative distance of the candidate does not differ more than is accepted

Algorithm 3.3.4 KMP-approx(T, P, ACCEPT_INST_ERR, ACCEPT_CUMUL_ERR)

```
 1: n ← length(T)
 2: m ← length(P)
 3: π ← Calculate-prefix(P)
 4: q ← -1
 5: scaleP = P[0]
 6: scaleT = T[0]
 7: for i ← 0 to n - 1 do
 8:    dist ← Distance(P[q+1], scaleP, T[i], scaleT)
 9:    maxDist ← ACCEPT_INST_ERR × scaleT × P[q+1]
10:    while q > -1 and dist > maxDist do
11:       dist ← Distance(P[q+1], scaleP, T[i], scaleT)
12:       q ← π[q]
13:       scaleT = T[i - (q+1)]
14:       maxDist ← ACCEPT_INST_ERR × scaleT × P[q+1]
15:    end while
16:    if dist ≤ maxDist then
17:       q ← q+1
18:    end if
19:    if q = m-1 then
20:       dist ← CumulativeDistance(P, T, i - m + 1)
21:       maxDist ← ACCEPT_CUMUL_ERR × patternSum × scaleT
22:       if dist ≤ maxDist then
23:          StoreSolution(dist / scaleT, i - m + 1)
24:       end if
25:       q ← π[q]
26:       scaleP = P[q+1]
27:       scaleT = T[i - (q+1)]
28:    end if
29: end for
```

by the cumulative error ($ACCEPT\_CUMUL\_ERR$) from the pattern itself. The pattern itself is represented by the patternSum term in the comparison. This is a sum of all the terms in the pattern and should be calculated only once, at the beginning of the algorithm. The pattern sum needs to be brought to the same scale as the candidate sequence and therefore the scaleT term is used. Filtering by an acceptable cumulative error that is smaller or equal to the acceptable instant error is useless. This conclusion is trivial when taking into consideration that the cumulative error is a sum of all the instant errors.

The use of the cumulative error changes the running time of the matching algorithm to $\Theta(n \times m)$ in the worst case, where n is the length of the string to match against and m is the length of the input pattern.

Interpolating the Values Found

Once approximate matches have been found, the problem of obtaining a relevant result from those matches is raised. Each match should have a contribution to the final result that is proportional to its relative distance to the pattern with respect to the other identified patterns. This corresponds to a weighted sum of the identified matches, where weights are calculated by considering the distance of the current match to the pattern and to the rest of the matches. Once the weights are calculated, the interpolation is performed between the following L elements after each approximate match. The result is a predicted sequence of length L.

Algorithm parameters

The algorithm accepts a number of parameters used for fine-tuning in accordance to each use-case. These parameters are:

- The maximum number of matches (called closest neighbors) to take into consideration (denoted K).
- The length of the predicted sequence (denoted L).
- The acceptable instant error representing the amount by which the identified sequence is allowed to differ on the smallest possible interval lengths from the pattern we are looking for.
- The acceptable cumulative error which represents the amount by which the identified sequence is allowed to differ as a whole from the pattern we are looking for.
- The input set of data representing the database of past requests.
- The input pattern representing a sequence with the last period of requests received.

The first parameter is not independent of the others. It is actually influenced considerably by the acceptable errors. The correlation is strong and can be expressed very easily: the larger the acceptable error, the more matches the algorithm identifies, but the more irrelevant they will be.

Calculating the acceptable errors

The value of the acceptable errors can be calculated based on the maximum number of neighbors that we wish to find. The approach for this is to use a binary search to zone in on the appropriate values for the acceptable errors. By using the binary search approach, we have obtained values that have proved to be good in practice. We have used a lower bound of 20% of K for a minimum of identified neighbors and 90% of K as the upper bound for maximum number of identified neighbors.

Calculating the appropriate pattern length

The length of the pattern that represents the last traces of server usage has a great impact on the results of the algorithm. Finding the appropriate

length is a problem in its own as we have a trade-off between patterns of big lengths that yield a small number of similar candidates, that might be to small in order to be usable, and patterns of small lengths, that find more candidates but they tend to be more irrelevant to our current situation. We have chosen two approaches to this problem. The first approach is to find the most lengths of the most frequent repetitive patterns and use the same length as input to the prediction algorithm.

We have the following constructive approach to identifying the length of the most frequent repetitive patterns:

1. find all similar patterns of length 2 in the historic data
2. take all similar patterns of length 2 and try to match the next element too. This yields all similar patterns of length 3.

   ...
3. take all patterns of length n and try to match the next element too. This yields all similar patterns of length $n + 1$.

The result is that the number of identified similar patterns decreases as the length of the patterns increases:

$$count[n + 1] \leq count[n] \leq ... \leq count[3] \leq count[2]$$

The conclusion is that the most frequent patterns are of the ones with length 2. In practice, using a pattern length of 2 would have the following consequences:

- Good for predicting very short in advance (i.e. 1)
- Loses meaning when trying to predict longer sequences
- The idea of trend is lost as the steps are very small while the trend is a longer sequence.

We need to have a better way for choosing the pattern length, that would give more relevant results and avoid pollution as much as possible.

The length of the pattern should be influenced by the time it takes to service a request on the server. We then have the following possibilities:

- Median / average
  - Representative of most of the requests
- Minimum
  - A large pattern cannot match against a smaller pattern that is half different
  - A small pattern can match against a large pattern that's half different
  - The minimum is very probably close to 0 (grid testing experiments)
  - A close minimum can be selected (ex. 5% - 10% from the bottom)

	Animoto	LCG	Nordugrid	Sharcnet
Avg	1296	8970	91893	33516
Min	4	0	0	0
Median	283	255	3861	12165
Max	22452	586702	1452763	7449415

Table 3.3: Job length statistics for the data sources. Values represent time in seconds based on the running time of the recorded jobs.

By using real-world grid traces from the workload archive of TUDelft University [71]. We have used the running time in seconds of each job and obtained the results given in Table 3.3. We have tested traces from several research grids [72, 73, 74] to get a real-world appreciation of possible values for the pattern length, by taking different metrics.

We can also consider plots of sorted job lengths in seconds.

The conclusions given by Figures 3.5(a), 3.6(a), and 3.7(a) along with the previous table are that, for all practical purposes, a pattern length that is a minimum or even median of the time it takes to service a request, is unusable when dealing with servers that have a similar usage to the research grids described above. In practice we have used the average of the request service time and have obtained good results.

3.4 Experimental Results

In all our experiments we have used a time unit of 100 seconds and we have discretized the traces by this time unit. The plots of the grid traces and the predicted traces represent the total number of CPUs used by different jobs running in parallel in the time unit of 100 seconds. We have focused only on CPU usage as the information of memory usage was not available. Nevertheless, should the information of memory usage be available our approach can also be applied for its prediction.

Data Sources

We have tested our auto-scaling approach with traces from one Cloud client application and three different research grids, each having its own usage particularities, with main differences in the frequency and amplitudes of changes in their overall usages.

Animoto [1]

Animoto is a Cloud client application that specializes in automatically-orchestrated videos starting from user-generated content. Their platform usage represents oscillations as per user activity.

1. `http://animoto.com`

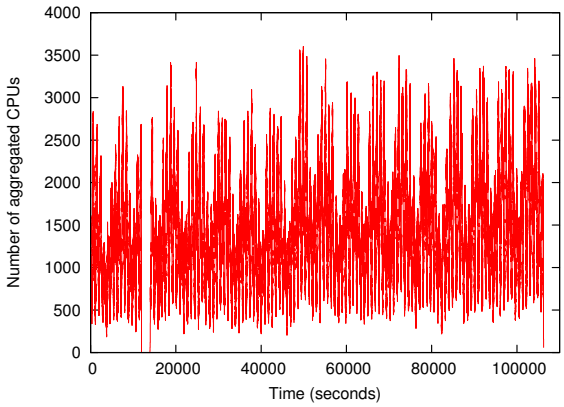
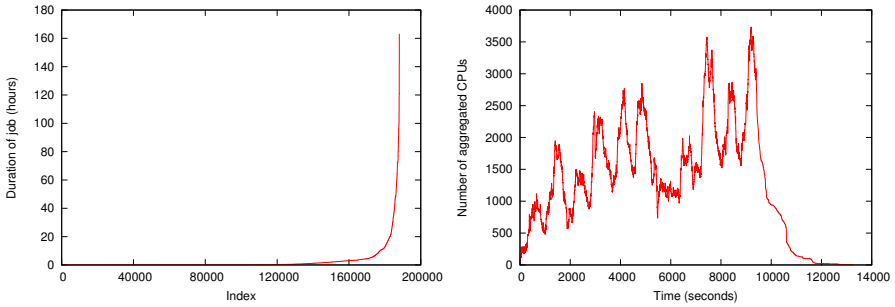


Figure 3.4: The Animoto platform - total number of CPUs aggregated over 100 seconds

LCG - Large Hadron Collider Computing Grid [2]

Here we find traces from several nodes from the computing grid associated to the Large Hadron Collider. Its behavior is mildly oscillatory and a plot of the total number of CPUs used in time slices of 100 seconds, discretized across time intervals of 100 seconds can be found in Figure 3.5(b).



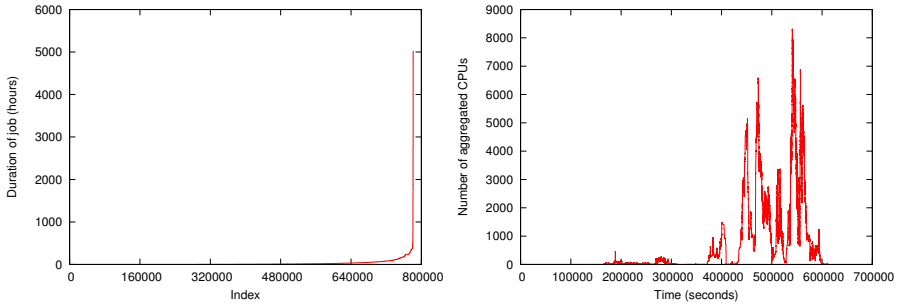
(a) Job running times in seconds, sorted (b) Total number of CPUs aggregated over 100 seconds

Figure 3.5: The LCG platform

NorduGrid [3]

Here we find higher amplitudes for oscillations as the grid is more heterogeneous than the previous. A plot of the total number of CPUs used in time slices of 100 seconds, discretized across time intervals of 100 seconds can be found in Figure 3.6(b).

2. `http://lcg.web.cern.ch/LCG/`
3. `http://www.nordugrid.org`

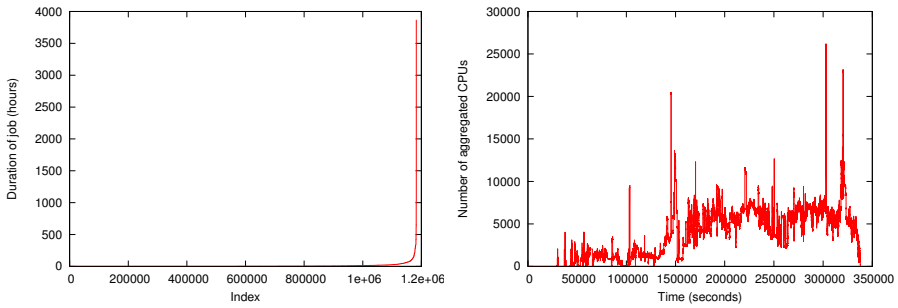


(a) Job running times in seconds, sorted (b) Total number of CPUs aggregated across 100 seconds

Figure 3.6: The NorduGrid platform

SHARCNET [4]

SHARCNET has been described as a "cluster of clusters". Its volatility is very high and its amplitudes can reach surprising peaks. A plot of the total number of CPUs used in time slices of 100 seconds, discretized across time intervals of 100 seconds can be found in Figure 3.7(b).



(a) Job running times in seconds, sorted (b) Total number of CPUs used per time slice of 100 seconds

Figure 3.7: The SHARCNET platform

Experiment setup

All the experiments use the server traces of the same form of input data as described above with time units of 100 seconds, and resource usage value consisting of the total number of CPUs used across the 100 seconds. A pattern length of 100 time units has been used for all the experiments (this is 100 ×

4. `http://www.sharcnet.ca`

100 seconds - approximately 2.7 hours of server time) and predictions are made for one time unit, this is 100 seconds, which is a little over 1 minute 30 seconds.

The results are displayed under the form of a set of standard metrics that include minimum, maximum, median, and average percentage and value difference between the prediction and the actual value.

A second set of metrics has also been used that allows the comparison to other existing auto-scaling algorithms. This metric was proposed and used by UCSB to compare the performance of three existing auto-scaling algorithms [59]: auto-regression of order 1, linear regression and the RightScale democratic voting algorithm.

We have also measured the average running time necessary for calculating one prediction. This has an impact on the practical usefulness of the prediction since it needs to be subtracted from the prediction time - which is 100 seconds - to calculate the effective prediction time.

The metric proposed by UCSB is influenced by platform availability and cost by the following formula:

$$\frac{(A_{log})^{\alpha}}{C} - \frac{\gamma C}{A_{log}} + \beta \tag{3.1}$$

where: $A = \#serviced\_requests/\#of\_requests$ represents the availability of the platform,

$$A_{log} = -log(1 + \delta_a - A), \delta_a < 1 \tag{3.2}$$

represents the availability in logarithmic scale and $C = \#CPU/(hours \times 0.10)$ represents the cost. The constants α, β, γ and δ_a have been chosen through experimentation.

We have used two versions of the metric proposed by the UCSB team:

- An instant score where we considered resource cost as being charged per fraction of an hour, although this is not the case in current cloud providers
- A second score where we take the maximum prediction over the course of an hour and use that as static provisioning for the whole hour

Results

We have done two types of tests: self-prediction in which a resource trace is used as historic data to predict itself, ignoring exact matches and cross-prediction tests in which the resource trace of one source is used to predict the resource trace of another source. The results can be seen in Table 3.4.

Predicting LCG with LCG as historic data

Metric	A w/ A	A w/ L	L w/ L	L w/ N	N w/ L	S w/ N
Min error (%)	0.0	0.0	0.0	0.0	0.0	0.0
Max error (%)	100	856.87	53.4	100.0	1146.00	528.03
Med error (%)	2.69	4.08	1.0	1.2	1.74	0.9
Avg error (%)	5.42	7.4	1.749	7.32	35.38	375.65
UCSB (max / hour)	-1.39	-15.95	10.66	3.43	30.64	-3.23
UCSB (instant)	-18.38	-38.75	-2.68	-10.71	27.27	-2.06
Runtime (ms)	186.625	27.63	41.734	514.956	162.949	528.418

Table 3.4: Results of prediction experiments with traces from the four data sources: Animoto, LCG, Nordugrid and SHARCNET. Experiments consist of predicting one platform's usage with another platform's traces as historic data, across time slices of 100 seconds. The naming of the columns is done after the following convention **A w/ B** where A is the platform whose usage is being predicted and B is the platform whose trace is being used as historic data.

We have done a self-prediction test by using LCG as historic data with the purpose of predicting LCG itself. When filtering out potential pattern candidates, exact matches have been ignored, since the pattern itself is a piece of the historic data. The results of this experiment can be found in Table 3.4, column "L /w L". Figure 3.8 shows a zoom-in of the actual value of resource usage in the LCG platform and the predicted resource usage.

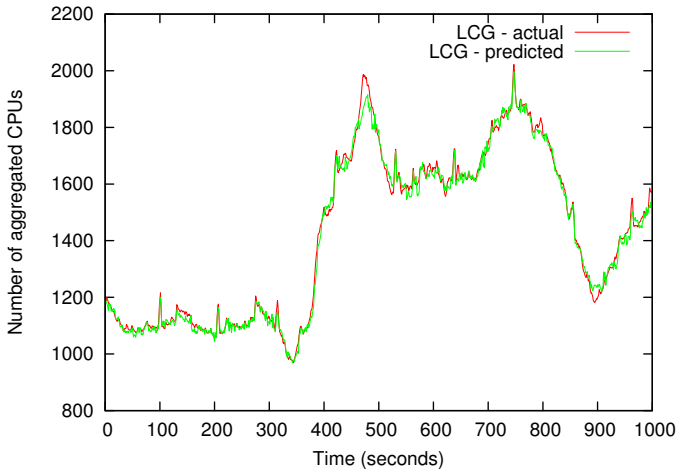


Figure 3.8: Zoom into the plot of CPUs used in time slices of 100 seconds versus time in units of 100 seconds for the LCG platform's actual resource usage (shown in red) and predicted resource usage (shown in green).

Predicting NorduGrid with LCG as historic data

We have experimented with using traces from a different grid which is close to the one we are trying to predict. In the current test case, we have tried to

predict NorduGrid workloads by using LCG as historic data. The experiment's results can be seen in Table 3.4, column "N w/ L". A zoom into the plots of the actual resource usage and the predicted usage is shown in Figure 3.9.

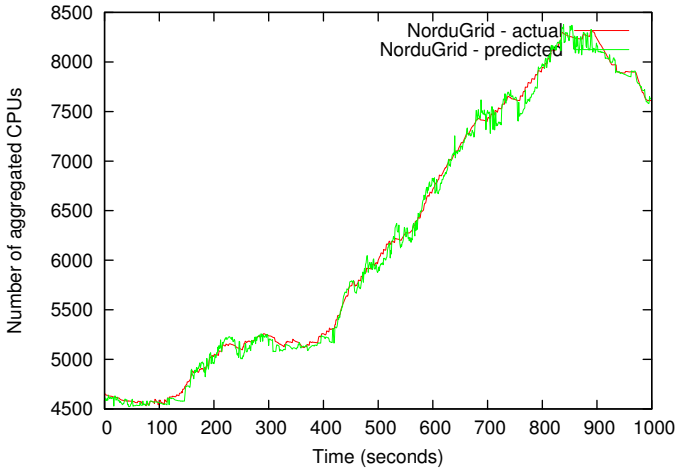


Figure 3.9: Zoom into the plot of CPUs used in time slices of 100 seconds versus time in units of 100 seconds for the NorduGrid platform's actual resource usage (shown in red) and predicted resource usage (shown in green) by using LCG as historic data

Predicting LCG with NorduGrid as historic data

We have experimented with the symmetric of the previous experiment in trying to predict LCG workloads by using NorduGrid as historic data. The results are shown in Table 3.4, column "L w/ N". A zoom into the plot of the actual resource usage of the platform and the predicted resource usage is shown in Figure 3.10.

Predicting SHARCNET with NorduGrid as historic data

We have also experimented the behavior of the algorithm when using historic data that does not have a high similarity to the workload that is being predicted. In our experiment, we have used NorduGrid traces as historic data when trying to predict SHARCNET traces. The results of this experiment are available in Table 3.4, column "S w/ N".

An analysis of the results reveals that this is a feasible approach to auto-scaling. It is clear that the algorithm yields better results when the set of historic data that is used has a similarity to the signal that is being predicted. This similarity is influenced by several parameters that constitute the domain of the server whose load is being predicted. It follows from the obtained results that data from the same domain can easily be used to predict one-another.

The time necessary for computing one prediction instance has proved in practice to be low relative to the prediction time.

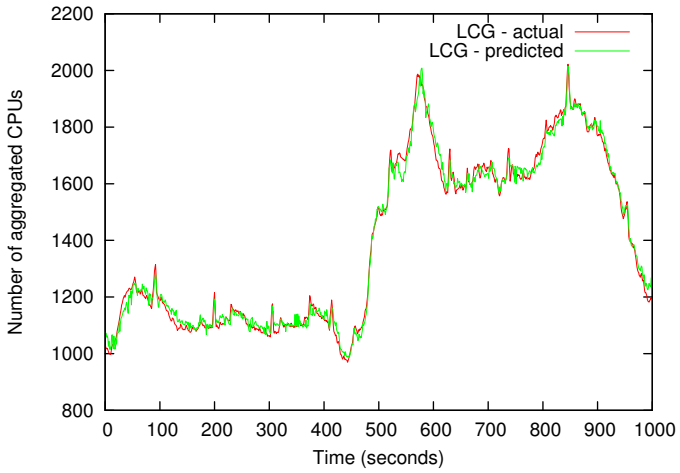


Figure 3.10: Zoom into the plot of CPUs used in time slices of 100 seconds versus time in units of 100 seconds for the LCG platform's actual resource usage (shown in red) and predicted resource usage (shown in green) by using NorduGrid as historic data.

Predicting LCG with LCG as historic data and varying pattern lengths and historic data lengths

Although we cannot show that the algorithm yields the best results, we can show that its results improve as we increase the size of the historic data and as we find the best pattern length to take into consideration when predicting. The tables below illustrate results when varying the pattern length and the length of the historic data used for prediction. We have varied the historic data from 100% - the full set, to 50%, 25% and 12.5% of the set. The pattern length has also been varied from 1000 time units to 500, 100, 50 and 25.

		Data length (%)			
		1.0	0.5	0.25	0.12
Pattern len (x 100 s)	1000	-18.99	-36.37	-57.83	-97.37
	500	-9.43	-19.97	-23.47	-43.06
	100	5.44	3.32	4.05	4.05
	50	9.41	9.6	8.48	8.21
	25	10.67	11.11	12.62	11.79

Table 3.5: Score given by the UCSB metric (maximum per one hour) for predicting LCG with LCG as historic data and by varying the length of the pattern used for prediction and the length of the set of historic data.

Table 3.6 contains the results of the experiment when calculating the metric proposed in [59] and using instant values for the the number of virtual

Pattern len (x 100 s)	Data length (%)			
	1.0	0.5	0.25	0.12
1000	-38.96	-59.57	-79.25	-103.57
500	-31.36	-38.54	-45.88	-63.18
100	-11.08	-13.81	-16.49	-15.44
50	-4.54	-4.83	-7.22	-8.23
25	-0.14	-0.3	-0.05	-1.36

Table 3.6: Score given by the UCSB metric (instant) for predicting LCG with LCG as historic data and by varying the length of the pattern used for prediction and the length of the set of historic data.

resources. Table 3.5 contains results of applying the previous metric by using the maximum across each hour as reference point for virtual resources and cost.

We have experimented with various lengths of the historic data set and of the pattern that is considered for input. The results with the prediction error in each case can be seen in Table 3.7. Although this does not show that the algorithm yields the best possible results, it does show that there is a clear tendency for the accuracy of the prediction to improve as we increase the size of the historic data and as we find the best pattern length to take into consideration when predicting. The results table illustrates results when varying the pattern length and the length of the historic data used for prediction. We have varied the historic data from 100% - the full set, to 50%, 25% and 12.5% of the set. The pattern length has also been varied from 1000 time units to 500, 100, 50, 25, 12 and 2 time units.

Pattern length (x 100 seconds)	Data length (%)			
	1.0	0.5	0.25	0.12
1000	5.3%	9.5%	19.7%	100%
500	3.7%	6.0%	8.6%	18.7%
100	1.0%	1.2%	1.3%	2.0%
50	0.6%	0.5%	0.9%	1.3%
25	0.3%	0.3%	0.4%	0.5%
12	0.2%	0.2%	0.2%	0.3%
2	98%	100%	100%	82%

Table 3.7: The prediction error obtained for various lengths of historic data and pattern lengths for the LCG platform.

The reader will note that in our experiments we have considered only CPU usage as measure and prediction target. In a Cloud environment, a virtual resource usually has more characteristics associated to it than just CPU power. In particular, memory usage is one of the most notable characteristics. Our

approach can also be used to have a prediction of the memory usage if the server traces also contain information about past memory usage. With predictions for both memory and CPU usages, the scaling component of the Cloud client should be able to more accurately decide the characteristics of the virtual resources that are to be instantiated or released. The topic of making a good scaling decision both in direction and in virtual machine characteristics is an interesting topic of research, yet it is beyond the scope of the current work.

3.5 Conclusions

One of the most important benefits of Cloud Computing is the ability for a Cloud client to adapt the number of resources used based on its actual use. This has great implications on cost saving as resources are not paid for when they are not used. Dynamic scalability is achieved through virtualization. The downside of virtualization is that it has a non-zero setup time that has to be taken into consideration for an efficient use of the platform. It follows that an accurate prediction method would greatly aid a Cloud client in making its auto-scaling decisions.

In this chapter, a new resource usage prediction algorithm is presented. It uses a set of historic data to identify similar usage patterns to a current window of records that occurred in the past. The algorithm then predicts the system usage by interpolating what follows after the identified patterns from the historical data. Experiments have shown that the algorithm has good results when presented with relevant input data and, more importantly, that its results can improve by increasing the historic data size. This makes the evaluation of the algorithm be context dependent. As a note, the current approach can be applied to predict any repetitive, non-periodic signal.

As future work directions we will be looking into ways that a relevant set of historic data can be composed for a particular application domain.

Chapter 4

Economic model based resource management

Infrastructure as a Service clouds are a flexible and fast way to obtain (virtual) resources as demand varies. Grids, on the other hand, are middleware platforms able to combine resources from different administrative domains for tasks execution.

This chapter explores the possibility of using an IaaS service by a grid platform as a provider of resources in the form of virtual machines. This requires grids to be able to decide when to allocate and release those resources. Here we introduce and analyze by simulations an economic mechanism a) to set resource prices and b) resolve when to scale resources depending on the users demand. This system has a strong emphasis on fairness, so no user hinders the execution of other users' tasks by getting too many resources.

Our simulator is based on the well-known GridSim software for grid simulation, which we expand to simulate infrastructure clouds. The results show how the proposed system can successfully adapt the amount of allocated resources to the demand, while at the same time ensuring that resources are fairly shared among users.

4.1 Introduction

As discussed in the previous chapters, the novelty in Cloud computing comes from scalability *i.e.* the ability to acquire and release resources on-demand, without a predetermined contract or lease and with a small time overhead. Virtualization, the key to on-demand resources, also allows lots of flexibility when it comes to the software stack installed on the virtual machine.

Grid systems, on the other hand, are a well-known technology that can provide a seemingly unique infrastructure from several resource providers, possibly heterogeneous. Typically, grid users send their tasks to the grid platform which will distribute them among the resources available. Activities such as resource location, execution scheduling, security handling, etc. are managed by the grid.

Grids can use clouds as infrastructure providers so they can deploy or release resources in order to react to changes on demand, or to anticipate to variations on that demand if load prediction systems (like the approach presented in Chapter 3) are available. This demand of resources will be induced by the amount (which depends on the triggering rate) and size of tasks sent to the grid. Thus, grids will be able to allocate only the infrastructure they require. Besides, grids can benefit from clouds flexibility as they will be able to run tasks with heterogeneous software requirements in the same host. We deem this is of special interest in some typical grid usage scenarios where several users compete for resources which are freely (in monetary terms) available, but are also limited. Examples of such scenarios are several scientific environments, where resources can be provided by one or several entities. This proposal is mainly oriented to that kind of setups.

However, this brings a new problem: how can grids decide when to scale up or down resources? For example, a grid system could decide to enqueue incoming tasks, or even to reject them, instead of allocating new resources. Hence, it seems reasonable that users should be able to point out if their tasks have a certain priority so they should be run as soon as possible, instead of being enqueued or discarded.

Here we propose an economic mechanism to enable grids to decide how to scale resources. A price is computed for each resource, so the cost of running each task can be calculated. These prices are adapted depending on the demand. Users have a limited, periodically renewed budget to run their tasks. Negotiation follows a *tender/contract-net* model [75] where users ask for offers for each task they want to run and choose the most suitable one following a utility function also defined by them. The tender/contract-net model is known to be the economic model that optimizes users' utility [76], which is the main goal in the scenarios we address. Also, as no user can demand too many resources due to budget restrictions, no user can get a unfair share of those resources. Tasks have a deadline, so those that could not be run (not suitable offer was received) before their deadline expires will be marked as failed.

The main contribution of this work is the introduction of a hybrid grid-cloud architecture where one or more clouds provide infrastructure resources and the grid:

- Automatically scales resources usage to attend a variable demand to run tasks with possibly heterogeneous software needs.
- Splits resources fairly among users. Here, *fair* does not mean *equally*. Maybe some users need more resources than others, and those should be

granted while there are enough resources for all.

In the presented architecture the grid system is not in charge of ordering users' tasks, which are processed following a FIFO approach. We assume instead that each user is the one who must prioritize her tasks following her own criteria, i.e.: the user is the one to decide which is the next task to execute. A tasks ordering mechanism is also proposed in this work, based not only on the priority assigned to each task by the user, but also on the risk of not being able to run that task which is computed using its size and the time to its deadline. This mechanism shows a better outcome than ordering tasks using only their priority.

We test and evaluate this proposal by simulations run using the GridSim [77] simulator, whose features we extended in order to suit the requirements of our experiments. Experiments are run over a hybrid architecture that combines a grid system with IaaS clouds. The grid system used as basis of this architecture is DIET [7]. In [78] Caron et al. introduce and discuss a first proposal of the architecture presented here.

The rest of the chapter is organized as follows: Section 4.2 details the architecture proposed; Section 4.3 explains how the system market approach is implemented, i.e. how currency flows, how offers for each task request are computed, how prices are adapted, etc.; Section 4.4 shows the results of some simulations that test key features of the proposed system; Section 4.5 presents an analysis of related work in the area of clouds and economy-based grid systems; finally Section 4.6 discusses the conclusions of the work presented.

4.2 Grid-Cloud Architecture

The solution proposed in this chapter combines a hierarchical grid system, DIET, with several clouds that will provide resources to the grid. To describe this solution we need first to outline how DIET works.

DIET [7] connects its components through a hierarchical tree for scalability. The basic DIET component is the *Agent*. Agents have scheduling and data management capabilities, but here we will focus on their primary and most basic functionality: service location. Figure 4.1 depicts DIET components organization. Each DIET grid has a unique *Master Agent* (MA) on the top of its hierarchy. This MA gets service requests from users. Each request goes down through the hierarchy formed by the agents until it reaches the *Server Daemons* (SeD), that interact with the execution environments and provide the actual execution services. Each Agent knows the services that can be executed by the SeDs at the bottom of each one of its children Agents, and it will not forward service requests to those Agents whose corresponding SeDs cannot run the service. Each SeD is connected to DIET's hierarchy through *Local Agents* (LA), LAs are intended to be at the resources provider site. When some request reaches the SeD, it builds a reply reporting its state. Replies are

sent back through the hierarchy up to the MA. Replies are ordered by some objective function that depends on the SeDs' state, so the "best" SeDs are first in the list. Finally the MA will send the list of replies to the user, who will pick some SeD in the list (usually the first one) and command it the task to run.

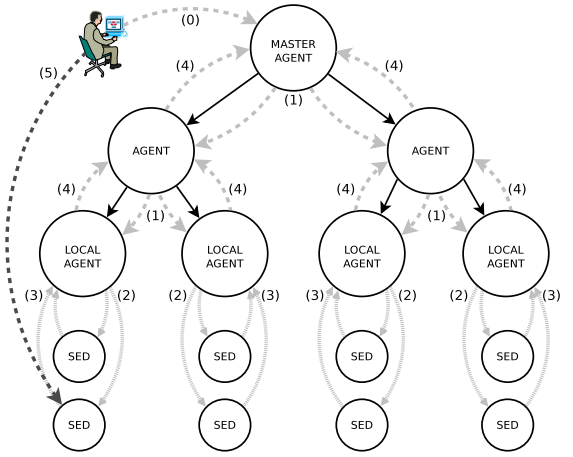


Figure 4.1: DIET hierarchical layout.

DIET's layout makes straightforward to connect IaaS clouds as resource providers to the grid. IaaS systems will be connected to the SeD nodes, who will decide when to scale (allocate and release) resources to attend users requests. Services will be run in the VMs hosted in the cloud. IaaS providers can be built on top of hardware providers by using several open solutions such as OpenNebula [79], Eucalyptus [80] or Nimbus [81]. Such solutions have simple remote interfaces that SeD nodes can use to request the creation of VMs and/or networks to connect them. Once a VM is created, the SeD node will be in charge of connecting to it to run services in order to attend users' tasks. Figure 4.2 shows a first sketch of the elements involved in the described layout, using OpenNebula as a possible IaaS Provider. In our proposal the user interacts at all times with DIET elements (MA and SeDs). She is totally unaware about the fact that SeDs may run tasks in VMs supplied by IaaS clouds.

The hybrid approach presented here is detailed in Figure 4.3. A new module for tasks allocation is placed between the IaaS system and DIET's SeD node (Task Allocation Module, TAM), that will be in charge of computing

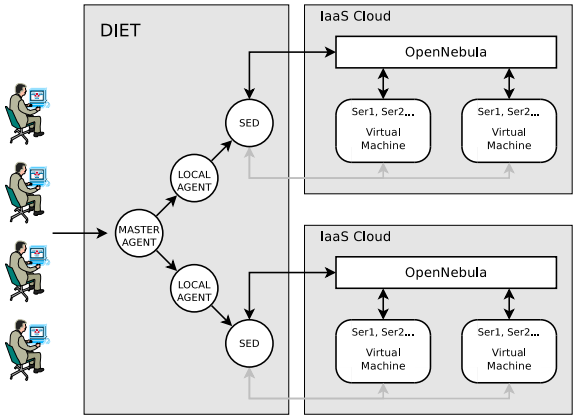


Figure 4.2: Sketch of the proposed hybrid grid-cloud layout.

where the tasks sent by users can be executed and will adapt prices as demand changes. A task can be run in an already active VM, or in a new VM that will be demanded by the TAM to the cloud provider. The cloud provider will have a catalog describing the hardware configuration of the VMs that the TAM can instantiate. Each VM will have one or more processing elements (virtual CPUs) with their corresponding queue of pending tasks. When computing allocations for a given task, the TAM must take into account the tasks already in the queues of each VM. The TAM can ask to the cloud provider whether a VM of a certain type can be instantiated or not which will depend on the resources of the physical hosts available. This is necessary so the TAM can determine allocations in new VMs. We assume that a set of disk images containing the VMs software stack (OS, libraries...) required to run the tasks is available.

Now, the main goal of the grid system is to ensure fairness in how resources are shared. To achieve this we propose a market-based approach, that is described in detail in the next section. The characteristics of this approach impose certain changes in the way SeD nodes run. Those changes are also explained in the next section.

4.3 Using Markets to Reach Fairness

Markets can be defined as a way to exchange "goods", in this case the right to run tasks on some infrastructure. In such market, resources have a certain price associated, and so users must take into account their (limited) budget to decide when and where to demand the execution of those tasks. If resource prices are set taking into account the demand, and budgets are allocated equally among users, by intrinsic market dynamics we can expect resources to be fairly shared (a more thorough discussion about the role of

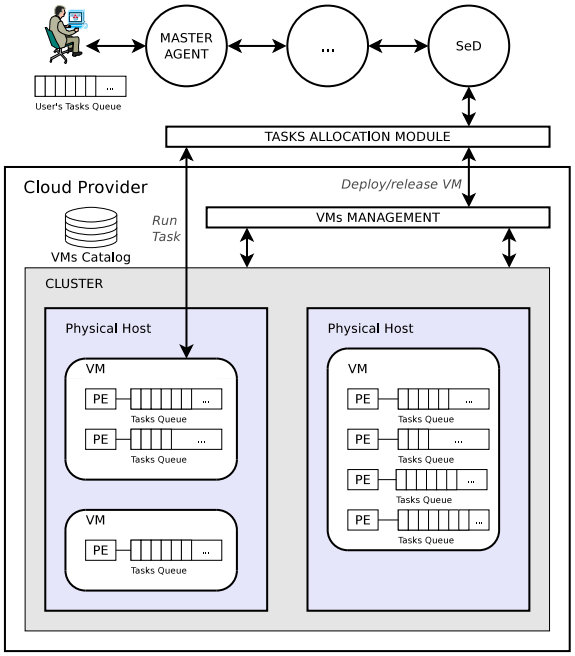


Figure 4.3: Architecture overview.

markets as a solution for fair resource sharing can be found in [82]).

When designing a market environment several decisions must be taken regarding different features:

- How currency is managed.
- How negotiation is performed, i.e. how requests are sent and how offers are collected.
- How offers for each user request are built.
- How resource prices, that determine each offer cost, are computed.
- How the user chooses the best offer.

The rest of this section describes the characteristics of our proposed market and explains the design decisions taken regarding them.

Currency

Users budget will be bounded by the amount of *virtual currency* they have (using real money is possible, but is has several drawbacks, see Section 4.5). An initial budget is assigned to each new user in the system. Users cannot run tasks beyond their budget. On the other hand, currency should be assigned to users to avoid the potential problems of *starvation* (users cannot access resources), *depletion* (users hoard currency to monopolize resource access at certain times) and *inflation* (prices grow due to uncontrolled addition of currency to the system) [83]. Several options are possible:

- The global value of all resources is periodically computed, taking into account their present prices. This would represent the total "wealth" of the system. This amount is then split and sent to the users.
- A given fixed amount is sent periodically to all users. Providers (i.e. clouds) do not hoard the money they get from users.
- SeD/Clouds do not hoard neither drop the money received from the users. Instead, all that money is periodically gathered and forwarded back to the users.

The two first options can easily lead to inflation as currency is injected to the system even if the demand of resources is low. Also, new users will be in an adverse situation as previous users can hold big amounts of virtual currency. Thus, the third option seems the more feasible, and is in fact similar to the idea proposed in Mirage [84] (see Section 4.5). Our proposal adds a new entity, the *Virtual Bank*, will be in charge of gathering all the incomes of the cloud providers. Periodically, the total of these incomes is split and sent to the users in equal parts. Payments from users to providers are done directly once the corresponding task execution is finished, with no intervention from the Virtual Bank.

Tasks Execution Negotiation

Every time the user needs to run a task, it sends a `REQUEST_FOR_OFFERS` message that through DIET hierarchy will reach all available SeD nodes (in fact, their corresponding TAM modules, see Figure 4.3) connected to some cloud provider. Our simulations take into account three resources (more can be easily added): CPU, disk and memory. Hence, each request contains information about the requested amount of all involved resources (CPU measured in MIs, memory measured in MBs, and disk measured in GBs).

When a `REQUEST_FOR_OFFERS` message reaches a certain TAM, this module will build a set of offers to execute the task. The process of creating offers for a request is detailed in Section 4.3. An allocation offer A is a tuple that contains the cost and time that it will take to run a given task $(A^{\text{TIME}}, A^{\text{COST}})$. A TAM can create none, one or many allocation offers for a task T_i. When all possible allocations to run the task have been computed by the provider they are sent back to the user in an `OFFERS` message. If the provider could not find any suitable offer then the message will be empty. `OFFER` messages are sent again through DIET. Each node in the hierarchy (LAs, Agents and the MA) will gather all the offers they receive from their children nodes for each `REQUEST_FOR_OFFERS` they had forwarded before, and will build a new `OFFERS` message with the offers carried by the `OFFERS` messages from its children. Of course, the node will not build and send the new `OFFERS` message for a task until it node has received an `OFFERS` message for that task from all its children.

Finally, only one `OFFERS` message will reach the user, containing all offers from all SeDs. Then, the user will choose the most suitable allocation offer

using some utility function, and will send a `RUN_TASK` message directly to the corresponding provider. If the `OFFERS` message is empty, or it does not contain any suitable offer, then the task is stored in a queue by the user to be tried again later. An offer is not suitable for a task if its cost A^{COST} is greater than the available user's budget, or if the time to execute it A^{TIME} exceeds the task deadline. Each user periodically checks the tasks stored, discarding as failed those tasks whose deadline has expired.

A `RUN_TASK` message carries the time and cost conditions from the original offer. When the TAM receives such message, it computes again possible allocations for the task to check if it still can honor the offer. If it is not so (due to shortage of resources or changes on resource pricing) then the user is notified. In such case the task is stored by the user as if it had no offer. If the task can still be run under the offer conditions then it is executed. When the task is finished the result is sent to the user by a `RUN_RESULT` message, which carries the task results or the corresponding errors. If the task could not be run due to some reason (e.g. unexpectedly the deadline was surpassed during execution) then the user discards the task as failed.

Building Tasks Allocations Offers

Before describing how offers are built by cloud providers, it is necessary to outline how physical hosts and VMs are characterized. Then we describe the process of computing all possible options to run a task. Each option will then become an allocation offer $(A^{\text{TIME}}, A^{\text{COST}})$ that will be sent in the corresponding `OFFERS` message.

Physical Hosts and Virtual Machines

Each cloud provider has a *catalog* of VM types available $\{V_1, \cdots V_n\}$. Each VM type V_j defines a hardware configuration with the resources it has: amount of Processing Elements PEs[1] V_j^C and their processing speed V_j^s (in MIPS); memory V_j^m; and disk V_j^d. Also, there is information about how long it takes to start a VM of that type V_j^{START} and the price of creating such instance V_j^{COST}. Each cloud provider has a set of m physical hosts $\{H_1, \dots H_m\}$. Each host H_k has a set of H_k^C CPUs all with the same processing capacity H_k^s. For each processor $p_{k,l}$ $(1 \leq l \leq H_k^C)$ in host H_k we represent by $p_{k,l}^a$ the available processing capacity of that CPU (in MIPS), i.e. the processing capacity not used for any of the VMs allocated in the host. Conversely $p_{k,l}^u$ is the used capacity, so $p_{k,l}^a + p_{k,l}^u = H_k^s$. Also, the amount of memory in host k is given by H_k^m, while $H_k^{m,a}$ and $H_k^{m,u}$ are the available and used memory in that host respectively. H_k^D represents the amount of disks of host k, and H_k^d represents

1. To avoid confusion with physical CPUs, we will denote as PEs the VMs' CPUs.

their capacity. For each disk $z_{k,l}$ ($1 \leq l \leq H_k^D$), $z_{k,l}^a$ and $z_{k,l}^u$ are the available and used storage capacity of disk $z_{k,l}$ ($z_{k,l}^a + z_{k,l}^u = z_{k,l}$).

When a new VM of type V_j is allocated in some host H_k then the corresponding values are updated. The PEs must be allocated in V_j^C physical CPUs (of course $V_j^C \leq H_k^C$) with enough available capacity. For example, processor $p_{k,1}$ would be assigned one of VM's PEs only if $p_{k,1}^a \geq V_j^s$. When one PE is assigned to some physical CPU its corresponding parameters are updated so for example $p_{k,1}^a = p_{k,1}^a - V_j^s$. Also, the host available memory must be enough to allocate the VM memory, and if so then it must be updated when the VM is finally created $H_k^{m,a} = H_k^{m,a} - V_j^m$. Finally, the capacity of the disk where the VM storage will be set is also updated so $z_{k,l}^a = z_{k,l}^a - V_j^d$. $H_k^{m,u}$ and $z_{k,l}^u$ are updated likewise. If there are more than one host where the VM can be created, then the host running more VMs is used for the new VM. The goal is to use as less physical hosts as possible at all times, which in turn should impact on the power consumption (unused hosts can be in sleep mode, which will demand less power). On the other hand, as time passes some VMs can become *idle*, i.e. they have run all tasks assigned and are waiting for new tasks to be executed. Periodically it is checked how long each one of these idle VMs has been in that state. If any VM has been idle for a period longer than a certain threshold time, that VM is switch off and its resources are released. So if the VM was of type V_j and was running on host H_k, then the available resources are updated as expected: $H_k^{m,a} = H_k^{m,a} + V_j^m$, and so on.

All PEs have a FIFO queue of tasks associated. When a `RUN_TASK` message reaches a cloud provider the set of possible allocations must be computed again to check whether that task can be run within the cost and time originally offered (which are carried by the `RUN_TASK` message). If so, the provider will choose among the found allocations the one that maximizes the user's utility function. Depending on the allocation, the task can 1) be assigned to a free PE and start immediately; 2) be assigned to a PE that is busy (an then it will be added to the PE's tasks queue); 3) require to start a new VM, in such case a new VM instance will be created, once it is ready the task will be assigned to any of its PEs. The algorithm to compute all possible allocations for a task is described in the next section.

Tasks Allocations Computation

An allocation for a task is the assignation of the task to a certain PE in some VM. Each allocation will have a cost and duration ($A^{\text{TIME}}, A^{\text{COST}}$).

When an user asks for offers to compute a task, or sends a request to execute it, potential allocations for that task must be looked for. In the former case, each allocation found is sent back to the user as an offer (see Section 4.3). In the latter case, if some of the possible task allocations meets the time and budget given by the user, then the task will be processed by the corresponding PE.

All the possible allocations for a task are calculated by an algorithm that comprises two steps: 1) first the TAM analyzes the VMs already present and whether they can run the task; 2) then the possibility of creating new VMs to run the tasks is checked. The output of each step will be a collection of allocations. Both sets will be combined resulting in the final set of potential allocations for the task. The remaining of this section details these two steps, specifying also how A^{COST} and A^{TIME} are computed for each allocation:

1. First, the TAM analyzes the state of the already present VMs in order to find running VMs where the task could be executed. They are grouped by the VM type (V_j) they belong to. These VMs can be active (running some other tasks) or idle (all PEs are free). Idle VMs are checked first. For a task i, let c_i, d_i and m_i be the amount of CPU, memory and disk required by that task respectively. The time to run the task i in an idle machine of type V_j is $A^{\text{TIME}} = c_i/V_j^s$. Regarding cost computation, let P_m, P_d and P_c be the price of 1MB of memory, 1GB of disk, or the computation of MI (prices computation is explained in Section 4.3), then the cost of the task is computed as:

$$A^{\text{COST}} = P_m m_i + P_d d_i + P_c c_i \tag{4.1}$$

   After looking for allocations in the idle VMs, active VMs are checked too, i.e. those VMs whose PEs are running some other tasks. For each active VM of type V_j, the TAM checks each one of its V_j^C PEs to see when it will be available (it will not be running any task and its queues are empty). Let q be the amount of tasks waiting in the PE's queue, numbered from 1 to q. Let $\{c_1, \dots c_q\}$, $\{m_1, \dots m_q\}$ and $\{d_1, \dots d_q\}$ the CPU, memory and disk those tasks demand. Let also c_0 the remaining MIs to be executed of the task being run when the allocations are computed. Then, the PE will be busy until $t_b = (c_0 + \sum_{0<x\leq q} c_x)/V_j^s$. If at t_b the amount of disk and memory that will be available in the VM (i.e., not used by the tasks run by the others PEs in at t_b, which is known studying their queues) will be enough to run the task, then a new allocation where the task is assigned to that PE can be built. The time to run the task will be:

$$A^{\text{TIME}} = \frac{c_0 + \sum_{0<x\leq q} c_x + c_i}{V_j^s} \tag{4.2}$$

   The cost of running the task is computed as before (see Eq. 4.1).

2. Second, each VM type V_j is analyzed to check a) if a *new* VM instance of that type could run the task, i.e: $V_j^m \geq m_i$; $V_j^d \geq d_i$; b) if there is any physical host H_k with enough spare capacity where the VM can be instantiated, that is, it has enough available memory $V_j^m \leq H_k^{m,a}$, it has some disk with enough available storage $V_j^d \leq z_{k,l}^a$ and it has V_j^C processors with enough spare processing capacity V_j^s.

If both conditions are met, then a new allocation has been found. The allocation time is computed as the addition of the time to start the VM, plus the time to run the task itself:

$$A^{\text{TIME}} = V_j^{\text{START}} + \frac{c_i}{V_j^s} \tag{4.3}$$

The allocation cost is computed as the addition of the cost of instantiating the VM, plus the cost of using the resources for the duration of the task which depends on their price. Let P_m, P_d and P_c be the prices of memory, disk and CPU (price computation is explained in Section 4.3). Then:

$$A^{\text{COST}} = V_j^{\text{COST}} + P_m m_i + P_d d_i + P_c c_i \tag{4.4}$$

Choosing the Best Allocation to Run a Task

When the SeD receives a task to run (in a `RUN_TASK` message) and the TAM has computed all the suitable allocations for that task, then one of those allocations must be chosen. The TAM applies the user' utility function to choose which is the best allocation choice.

But in some cases different allocations will have the same time and cost (and so the same utility value). For example, one allocation can run the task in an already active VM with some free PEs, and another one can run the task in an idle VM of the same type. So, when several allocations have the same time and cost the TAM applies some heuristics that favor energy saving to choose the definitive allocation for the task:

- The grid will prioritize those allocations that will run the task in an already active VM (i.e. one or more of its PEs are running tasks).
- If no allocation in an active VM is found, then the grid will prioritize those allocations that assign the task to an already present VM (which will be idle). If there are several idle machines, the VMs that have been idle for the shortest period of time are preferred. The goal is to keep idle machines in that state while possible, so their resources will be eventually freed when they are shut down (the grid shuts down the VMs that have been idle for longer than a certain time).
- Only if no allocations in active or idle VMs are found, then allocations that require instantiating a new VM are considered.

Resource Prices Computation

The price adaptation mechanism applied takes into account the resources demand to change prices accordingly. This algorithm is run periodically by the TAM to compute the price of the resources in the cloud.

A cloud provider will price resources differently depending on the goals pursued. To maximize benefits, the provider could apply the approach explained in [85]. In collaborative environments, the cloud provider can also

try to maximize resource usage and so the amount of tasks run. This is the approach taken in this work.

The algorithm goes as follows. Let r be the total amount of some resource in the provider's site, measured in a certain unit (e.g. MBs of memory). Let $r^d(t)$, $r^a(t)$ and $r^w(t)$ the amount of demanded resources by all tasks in the grid (running or in queues), available resources, and resources demanded only by waiting tasks at time t. The amount of free resources $r^a(t)$ is given by the addition of the free resources in all physical hosts plus the available resources in all the virtual machines they run (i.e. unused by the tasks being processed at that moment). At all times $r^d(t) = r - r^a(t) + r^w(t)$. Also, often (but not always) if $r^w(t) > 0$ then $r^a(t) = 0$. Let $P_r(t)$ by the price at time t. Price is adapted periodically every s seconds as described in Equation 4.5 (let $t' = t + s$):

$$P_r(t') = \begin{cases} P_r(t) \times (1 + \frac{r^d(t') - r}{r})^{\frac{r^d(t')}{r^d(t)}} & \text{if } r^d(t') > 0 \wedge r^d(t) > 0 \\ P_r(t) \times (1 + \frac{r^d(t') - r}{r}) \& \text{if } r^d(t') > 0 \wedge r^d(t) = 0 & \\ P_r(t)/2 \& \text{if } r^d(t') = 0 & \end{cases} \tag{4.5}$$

The first case in Equation 4.5 aims to increase (decrease) the price depending on the amount of resources demanded over (below) the total available. The exponent modulates the adaptation depending on how sharp the change on resources demand has been since the last price recomputation. The second case is identical to the first one, to be applied when $r^d(t) = 0$. Finally, if the amount of resources used at t is 0, then the price is divided by 2.

Processing of Offers by Users

To simulate real users behavior is far from trivial. Usually, logs of task requests in real-world grids are helpful to reproduce a real load. However, they do not capture users reactions to situations where their requests could not be run due for example to resource contention, i.e. how they prioritize their tasks, how they choose between different execution options from different grids when available (usually grid usage logs refer to a single grid), how many tasks were outdated while waiting in the users queues, etc. Thus, instead of going through static load records, we chose to simulate users as dynamic entities that take planning decisions about their tasks.

Utility Function for Offer Selection

For each `REQUEST_FOR_OFFERS` issued by the user, the MA will send back a list of possible allocations (offers). The user will first filter those offers that cannot be accepted because of time or cost restrictions. Each task has a

deadline associated, so offers that would last beyond that deadline will not be considered by the user. Also, if the offer cost is greater than the user actual budget the offer is likewise rejected.

Then, the user must choose the best offer among the remaining ones. This depends on the user own priorities. Users will define a *utility function* $u : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$ to express the "utility" or worth of each task depending on the cost and time to execute it. The utility function is applied to the offers received, and the offer with the greatest utility value will be chosen. A possible utility function is $u(A^{\text{COST}}, A^{\text{TIME}}) = (A^{\text{COST}} \times A^{\text{TIME}})^{-1}$. If the user is only concerned about the time to execute the task regardless of its cost then the utility could be defined as $u(A^{\text{COST}}, A^{\text{TIME}}) = (A^{TIME})^{-1}$. The goal is to enable users to express their preferences, e.g. not to spend too much in a task (although it takes longer to run it) or to run the task as quickly as possible (even if it is expensive).

Negotiation Strategies

When some user requests execution offers for a task, she can face different situations:

- No offer is received, or all fail to meet the time and cost bounds imposed, that is the task deadline and the user budget. Then, the user can just label the task as "failed" or store it in a queue for later retrial.
- Some offers are suitable. Then, the best is chosen using the utility function as described in Section 4.3. The task is sent to the corresponding provider. In such case, still two things can happen.
  - The offer can still be honored by the provider, and so the task is executed.
  - The provider cannot fulfill the offer any more (e.g. due to a load burst after the offer was computed). Again, the user can then ignore the task or retry it.

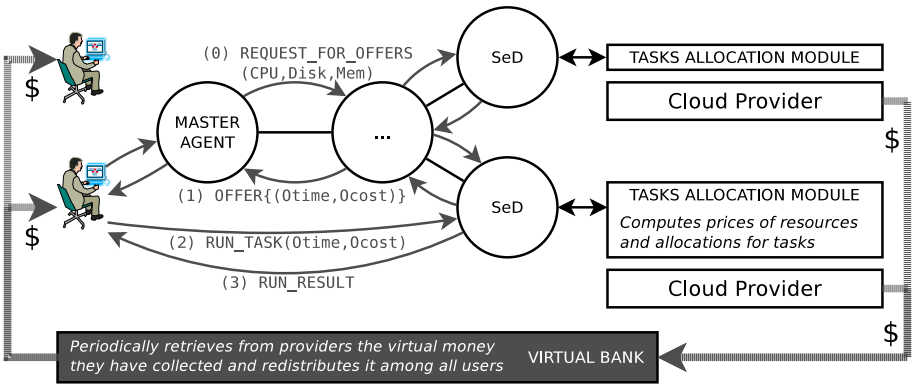
If failed tasks are stored, then users will prioritize them to set which tasks must be tried again first. Each task i will have a value I_i to represent its importance/priority. A basic strategy is to order tasks by importance so those with higher I_i values are retrieved from the queue and tried again first. We denote this strategy *Priority by Importance*. Yet, in real situations users will take also into account the "risk" of not being able to execute some task before its deadline expires. For example, if one task has a higher importance than another one, but there is still plenty of time to run the first while the second task's deadline is close, then the user can prefer to run the second task first. We propose the following mechanism for our simulations to set the tasks priorities: for each task i we compute the risk of not being able to run it on time as the coefficient between the task size (c_i) and the remaining time until the task deadline T_i (how the deadline is computed is explained in Section 4.4), which at time t is $T_i - t$. Then, this risk is multiplied by the task importance I_i to

get the priority of the task. So, the priority of task i at time t is computed as $c_i \times (T_i - t)^{-1} \times I_i$. The user queue that stores the tasks will order them by this value. We denote this other strategy *Priority by Risk*.

Users will periodically check if there are pending tasks stored, choose the one with the highest priority, and start the negotiation to run the task (see Section 4.3). This is done also every time that the user receives the result of another task.

Also, when the user has chosen the best offer for a certain task, she can store the other suitable offers instead of just discarding them. Thus, if the offer initially chosen is not valid anymore, then the user can try the other alternative offers before requesting new ones. In that case, providers can also send alternative offers when they are not able to run the task with the conditions of the original offer. These new alternative offers will be blended with the ones the user already stores for the task. As long as there are suitable alternative offers, the user will not send any new `REQUEST_FOR_OFFERS` message.

Figure 4.4 summarizes the main architectural elements presented in this section and their interactions as part of the market-oriented grid architecture proposed.



(0) A user requests offers to run a task. The REQUEST_FOR_OFFERS message (which carries the CPU, mem and disk requirements for the task) traverses the hierarchy until it reaches the TAMs.
(1) Each TAM builds offers for that task. All resulting offers are aggregated in a list as they go back to the Master Agent.
(2) The user chooses the most suitable offer depending on the the utility function used, and sends a petition to run the task to the corresponding SeD with the original offer conditions (time and cost)
(3) The execution result is sent to the user.

Figure 4.4: Main Architecture Elements and Interactions

4.4 Simulations Results

This section studies the best strategies for the user, and also two features of the system: adaptability to load changes and fairness.

Cloud Provider Setup

As explained in Section 4.2, each cloud provider has a catalog of types of VMs that it can instantiate to attend users requests. For the experiments presented here, providers are assigned a catalog defining three types of VMs that can be instantiated (these types closely correspond to the ones defined in EC2 catalog [2]). This catalog is described in Table 4.1. These types correspond to the set $\{V_1, \cdots V_n\}$ introduced in Section 4.3. The V_j^{COST} and V_j^{START} parameters for each type are set to minimal values so they do not interfere in the results outcome. Thus, the cost is set to 0, although in real settings administrators could choose to discourage the usage of certain VM types by assigning them higher prices. Also, the creation time is set to 1, which the authors know is fairly optimistic but will not introduce biases in the results: the goal is not to study which is the best/most chosen VM type, but the performance of the system as a whole.

VM Type	**PEs** ($V_j^C \times V_j^s$)	**Mem** (V_j^m)	**Disk** (V_j^d)
Normal	1 PE at 1 GHz	1.5 GBs	160 GBs
Large	4 PEs at 1 GHz	7.5 GBs	850 GBs
Extra-large	8 PEs at 1.5 GHz	15 GBs	1690 GBs

Table 4.1: Catalog of VMs Types Available.

Tasks processing requirements will be expressed in MIs, so we need to convert GHzs to MIPS. Such conversion is never accurate in any architecture, and it strongly depends on the software being run. But an approximate conversion can be 1GHz=6000MIPS [3].

Also, it is needed to define the amount of hardware resources of each cloud provider. Table 4.2 shows the amount of physical hosts and the resources of each one: memory, CPUs (with their processing speed) and disks (with their size). The hardware configuration of a standard cluster host is close to typical blade hardware settings [4], the configuration of hosts in the other cluster types are defined taken that one as reference.

Cluster	**Hosts**	**CPUs/Host** ($H_k^C \times H_k^s$)	**Mem/Host** (H_k^m)	**Disks/Host** ($H_k^D \times H_k^d$)
Small	4	2 CPUs at 2 GHzs	16 GBs	2 x 1 TBs
Standard	5	8 CPUs at 2 GHzs	64 GBs	8 x 2 TBs
Powerful	10	12 CPUs at 3 GHzs	96 GBs	12 x 2 TBs

Table 4.2: Hw Configurations for Cloud Providers in Experiments.

2. `http://aws.amazon.com/ec2/instance-types/`
3. See for example `http://en.wikipedia.org/wiki/Instructions_per_second`
4. See for example `http://www.sgi.com/products/servers/half_depth/2u_intel_2p.html`

Each provider updates the prices of resources every 50 seconds. All providers set initial prices as follows: $P_c = 100$ (per MI), $P_m = 1000$ (per MB), $P_c = 1000$ (per GB). Also, as commented in Section 4.3, each provider will check for *idle* VMs every 50 seconds. When a VM is found that has been idle for more than 600 seconds, the VM is turn off.

Nodes Setup

Nodes in the system (DIET nodes, TAMs, the Virtual Bank) have all the same bandwidth, 1Mb (which is quite conservative). All messages are 1Kb. The Virtual Bank retrieves money from providers and splits it among users every 1000 seconds. We assume messages processing time is negligible. This can be safely assumed even for messages that imply the computation of allocations for a task, as the process has little complexity and this complexity grows linearly with the amount of present VMs and the cluster size.

Users Behavior

We deem interesting to study which strategy is better suited for the user benefit before further research. That way, we can make a reasonable assumption about how users will behave in real situations, which we will apply in our later experiments.

The setting applied to study users strategies is as follows. We assume a scenario with two private clouds, each one getting resources from a *Small* cluster (see Table 4.2). Also, we assume 20 users, each one with 500 tasks to run. Time between the issuing of new tasks follows an exponential distribution. Initially, the average time between tasks is set to $\lambda^{-1} = 30$ s.

For each task i, its size c_i also follows an exponential distribution with average size 10^6 MI. Also, for each task it is necessary to know the maximum amount of time the user will accept to wait to get the task result. This time will be proportional to the task size and a new magnitude that we denote *urgency factor* f_i. This magnitude simulates the fact that not all results are equally critical for the user, so more important ones will get a higher f_i value. Then, if task i is created at t then the task deadline will be $T_i = t + c_i \times f_i$, that is, the time the user is ready to wait to obtain the result is proportional to both the size and importance of the task. In our experiments f_i is uniformly chosen from the following values: $\{0.001, 0.01, 0.1, 10, 100\}$. The memory m_i and disk d_i required are also uniformly chosen from different sets of values. In our experiments these were $\{10, 20, 30, 40, 50\}$ MBs for memory size and $\{10, 20, 30, 50, 60, 100\}$ GBs for disk size.

Finally, each task i importance (I_i), which is required to know its priority against other tasks (see Section 4.3), must be computed too. As in [86], we split tasks into two categories: *high importance tasks* and *low importance tasks*. Also as in [86], 20% of tasks will be of high importance. The importance of

tasks of high importance follows a normal distribution with mean 100 and standard deviation 50. The importance of tasks of low importance follows a normal distribution with mean 10 and standard deviation 5. Note that we do not relate importance with the maximum amount of currency the user will accept to pay for a task. As long as one offer's cost is not greater than the present user budget (minus the cost of the tasks already under execution, to ensure that the user never runs out of enough currency to pay an executed task), the offer can be accepted by the user.

The utility function u applied by users to choose the best offer is:

$$u(A^{\text{COST}}, A^{\text{TIME}}) = (A^{\text{COST}} \times A^{\text{TIME}})^{-1} \tag{4.6}$$

The initial price of processing one MI is 100. The initial price of one MB and of one GB of disk is the same, 10000. Each user has an initial budget of 10^9 currency units.

As explained in Sec. 4.3 users can follow two different strategies:

- Retry asking for offers for those tasks that do get an acceptable offer. Those tasks are stored in a queue ordered by priority. Each user will pick the first task in the queue and send a `REQUEST_FOR_OFFERS` message for that task every time the result of another task is received, and periodically at a certain rate. Experimentally we have seen that a low rate is enough to ensure that stored tasks do not have to wait long periods of time. We set this rate to 500 seconds.
- Keep alternative offers sent in the `OFFERS` message, i.e. those offers that were not chosen initially by the user. If the grid replies in the `RUN_RESULT` message that it failed to run the task, the user will check first whether there are still alternative offers for that task. If so, one of them will be chosen (using the user' utility function). Only when the user runs out of alternative offers a new `REQUEST_FOR_OFFERS` message will be sent.

Four sets of five experiments were run, each set corresponding to a different users' strategy. Results are shown in Fig 4.5 in four set of histograms, one histogram per experiment. Each histogram depicts the amount of tasks that were successful, that did not find a suitable offer due to budget limitations, etc. If we look at the first set of histograms, we see that the proportion of failed tasks is really high (due also in part to the high load). Almost all failed tasks are due to budget constraints: the user cannot afford paying for the task given the offers received ("*No Offer Fits Budget*"). A small set of tasks fail because no offer can run the task before the deadline is met ("*No Offer Fits Deadline*"). And another set of tasks fail because when the cloud provider is asked to run a task under certain cost and time conditions (extracted from the offer chosen by the user) those conditions cannot be fulfilled any more ("*No Allocation Possible For Offer*").

The second set of histograms show the results when the user applies the alternative offers. This policy does not bring any significant improvement in terms of the successful tasks rate.

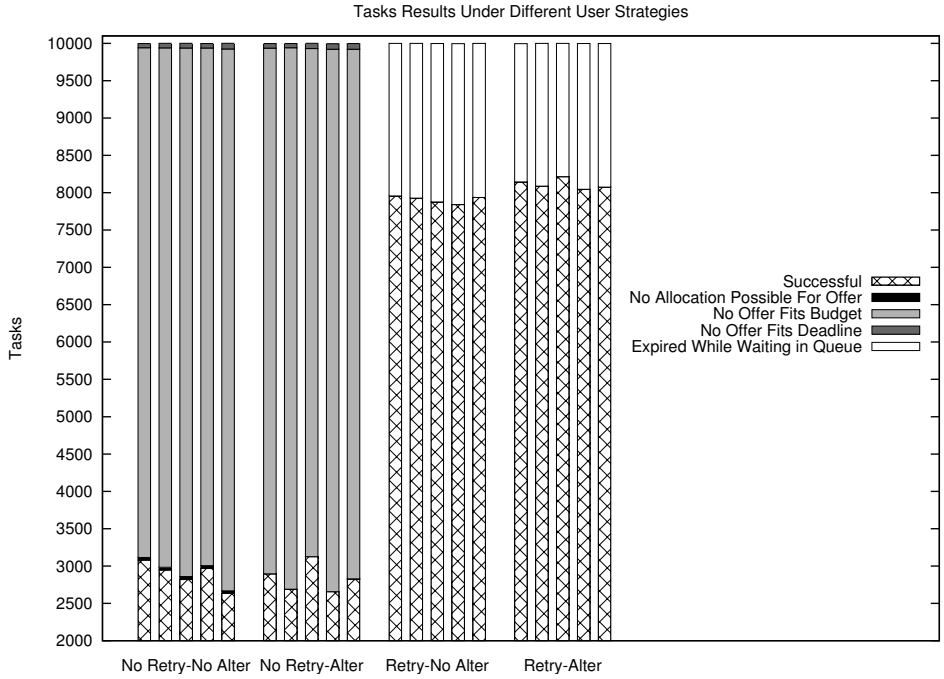


Figure 4.5: Impact of Retrials on Request for Offers and Usage of Alternative Offers.

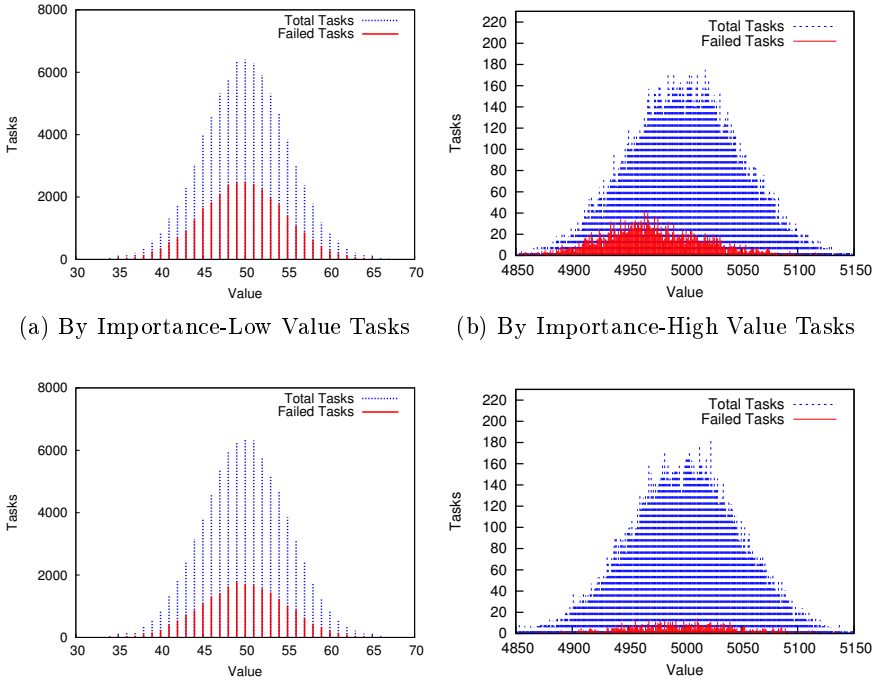
Much more useful is asking for new offers (i.e. as long as the task deadline can be met) as shown in the third and fourth group of histograms. Now each task is stored until either a provider runs it or the task expires. Using alternative offers slightly improves the rate of successful tasks. I.e., it is better to use alternative offers before performing requests for new ones. Another interesting metric to study is the sum of the values (importance) assigned to the executed and failed tasks, $\sum_{\texttt{Success}} I_i$ and $\sum_{\texttt{Failed}} I_i$, which should be maximized and minimized respectively. In both cases, the combination of using alternative offers and asking for new ones get the best results.

Our results show how simple user strategies such as storing tasks with no offers to retry them later have a significant positive effect on the final system outcome. Thus, it should be assumed that users will implement such strategies in real world situations. In the rest of the simulations presented users will use alternative offers if there are any. When no alternatives offer are available, then the user will store the task in her queue and resend `REQUEST_FOR_OFFERS` messages when the task is chosen again to be executed among the enqueued ones. This contrasts with typical approaches where failed tasks are simply discarded.

Tasks Priority

Also, we have studied the positive impact of the prioritization mechanism for stored tasks we propose (see Section 4.3), *Priority by Risk*, compared with the most straightforward *Priority by Importance* approach. In these experiments users will retry failed tasks and will use alternative offers. The setting of all parameters is similar to the one used in the previous experiments, but each user will run 5000 tasks, and load is changed by setting an average time between tasks of 50.

Figure 4.6 shows the results. Recall that the priority is represented by the value assigned to the task, and that tasks can be of two kinds, those with high value and those with low value (see Section 4.4). Figures 4.6(a) and 4.6(b) show the amount of failed and total tasks for both types, when users apply priority by importance. Figures 4.6(c) and 4.6(d) show the results when users order tasks by risk.



(a) By Importance-Low Value Tasks (b) By Importance-High Value Tasks

(c) By Risk-Low Value Tasks (d) By Risk-High Value Tasks

Figure 4.6: Failed and Total Tasks for Different Task Priority Mechanisms.

It can be observed from Figure 4.6 that, for both sets of values, when applying priority by risk the proportion of successful tasks over the total is greater (around 77% in total) that when applying priority by importance (when is only around 66%). Regarding the total value of the successful tasks ($\sum_{\texttt{Success}} I_i$), priority by risk yields an improvement of a 11% compared with the same value

when using priority by importance. On the other hand $\sum_{\texttt{Failed}} I_i$ is 62% lower when using priority by risk than when using priority by importance. Due to its better performance, *it can be assumed that users will prefer using the priority by risk strategy to order their tasks.* This will be assumed during the next experiments. Also, this was the policy applied in the experiments shown in previous Section 4.4 (we tested that using priority by importance does not alter the conclusion that retrying tasks and using alternative offers is the best choice).

About Users Behavior: Summing Up

The main goal of this section was to find out the strategies that bring the best outcome for users:

- When no suitable offer is found for some task, it is worth to store it for later retrieval instead of just discarding it, even it this strategy means that tasks will have to "compete" as user' budget is limited.
- It is better to use alternative offers before requesting new offers to the grid.
- The *Priority by Risk* strategy to order enqueued tasks results in less failed tasks.

Once these best strategies have been identified we can build a representative characterization of users. This is necessary to simulate market-based scenarios realistically, where users take decisions regarding tasks ordering, etc., instead of just discarding failed tasks.

System Adaptability

Once the most beneficial/likely strategies for users have been settled, it is time to study the behavior of the market-based system proposed. Two properties must be analyzed: *adaptability* and *fairness*. This section addresses the ability of a cloud provider to adapt to a changing load, while fairness is studied in the next section.

Recall that load can be controlled by setting the average time between tasks for each user, λ^{-1}, to different values. To check system adaptation an experiment will be run where λ^{-1} will be changed to check the performance under different loads. Thus, λ^{-1} is set initially ($t = 0$) to 75, to 7.5 at $t = 10000$, to 75 again at $t = 15000$ and finally to 750 at $t = 20000$. There will be 10 users with 2000 tasks each to run, and a single cloud provider on top of a *Powerful* cluster (see Table 4.2). The rest of the setup is similar to the previous experiments.

Results are shown in Figure 4.7. It depicts the amount of allocated and running PEs, along with the number of tasks waiting in VMs queues (the number of tasks in execution is of course equal to the number of running PEs). It can be observed that the system successfully reacts to the increased

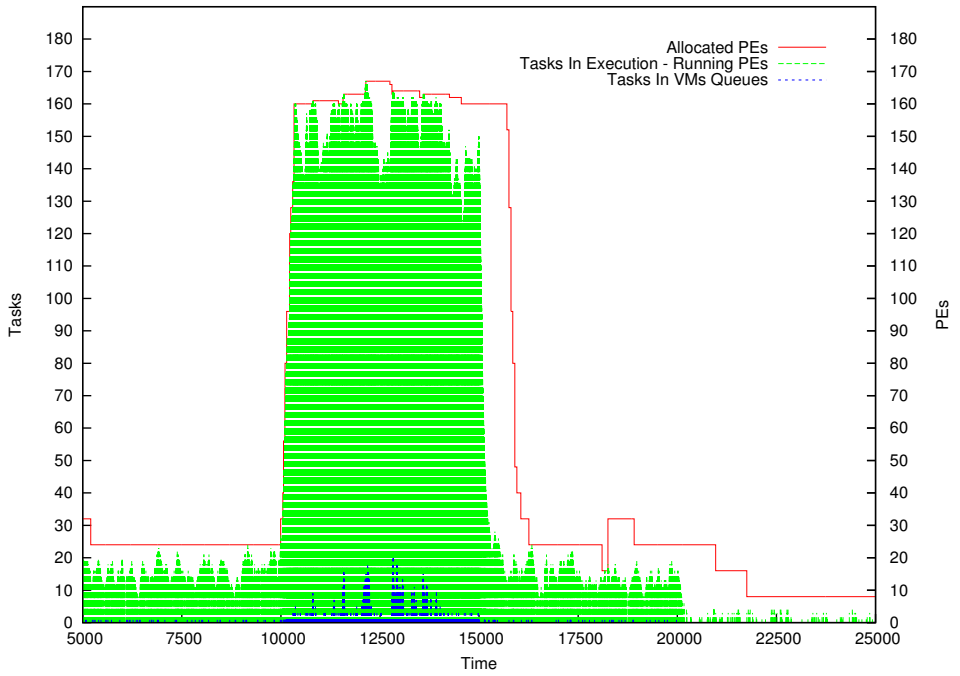


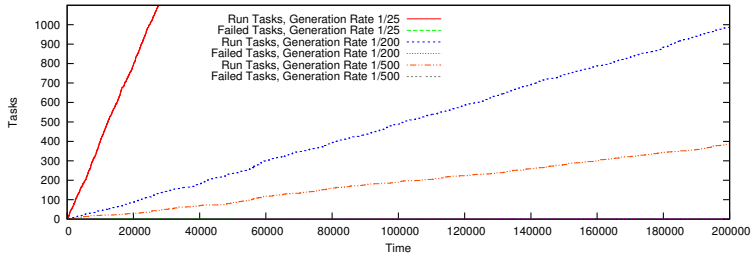
Figure 4.7: Adaptability: Allocated PEs under Varying Load.

demand of resources by allocating new Processing Elements in new VMs at $t = 10000$, where tasks will be run. Likewise, when the rate is decreased again at $t = 15000$ to the initial value the amount of running PEs falls abruptly, and so does the amount of PEs allocated later. Finally, when the rate shrinks at $t = 20000$ once again the system adapts and uses a minimum amount of resources. The reason because the changes on the amount of allocated PEs is abrupt is that users choose offers that will cause their tasks to be run in VMs of *Extra-large* type, which are faster (see the VM definitions in Table 4.1), but require more CPU resources.

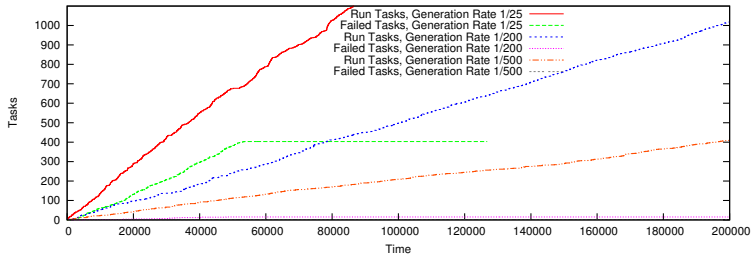
Fairness

Achieving *fairness* is the main goal of grid market based systems. Fairness refers to how resource usage is split among users by providers. No user should be able to require resources without limits as this could lead to resource shortage for others. But users should be able to run their tasks as long as they do not impact on other users throughput even if they have a higher resource demand. On the other hand, there should also be a limit so under high demand from several users, those demanding too many resources will not be able to get all of them, and so being "penalized" because of the tasks they will not be able to run.

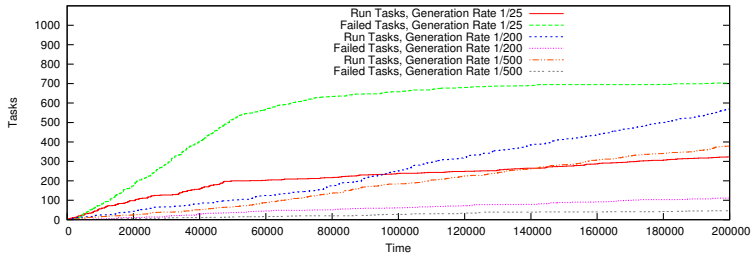
To test the fairness of our system three experiments were run, each one assigning to the (only) cloud one *Small* cluster, one *Standard* cluster and one *Powerful* cluster respectively. All experiments have 30 users, split into three sets of 10 users each. The average time between tasks for each set are 500, 200, and 25 seconds respectively. Each user will try to run 2000 tasks.



(a) Powerful Cluster



(b) Standard Cluster



(c) Small Cluster

Figure 4.8: Number of tasks successfully run or failed

Figure 4.8 shows the amount of tasks successfully run or failed (they expired before they could be executed) during the initial part of the experiment for three users, each one with a different task generation rate (users with the same rate show all very similar behavior), in the three different settings. In a powerful cluster (Fig. 4.8(a)) all users can run their tasks with no restriction, as there are enough resources to attend all petitions regardless of the resources they demand. But in a standard cluster (Fig. 4.8(b)) the cloud cannot serve all petitions, i.e. there is a certain resources shortage, and so some tasks fail.

Yet, this does not affect all users equally: users with low demand are not affected by this resource shortage and can run their tasks as in the previous setting. Also, users with medium demand are able to run almost all their tasks as before, and are only very lightly impacted by the lack of resources. Users with a high load, however, cannot run all their tasks anymore as they demand more resources than those the cloud will grant to any user. As a result, many tasks from users with high load will fail. Note that around $t = 50000$ users with high load will have already initiated all their tasks, so from that moment on they will only request to run the tasks enqueued. Finally, when using a small cluster (Fig. 4.8(c)), the same effect seen in the standard cluster is found again but in a higher degree. Users with small and medium load are only lightly affected, as their demand for resources can be attended by the cloud. In contrast, many tasks from users with a high generation rate fail (more in fact that the amount of run tasks). Note also how the rate of successful tasks from the users with medium rate is increased little after $t = 50000$. The reason is that users with high rate are only trying to run tasks stored in their queues, thus effectively lowering their need for resources.

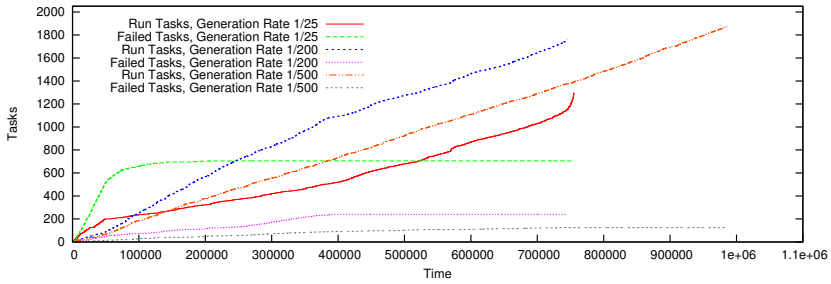


Figure 4.9: Fairness: Tasks Run and Failed in Small Cluster.

In Figure 4.9 we show the run and failed tasks for the small cluster until the end of the experiment. While users with low and medium demand keep the same tasks execution rate, users with high demand only very slowly are able to keep running tasks. After users with medium load have left the system (they have finished all their tasks) the demand of resources is so low that users can again run requests at high rate (recall that users check their tasks stored in their queues every 500 seconds and also every time that one task result is received, which allows for fast re-sending of tasks when resources are available).

These results lead to interesting conclusions. Users can try to run more tasks up to a certain rate as long as they do not interfere with other users. This is positive, as we do not force all users to work at the lowest rate. But if the resource demand by some user is too high then the system penalizes the user by not running many of her tasks –which will have to store them until many eventually expire (fail), the system does not subtract resources from other users needs.

4.5 Related Work

Despite being a recent technology, cloud computing has already raised the interest of the research community. Present research on IaaS systems is focusing on two main topics:

- Enabling the allocation of distributed resources on federated cloud systems. Open Cirrus [87], the *Sky Computing* [81] initiative or the EU Reservoir [88] joint research project are works oriented to the construction of such environments.
- Automatic scaling to adjust resources allocation to the demand. Automatic scaling is already implemented in some commercial solutions such as Amazon, where users configure scaling actions based on hardware state metrics such as CPU usage, etc. Other works [27, 89] propose more flexible scaling mechanisms based on service state in federated environments.

Regarding clouds and grids, there was some initial confusion about the differences and similarities between the two, although this was soon addressed by the community [11]. Later work [90] has further clarified the distinction between them, and analyzed how grids could evolve to benefit from the ideas introduced by the cloud (or the other way around, see for example [91]). In [92, 93] the authors present an architecture for the dynamic provision of resources to the virtual organizations (VO) of a grid. Part of the same team lead the StratusLab[5] project, a strong initiative in this regard. StratusLab is an EU joint research project that views clouds and grids as complementary technologies. StratusLab proposes three methods to integrate them:

- Deploy a grid site (based on EGEE[6] software) within a public cloud (Amazon's).
- Apply clouds for resource provisioning in grids.
- Add IaaS-like interfaces to existing grid services.

The second method lies close to the approach introduced in this work. But StratusLab goal is to virtualize an entire grid site for dynamic provision of worker nodes, while this proposal rather connects a grid system (DIET) to one or more clouds to get a supply of VMs in the same dynamic mode. StratusLab, on the other hand, does not apply economic models to ensure fair resource sharing.

Economy-based Grid Systems

Applying an economic approach in a grid system is hardly a new idea. Buyya et al. [75] already introduced a market based framework for grids, with an analysis of different market approaches such as auctions, posted price, tendering/contract-net, etc. In [94] the authors further discuss how economy

5. `http://stratuslab.eu`
6. `http://www.eu-egee.org`

can be applied to efficiently manage resources on grid environments and the advantages of such solutions (automatic regulation of supply and demand, scalability...).

It is not our intention to make a complete survey of all economy-based grid proposals (see [83, 76] for such overview of market-oriented grid systems). But in the remaining of this section we will comment how some of those works relate to two main aspects of the system proposed here, i.e. price computation and currency distribution.

Price Computation

Regarding proposals for resource price computation, Libra [95] and Libra+$ [86] suggest mechanisms for setting resource prices depending on demand. However they depend on some parameters whose values are arbitrary (must be tuned depending on the system and tasks). In contrast, our pricing solution does not requires such parameter values guessing.

G-commerce [96] proposes a formal pricing solution based on markets theory that aims to get the *equilibrium* prices of all resources. The equilibrium price is a market concept. In a market scenario, if the price of a commodity is low the demand will grow, in turn if the price of a commodity is high the demand will decrease. The equilibrium price in a market is the price reached when supply is equal to the demand. Unfortunately, such solution cannot be applied here. To compute the equilibrium price is required to have knowledge of the *global* demand of all resources in all SeDs, which we assume is not feasible in many scenarios.

On the other hand auction systems such as Bellagio [97], Mirage [84], and Tycoon [98] do not need providers to compute the price of resources. Users are the ones who must compete for the resources they require by bidding, so the resource is assigned to the highest bids. But then is the users who must decide policies to set the initial bid, how much increase the bid each time, what is the maximum bid, etc. So an auction approach is not easier to implement, it simply assigns more responsibility to users.

Other proposals such as FirstPrice [99], FirstReward [100] or FirstOpportunity [101] do not propose any resource price computation mechanism.

Distribution of Virtual Currency

Currency creation and circulation is an important concept in any market system. An option suggested in some works is to use real currency instead of virtual one [97], so users will take real care when demanding resources. Also, this solution frees the system from having to inject virtual currency and assign it to users. However, this approach has several inconveniences. For example, users with more economic means (e.g. better funding) will get more resources, leading to unfair situations. Also, in scientific environments users could be

reluctant to spare real currency for resources as they often work in other kind of settings were resources, even if scarce, are freely available. Thus, using *virtual currency*, created and distributed by the system seems to be the most feasible solution.

There are several options to inject and circulate virtual currency:

- Each provider (SeD) periodically reports to some central entity (Virtual Bank) about the value of all the resources it can deliver, taking into account their updated price. This would represent the 'wealth' of the system. This amount is then split and send to the users. However this solution would cause permanent inflation.
- The Virtual Bank periodically sends a certain amount to all users. Providers do not hoard money. But it is then necessary to decide how much assign to users, i.e. how much currency inject to the system. Arbitrary amounts could cause artificial inflation or deflation.
- The SeD/Cloud does not hoard neither drops the currency it gathers. Instead, it sends it to the Virtual Bank which will forward it to the users. This approach seems the more suitable.

In fact, the third approach is similar to the idea proposed in Mirage [84]. Also, to avoid hoarding by users, Mirage implements a *taxing* system that periodically reduces users budget so they do not tend to store currency too long. The currency obtained thorough this taxing system is then distributed back to user, as in the case of the clouds income. A similar mechanism could be used in our proposal.

Other works do not shed light to this problem, at least in the scenario proposed here. In Bellagio [97] users receive a budget proportional to the resources they provide, but this cannot be applied here as for the sake of flexibility DIET users are not assumed to be providers as well (although they could be). G-commerce [96] follows a similar approach to the one defined in the second point of the list above. They do not set any mechanism to decide how much to assign to users at each iteration. Tycoon [98] does not make any assumption, users "*... are funded at some regular rate. The system administrators set their income rate based on exogenously determined priorities*", or "*... bring resources ... must earn funds by enticing other users to pay for their resources*". FirstPrice [99] does not say anything about the subject. FirstReward [100] proponents explicitly state that they do not address how currency is injected or recycled. FirstProfit, FirstOpportunity [101] and Aggregate Utility [102] do not say anything about the subject (Aggregate Utility [102] in fact encourages using real currency).

4.6 Conclusions

The current chapter presents a proposal to combine grid and cloud systems through a market-based approach. Grids can benefit from clouds by requesting

and releasing resources from them, thus not being forced to have their own pool of resources. However, the grid system needs some criteria to know when to take resource scaling decisions. This criteria must of course take into account the demand induced by the tasks sent by the users.

By applying the pricing adaptation mechanism here proposed, grids can now scale resources automatically, while at the same time ensuring fairness in resource sharing. Future work will consist on implementing this on a real system: adapting DIET, programming the Virtual Bank and TAM, and connecting the TAM to some IaaS cloud provider, based for example on OpenNebula.

Part 3: Application management

Chapter 5

Running workflow-based applications in Cloud environments

Having explored the direction of resource management for IaaS Cloud platforms, we now focus on more concrete applications suitable for Cloud environments.

Many scientific applications are described through workflow structures. Due to the increasing level of parallelism offered by modern computing infrastructures, workflow applications now have to be composed not only of sequential programs, but also of parallel ones. Cloud platforms bring on-demand resource provisioning and pay-as-you-go payment charging. Then the execution of a workflow corresponds to a certain budget.

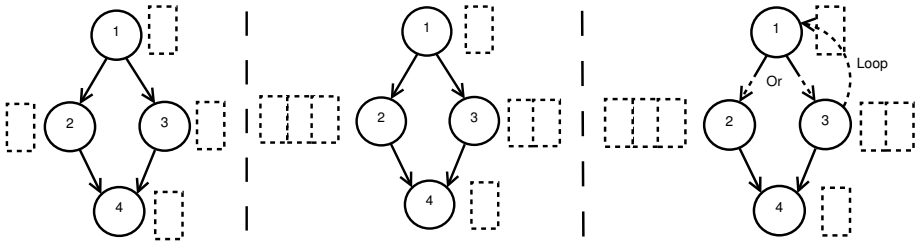
The current chapter focuses on running workflow applications in IaaS environments. We have chosen the non-deterministic worlkflow application model because it is the most general. We will address the problem of resource allocation for this type of workflow, given budget constraints. We will present a way of transforming the initial problem into sub-problems that have been studied before. We will also detail two new allocation algorithms that are capable of determining resource allocations under budget constraints and we present ways of using them to address the problem at hand.

Towards the end of this chapter, we will present a practical validation of the current work. We have tested our approach by using a scientific workflow application used for cosmological simulations.

5.1 Introduction

Many scientific applications from various disciplines are structured as *workflows*. Informally, a workflow can be seen as the composition of a set of basic operations that have to be performed on a given input set of data to produce the expected scientific result. The interest for workflows mainly comes from the need to build upon legacy codes that would be too costly to rewrite. Combining existing programs is also a way to lead to new results that would not have been found using each component alone. For years, such program composition was mainly done by hand by scientists, that had to run each program one after the other, manage the intermediate data, and deal with potentially tricky transitions between programs. The emergence of Grid Computing and the development of complex middleware components [103, 104, 105, 106, 107, 108, 109] automated this process.

The evolution of architectures with more parallelism available, the generalization of GPU, and the main memory becoming the new performance bottleneck, motivate a shift in the way scientific workflows are programmed and executed. A way to cope with these issues is to consider workflows composing not only sequential programs but also parallel ones. This allows for the simultaneous exploitation of both the task- and data-parallelisms exhibited by an application. It is thus a promising way toward the full exploitation of modern architectures. Each step of a workflow is then said to be *moldable* as the number of resources allocated to an operation is determined at scheduling time. Such workflows are also called Parallel Task Graphs (PTGs), as depicted in Figure 5.1(b).



(a) Classic (b) Parallel Task Graph (PTG) (c) Functional

Figure 5.1: Workflow types

In practice, some applications cannot be modeled by classical workflows or PTGs. For such applications the models are augmented with special semantics that allow for exclusive diverging control flows or repetitive flows. This leads to a new structure called a *non-deterministic* or *functional* workflow, as depicted in Figure 5.1(c). For instance, we can consider the problem of gene identification by promoter analysis [110, 111] as described in [107], or the GENIE (Grid ENabled Integrated Earth) project that aims at simulating the long

term evolution of the Earth's climate [112].

The scalability and "pay for what you use" billing model inherent in Cloud platforms make them a good candidate for running workflow applications. While the elasticity provided by IaaS Clouds gives way to more dynamic application models, it also raises new issues from a scheduling point of view. An execution now corresponds to a certain budget, that imposes certain constraints on the scheduling process. In this chapter we detail a first step to address this scheduling problem in the case of non-deterministic workflows. Our main contribution is the design of an original allocation strategy for non-deterministic workflows under budget constraints. We target a typical IaaS Cloud and adapt some existing scheduling strategies to the specifics of such an environment in terms of resource allocation and pricing.

5.2 Related Work

The problem of scheduling workflows has been widely studied by the aforementioned workflow management systems. Traditional workflows consists in a deterministic DAG structure whose nodes represent compute tasks and edges represent precedence and flow constraints between tasks. Some workflow managers support conditional branches and loops [113], but neither of them target elastic platforms such as IaaS Clouds nor address their implications.

Several algorithms have been proposed to schedule PTGs, *i.e.,* deterministic workflows made of moldable tasks, on various non-elastic platforms. Most of them decompose the scheduling in two phases: (i) determine a resource allocation for each task; and (ii) map the allocated tasks on the compute resources. Among the existing algorithms, we based the current work on the CPA [114] and biCPA [115] algorithms. We refer the reader to [115] for details and references on other scheduling algorithms.

The flexibility provided by elastic resource allocations offers great improvement opportunities as shown by the increasing body of work on resource management for elastic platforms. In [116], the authors give a proof of concept for a chemistry-inspired scientific workflow management system. The chemical programming paradigm is a nature-inspired approach for autonomous service coordination [117]. Theirs results make this approach encouraging, but still less performing than traditional workflow management systems. In contrast to the current work, they do not aim at conditional workflows or budget constraints. An approach to schedule workflows on elastic platforms under budget constraints is given in [118], but is limited to workflows without any conditional structure.

5.3 Problem Statement

Platform and Application Models

An IaaS Cloud can be seen as a virtually infinite set of resources that are reserved and instantiated by users according to their needs. We consider that users have access to a *catalog* that comprises different types of resources, each corresponding to a unique combination of characteristics. Such a catalog is inspired by the offers of major providers such as Amazon EC2. A resource, or virtual machine instance, vm, can be described by:

- A number of equivalent virtual CPUs, $nCPU$. The number of virtual CPUs does not correspond to the number of physical CPUs in the instance, but allows users to easily compare the relative performance of different instances;
- A computing speed per virtual CPU, s. This corresponds to the amount of computing operations a single CPU can process per second.
- A monetary cost per running hour, $cost$, expressed in a currency-independent manner. As most providers do, we also consider that each started hour has to be entirely paid even when not fully used. This cost is then proportional to the number of full hours the instance runs since it becomes usable.

In our study, we consider that every virtual CPU in the IaaS Cloud have the same computing speed. Instances of the same type are then homogeneous, while the complete catalog is a heterogeneous set of resources. Thus, we do not include this speed in our formal definition of the catalog $\mathcal{C}$ that is

$$\mathcal{C} = \{vm_i = (nCPU_i, cost_i) | i \geq 1\}.$$

We also consider that a virtual CPU can communicate with several other virtual CPUs simultaneously under the *bounded multi-port* model. All the concurrent communication flows share the bandwidth of the communication link that connects this CPU to the remaining of the IaaS Cloud.

Our workflow model is inspired by previous work [112, 119]. We define a non-deterministic workflow as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i | i = 1, \ldots, V\}$ is a set of V vertexes and $\mathcal{E} = \{e_{i,j} | (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of E edges representing precedence and flow constraints between tasks. Without loss of generality we assume that $\mathcal{G}$ has a single entry task and a single exit task. The vertexes in $\mathcal{V}$ can be of different types. A **Task** node represents a (potentially parallel) computation. Such nodes can have any number of predecessors, *i.e.,* tasks that have to complete before the execution of this task can start, and any number of successors, *i.e.,* tasks that wait for the completion of this task to proceed. Traditional deterministic workflows are made of task nodes only. The relations between a task node and its predecessors and successors can be represented by control structures, that we respectively denote by **AND-join** and **AND-split** transitions.

Task nodes are moldable and can be executed on any numbers of virtual resource instances. We denote by $Alloc(v)$ the set of instances allocated to task v for its execution. The total number of virtual CPUs in this set is then: $p(v) = \sum_j nCPU_j | vm_j \in Alloc(v)$. It allows us to estimate $T(v, Alloc(v))$ the execution time of task v if it were to be executed on a given allocation. In practice, this time can be measured via benchmarking for several allocations, or it can be calculated via a performance model. In this work, we rely on Amdahl's law. This model claims that the speedup of a parallel application is limited by its strictly serial part α. The execution time of a task is given by

$$T(v, Alloc(v)) = \left(\alpha + \frac{(1-\alpha)}{p(v)}\right) \times T(v, 1),$$

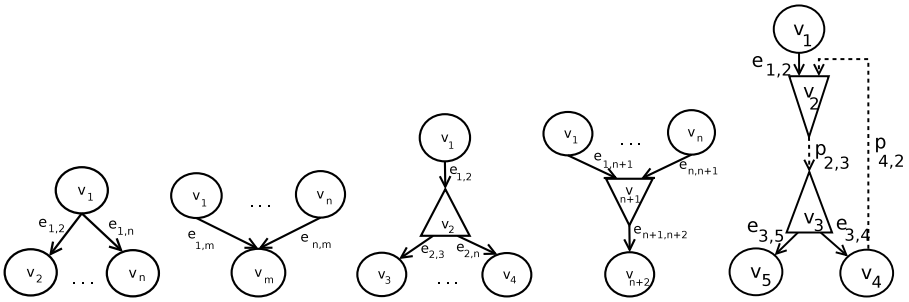
where $T(v, 1)$ is the time needed to execute task v on a single virtual CPU. The overall execution time of $\mathcal{G}$, or *makespan*, is defined as the time between the beginning of $\mathcal{G}$'s entry task and the completion of $\mathcal{G}$'s exit task. The total number of CPUs needed to achieve this makespan is $p = \sum_{i=1}^{V} p(v_i)$.

In our model, we consider that each edge $e_{i,j} \in \mathcal{E}$ has a weight, which is the amount of data, in bytes, that task v_i must send to task v_j. We do not impose any type of restrictions for inter-task communications. The actual communication time may be higher than the time needed to transfer the data, as the source and destination tasks might be mapped to a different number of virtual resources, which might cause an overhead.

To model the non-deterministic behavior of the considered workflows, we add the following control nodes to our model. A **OR-split** node has a single predecessor and any number of successors, that represent mutually-exclusive branches of the workflow. When the workflow execution reaches an OR-split node, it continues through only one of the successors. The decision of which successor to run is taken at runtime. Then in the scheduling phase, all the sub-workflows deriving from an OR-split node have to be considered as equally potential execution paths. Conversely an **OR-join** node has any number of predecessors and a single successor. If any of the parent sub-workflows reaches this node, the execution continues with the successor.

Finally, our model of non-deterministic workflows can also include **Cycle** constructs. This is an edge joining an OR-split node and one OR-join ancestor. A cycle must contain at least one OR-join node to prevent deadlocks. Figure 5.2 gives a graphical representation of these control nodes and constructs.

Figure 5.2(e) is a simple representation of the Cycle construct. $p_{2,3}$ and $p_{4,2}$ are not edges of the workflow, but paths leading from v_2 to v_3 and from v_4 to v_2 respectively. These paths are a weak constraint that ensure the creation of a cycle in the graph, in combination with the OR-join and OR-split nodes v_2 and v_4. However, a Cycle can contain any number of OR-split or OR-join nodes and even an unbound number of edges leading to other parts of the workflow.



(a) AND-split (b) AND-join (c) OR-split (d) OR-join (e) Cycle

Figure 5.2: Non-deterministic workflow control nodes and constructs.

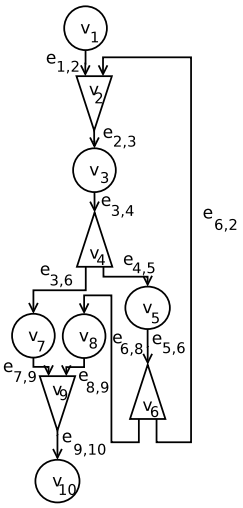


Figure 5.3: A more complex workflow example.

We give a more complex example of functional workflow in Figure 5.3, in which the path deriving from the edge $e_{6,2}$ comprises a OR-split node (v_4). This implies that the Cycle construct does not determine the number of iterations of the cycle path by itself, as in a loop construct for instance. Decisions taken at runtime for v_4 may make the execution flow exit the cycle before reaching v_6.

Metrics and Problem Statement

We consider the problem of determining allocations for a single non-deterministic workflow on an IaaS Cloud. It amounts to allocate resource instances to the tasks of this workflow so as to minimize its makespan while respecting a given budget constraint. Targeting an IaaS Cloud indeed implies such a constraint, as using more resources is likely to lead to smaller makespans but also in-

creases the monetary cost associated to the execution of the workflow. An additional issue is to deal with the non-determinism of the considered workflows. At scheduling time, all the possible execution paths have to be considered. But at runtime, some sub-workflows will not be executed, due to the OR-split construct, while others may be executed several times, due to the Cycle construct. This raises some concerns relative to the respect of the budget constraint. Our approach is to decompose the workflow into a set of deterministic sub-workflows with non-deterministic transitions between them. Then, we fall back to the well studied problem of determining allocations for multiple Parallel Task Graphs (PTGs).

In the following we define the makespan as $C = \max_i C(v_i)$ where $C(v_i)$ is the finish time of task v_i. We denote by B the budget allocated to the execution of the original workflow and by B^i the budget allocated to the i^{th} sub-workflow. These budgets are expressed in a currency-independent manner.

Finally, $Cost^i$ is the cost of a schedule $\mathcal{S}^i$ built for the i^{th} sub-workflow on a dedicated IaaS Cloud. It is defined as the sum of the costs of all the resource instances used during the schedule. Due to the pricing model, we consider all started hour as fully paid.

$$Cost^i = \sum_{\forall vm_j \in \mathcal{S}^i} \lceil T_{end_j} - T_{start_j} \rceil \times cost_j,$$

where T_{start_j} is the time when vm_j is launched and $T_{end_{i=j}}$ the time when this resource instance is stopped.

5.4 Allocating a Non-Deterministic Workflow

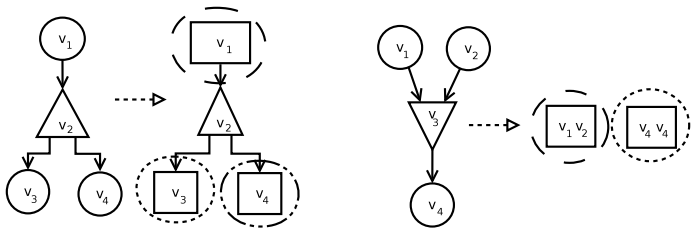
Our algorithm is decomposed in three steps: (i) Split the non-deterministic workflow into a set of deterministic PTGs; (ii) Divide the budget among the resulting PTGs and (iii) Determine allocations for each PTG. The following sections details these steps. We also discuss some runtime issues.

Splitting the Workflow

Transforming a non-deterministic workflow into a set of PTGs amounts to extract all the sequences of task nodes without any non-deterministic construct. A similar approach to decompose a workflow into smaller parts is taken by DagMan [103]. It allows users to split nested workflows by hand and is considered as part of the workflow definition.

Figure 5.4 shows how we extract sub-workflows in presence of OR-split and OR-join nodes. For the sake of simplicity we have omitted edge labels in this figure. These control nodes define boundaries between sub-workflows and do not belong to any of them. An OR-split node leads to $n + 1$ sub-workflows, one ending with the predecessor of the node and n starting with

each of the successors of the OR-split node. If two OR-split nodes share a common successor, we consider the two resulting sub-workflows as different, even though they have the same structure. Indeed these sub-workflows come from different non-deterministic transitions and therefore different contexts.



(a) OR-split (b) OR-join

Figure 5.4: Extracting sub-workflows from OR-split and OR-join nodes.

Splitting a workflow that contains an OR-join node can lead to as many sub-workflows as there were predecessor sub-workflows of the OR-join node. The successors of the OR-join node are replicated for all of its predecessors, including the ones that are part of the same sub-workflow. It is worth noting that OR-join nodes do not actually lead to the creation of new sub-workflows since they do not have a non-deterministic nature and therefore they do not lead to non-deterministic transitions. What they actually do is preserve the number of sub-workflows that they have from their inwards transitions.

Extracting sub-workflows from a Cycle node is more complex as shown in Figure 5.5. Here we extract three sub-workflows. Two of them include an instance of task v_3. One comes as a result of the execution of task v_1, while the other derives from following the cycle branch. Task v_5 is then the predecessor of this second instance.

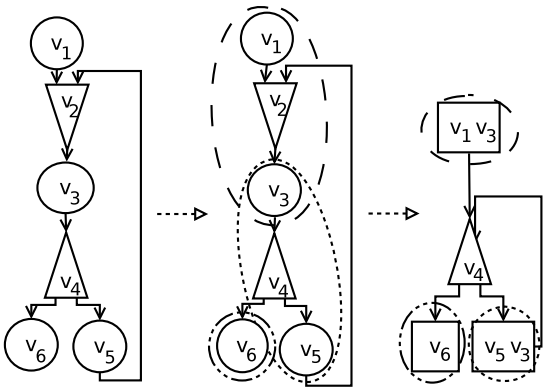


Figure 5.5: Extracting sub-workflows with regard to a Cycle construct.

Figure 5.6 details how we decompose the complex workflow given in Figure 5.3. It is worth noting that a Cycle constructs does not necessarily correspond to a unique sub-workflow. In this example, the Cycle $e_{6,2}$ is split into two different sub-workflows v_3 and v_5 that both belong to the cycle path. This will have an impact on budget distribution as detailed in the next section.

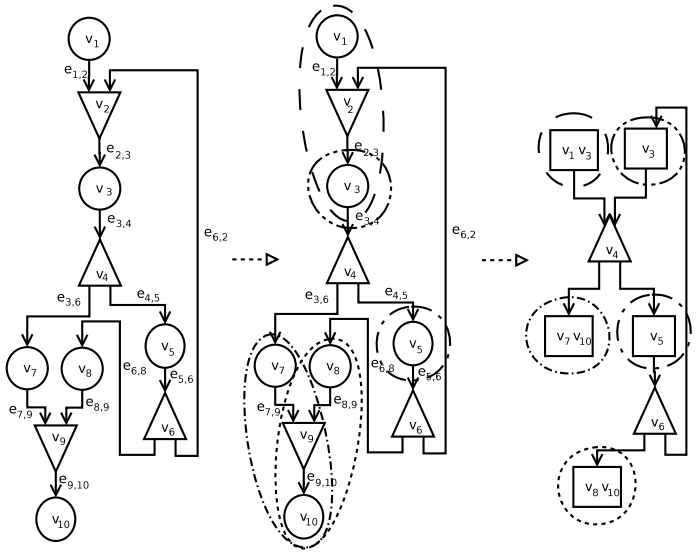


Figure 5.6: Extracting sub-workflows from a more complex workflow.

Distributing Budget to Sub-Workflows

As we target an IaaS Cloud, we have to decide how much money we can dedicate to each sub-workflow obtained after the split of the original application to determine its resource allocation. Because of the non-deterministic transitions between sub-workflows, we first have to estimate the odds to execute each of them. Moreover, as cycle paths may comprise several sub-workflows, we have to estimate how many times each sub-workflow could be executed at runtime.

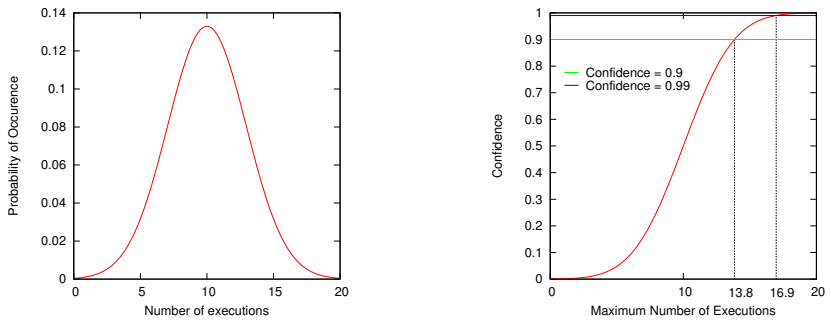
Each sub-workflow, apart from the entry sub-workflow, has one and only one non-deterministic transition that triggers its execution. This is the transition from its parent OR-split node to its starting task. We can therefore conclude that the number of executions of a sub-workflow is described completely by the number of transitions of the edge connecting its parent OR-split to its start node. We model this behavior by considering that the number of transitions of each outwards edge of an OR-split, and therefore the number of executions of a sub-workflow $\mathcal{G}^i$ is described by a random variable according to a distinct *normal distribution* D^i. Moreover we use a parameter that express the *Confidence* the algorithm has that a given sub-workflow will not be exe-

cuted more than a certain number of time. This parameter takes its value in the $[0, 1)$ interval. This way, we aim at guaranteeing that the whole workflow will be able to finish while respecting the budget constraint. More formally, the expected maximum number of executions of a $\mathcal{G}^i$ is

$$nExec^i \leftarrow CDF^{-1}(D^i)(Confidence)$$

where $CDF^{-1}(D^i)$ is the reverse *Cumulative Distribution Function* (CDF) for distribution D^i. Figures 5.7(a) and 5.7(b) illustrate our approach.

Figure 5.7(a) displays the normal distribution $\mathcal{N}(10, 3)$ of a random variable. The distribution median is $\mu = 10$ and its variance is $\sigma^2 = 3$. In our context, it correspond to the probability that the sub-workflow execution modeled by this random variable is repeated a certain number of times.



(a) Normal distribution $\mathcal{N}(10, 3)$.

(b) CDF of $\mathcal{N}(10, 3)$.

Figure 5.7: Estimation of the maximum number of executions of a sub-workflow, described by a normal distribution, with a certain confidence.

Figure 5.7(b) shows the CDF of this distribution. It allows to estimate, for a given confidence, how many time we will repeat the considered sub-workflow *at most*. For instance, with a confidence of 0.9 (or 90%), this sub-workflow is likely to not be executed more than 13.8 times. With a higher confidence of 0.99 (or 99%), this estimation raises up to 16.9 executions at most.

This estimation of the number of times a sub-workflow could be executed is not the only metric to consider to distribute the budget as best as possible. Indeed, it may be more important to give an important share of the budget to a sub-workflow with many time-consuming tasks that may be executed only once than to a sub-workflow with a few short tasks that is repeated several times. To find a good balance, we include the contribution of a sub-workflow with regard to the whole application in the determination of the budget distribution. We determine the contribution ω^i of sub-workflow $\mathcal{G}^i$ as the sum of the average execution times of its tasks multiplied by the number of times this sub-workflow could be executed. As the target platform is virtually infinite, we compute the average execution time of a task over the set of resource instances in the catalog

$\mathcal{C}$. This allows us to take the speedup model into account, while reasoning on a finite set of possible resource allocations. We denote by ω^* the sum of the contribution made by all the sub-workflows.

Algorithm 3 Share_Budget($B, \mathcal{G}, Confidence$)

1: $\omega^* \leftarrow 0$
2: **for all** $\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i) \subseteq \mathcal{G}$ **do**
3: $\quad nExec^i \leftarrow CDF^{-1}(D^i, Confidence)$
4: $\quad \omega^i \leftarrow \sum_{v_j \in \mathcal{V}^i} \left(\frac{1}{|\mathcal{C}|} \sum_{vm_k \in \mathcal{C}} T(v_j, vm_k) \right) \times nExec^i$
5: $\quad \omega^* \leftarrow \omega^* + \omega^i$
6: **end for**
7: **for all** $\mathcal{G}^i \subseteq G$ **do**
8: $\quad B^i \leftarrow B \times \frac{\omega^i}{\omega^*} \times \frac{1}{nExec^i}$
9: **end for**

Algorithm 3 describes how we distribute the global budget B among the sub-workflows. Once we have estimated the number of execution of each workflow and its relative contribution, the budget B^i assigned to one iteration of the sub-workflow $\mathcal{G}^i$ is simply obtained by multiplying the global budget by the ratio ω^i/ω^* and dividing by the estimated number of executions of the workflow $nExec^i$ (line 8).

Determining PTG allocations

Once the non-deterministic workflow has been split into a set of deterministic sub-workflows, and that a budget has been assigned to each sub-workflow, our algorithm has to find an allocation for each of them. In other words, we have to determine which combination of virtual instances from the resource catalog leads to the best compromise between the reduction of the makespan and the monetary cost for each sub-workflow, *i.e.,* a PTG. We base our work upon the allocation procedures of seminal two-step algorithms, named CPA [114] and biCPA [115], that were designed to schedule PTGs on homogeneous commodity clusters. We adapt these procedures to the specifics of IaaS Cloud platforms.

As the biCPA algorithm is an improvement of the original CPA algorithm, we start by briefly explaining the common principle of their respective allocation procedures. It starts by allocating one CPU to each task in the PTG. Then it iterates to allocate one extra CPU to the task that belongs to the critical path of the application and benefits the most of it. The procedure stops when the average work T_A becomes greater than the length of the critical path T_{CP}. The definition of the average work used by the CPA algorithm was

$$T_A = \frac{1}{P} \sum_{i=1}^{|\mathcal{V}^i|} W(v_i),$$

where $W(v_i)$ is the work associated to task v_i, *i.e.,* the product of its execution time by the number of CPUs in its allocation, and P the total number of CPUs in the target compute cluster. In biCPA, the value of P is iterated over from 1 to the size of the target compute cluster and its semantics is changed to represent the total number of CPUs that any task can have allocated to it.

The definition of the length of the critical path was

$$T_{CP} = max_i BL(v_i)$$

where $BL(v_i)$ represents the *bottom level* of task v_i *i.e.,* its distance until the end of the application. For the current work we keep this definition for T_{CP}.

On an IaaS Cloud, the size of the target platform is virtually infinite. Then it is impossible to use such a definition that includes a total number of CPUs. Instead, we propose to reason in terms of budget and average cost of an allocation. Moreover, the pricing model implies that each started hour is paid, even though the application has finished its execution. Then, some *spare time* may remain on a virtual resource instance at the end of an execution.

When building an allocation, we don't know yet in which order the tasks will be executed. Then we cannot make any strong assumption about reusing spare time left behind after executing a task. As we aim at building an allocation for $\mathcal{G}^i$ that costs less than B^i, a conservative option would be to consider that this spare time is never used. This corresponds to always overestimating the cost of the execution of a task by rounding its execution time up to the end of the last started hour. Then we define this cost as

$$cost(v_i) = \lceil T(v_i, Alloc(v_i)) \rceil \times \sum_{vm_j \in Alloc(v_i)} cost_j.$$

This, in turn, leads us to a first adapted version of the definition of T_A

$$T_A^{over} = \frac{1}{B'} \times \sum_{j=1}^{|\mathcal{V}^i|} \left(T(v_j, Alloc(v_j)) \times cost(v_j) \right),$$

in which we sum the time-cost area of each task, that is its execution time multiplied by its overestimated monetary cost. We then average the obtained value over the allowed budget B'. $B' \leq B^i$ is the maximum budget that any task can use in order to run. It is different from the maximum budget for the whole allocation, B^i, which we will use as the stop condition for the allocation algorithm.

Overestimating the costs this way allows us to guarantee that the produced allocation will not exceed the allowed budget. However, it may have a bad impact on makespan depending on how much spare time is lost. Consider a simple example to illustrate this. We want to build an allocation for a chain of 10 tasks with a budget of 10 units. One hour on a virtual instance costs 1 unit. Unfortunately each task runs for only ten minutes. With the above formula, each task will be allocated only one virtual instance as the budget limit is already reached. However, it is likely that, once scheduled, all the tasks will reuse the same instance for a total running time of 100 minutes and a cost of two units! A tighter estimation of the cost may have allowed each task to run for five minutes on two virtual CPUs, leading to a makespan divided by two for the same cost.

To hinder the effect of this overestimation, we can assume that the spare time left by each task has one in two chance to be reused by another task. The risk inherent to such an assumption is that we do not anymore have a strong guarantee that the resulting allocation will fall short of the allowed budget once scheduled. Nevertheless, we modify the definition of $cost(v_i)$ as follows:

$$cost(v_i) = \frac{\lceil T(v_i, Alloc(v_i)) \rceil + T(v_i, Alloc(v_i))}{2} \times \sum_{vm_j \in Alloc(v_i)} cost_j.$$

The definition of T_A^{over} remains unchanged. However, in the remaining of this chapter, it relies on this second definition of $cost(v_i)$.

Based on this definition, we propose a first allocation procedure detailed by Algorithm 4. This procedure determine one allocation for each task in the considered sub-workflow while trying to find a good compromise between the length of the critical path (hence the completion time) and the average time-cost area as defined by T_A^{over}.

Since the purpose of this algorithm is to determine only one allocation, we cannot simply iterate B' from 0 to B^i. We need to estimate the value of B' such that the values of T_A^{over} and T_{CP} will reach a trade-off at the end of the allocation.

At convergence time, the two values are equal. B' is the maximum cost of running any single task at convergence time and B^i is the total cost of the allocation. As a heuristic to determine B' we assume that the proportion between the total work area and the maximum work area is constant. We can therefore calculate these areas for an initial iteration and determine the value of B' when convergence occurs.

$$\frac{B'}{B^i} = \frac{\sum_{j=1}^{|\mathcal{V}^i|} \left(T(v_j, Alloc^{init}(v_j)) \times cost^{init}(v_j) \right)}{T_{CP}^{init} \times \sum_{j=1}^{|\mathcal{V}^i|} cost^{init}(v_j)}$$

$Alloc^{init}$ represents the initial allocation in which we give an instance of the smallest type to every task.

Algorithm 4 Eager-allocate($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), B^i$)

1: **for all** $v \in \mathcal{V}^i$ **do**
2: $\quad Alloc(v) \leftarrow \{\min_{vm_i \in \mathcal{C}} CPU_i\}$
3: **end for**
4: Compute B'
5: **while** $T_{CP} > T_A^{over} \cap \sum_{j=1}^{|\mathcal{V}^i|} cost(v_j) \leq B^i$ **do**
6: $\quad$ **for all** $v_i \in$ Critical Path **do**
7: $\quad\quad$ Determine $Alloc'(v_i)$ such that $p'(v_i) = p(v_i) + 1$
8: $\quad\quad Gain(v_i) \leftarrow \frac{T(v_i, Alloc(v_i))}{p(v_i)} - \frac{T(v_i, Alloc'(v_i))}{p'(v_i)}$
9: $\quad$ **end for**
10: $\quad$ Select v such that $Gain(v)$ is maximal
11: $\quad Alloc(v) \leftarrow Alloc'(v)$
12: $\quad$ Update T_A^{over} and T_{CP}
13: **end while**

Each task's allocation set is initialized with the number of CPUs of the smallest virtual instance in the catalog. Then, we determine which task belonging to the critical path would benefit the most from an extra virtual CPU, and increase the allocation of this task. We iterate this process until we find a compromise between makespan reduction and estimated cost increase. Note that the determination of $Alloc'(v_i)$ (line 7) may mean either adding a new instance with one virtual CPU to the set of resource instances already composing the allocation, or switching to another type of instance from the catalog.

Figure 5.8 shows an evolution of the values of T_A^{over} and T_{CP} across the allocation process, for a budget limit of 10 units. We have used a resource catalog inspired by Amazon EC2's catalog, which can be found in Table 5.1. There is a single point of convergence between the two, which represents a good trade-off between the two values. The allocation process stops if this point is reached or if the estimated costs of the allocation exceeds the budget limit. In the current example, a trade-off is reached after 57 iterations.

In practice it is only worth continuing the allocation process if the value if T_{CP} continues to decrease. We have added a supplementary stop condition that is triggered if the value of T_{CP} does not decrease more than one second. We call this the T_{CP} *cut-off*.

As this first procedure may produce allocations that do not respect the budget constraint, we propose an alternate approach based on a similar principle as that used by the biCPA algorithm [115]. Instead of just considering the allocation that is eventually obtained when the trade-off between the length of the critical path and the average cost is reached, we keep track of intermediate allocations build as if the allowed budget was smaller. Once all these candidate allocations are determined, we build a schedule for each of them on a dedicated platform to obtain a precise estimation of their makespan they

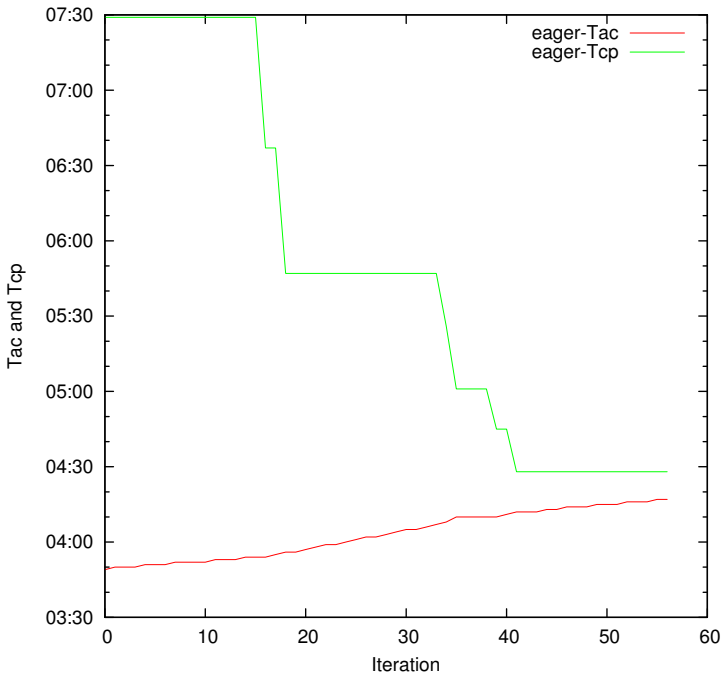


Figure 5.8: The evolution of T_A^{over} and T_{CP}

.

achieve and at which cost. Then it is possible to choose the "best" allocation that leads to the smallest makespan for the allowed budget.

In this second procedure, we can rely on a tighter definition of the average time-cost area that does not take spare time into account. Indeed, if some spare time exists, it will be reused (or not) when the schedule is built. Since we select the final allocation based on the resulting scheduling, we do not have to consider spare time in the first step. To some extent, it amounts to underestimate the cost of the execution of a task. Our second allocation procedure will then rely on T_A^{under}, defined as

$$T_A^{under} = \frac{1}{B'} \times \sum_{j=1}^{|\mathcal{V}^i|} (T(v_j, Alloc(v_j)) \times cost_{under}(v_j))$$

This definition differs from that of T_A^{over} by the use of

$$cost_{under}(v_j) = T(v_j, Alloc(v_j)) \times \sum_{vm_k \in Alloc(v_j)} cost_k$$

that includes the exact estimation of execution time of v_j and of a new variable B' instead of the allowed budget B^i. This parameter allows us to

mimic the variable size of the cluster used by the biCPA algorithm, and represents the maximum budget allowed to determine any one task's allocation. Its value will grow along with the allocation procedure, starting from the largest cost of running any task from the initial allocation and up to B^i. The use of B' has a direct impact on the computation of the average time-cost area and will lead to several intermediate trade-offs and corresponding allocations. We refer the reader to [115] for the motivations and benefits of this approach.

Algorithm 5 Deferred-allocate($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), B^i$)

1: **for all** $v \in \mathcal{V}^i$ **do**
2: $Alloc(v) \leftarrow \{\min_{vm_i \in C} CPU_i\}$
3: **end for**
4: $k \leftarrow 0$
5: $B' \leftarrow \max_{v \in \mathcal{V}^i} cost_{under}(v)$
6: **while** $B' \leq B^i$ **do**
7: $T_A^{under} = \frac{1}{B'} \times \sum_{j=1}^{|\mathcal{V}^i|} (T(v_j, Alloc(v_j)) \times cost_{under}(v_j))$
8: **while** $T_{CP} > T_A^{under}$ **do**
9: **for all** $v_i \in$ Critical Path **do**
10: Determine $Alloc'(v_i)$ such that $p'(v_i) = p(v_i) + 1$
11: $Gain(v_i) \leftarrow \frac{T(v_i, Alloc(v_i))}{p(v_i)} - \frac{T(v_i, Alloc'(v_i))}{p'(v_i)}$
12: **end for**
13: Select v such that $Gain(v)$ is maximal
14: $Alloc(v) \leftarrow Alloc'(v)$
15: Update T_A^{under} and T_{CP}
16: **end while**
17: **for all** $v \in \mathcal{V}^i$ **do**
18: Store $Allocs^i(k, v) \leftarrow Alloc(v)$
19: **end for**
20: $B' \leftarrow \max_{v \in \mathcal{V}^i} cost_{under}(v)$
21: $k \leftarrow k + 1$
22: **end while**

This second allocation procedure is detailed in Algorithm 5. The first difference is on lines 5 and 20 where we determine and update the value of B' to be the maximum cost of running any one task. The main difference with our first allocation procedure lies in the outer while loop (lines 6-22). This loop is used to set the value of T_A^{under} that will be used in the inner loop (lines 8-16). This inner loop actually corresponds to an interval of iterations of our first allocation procedure. Each time $T_{CP} \leq T_A^{under}$, the current allocation is stored for each task (lines 17-19), and the current allowed budget is updated (line 20). At the end of this procedure, several candidate allocations are associated with each task in the PTG.

Figure 5.9 shows an evolution of the values of T_A^{under} and T_{CP} across the

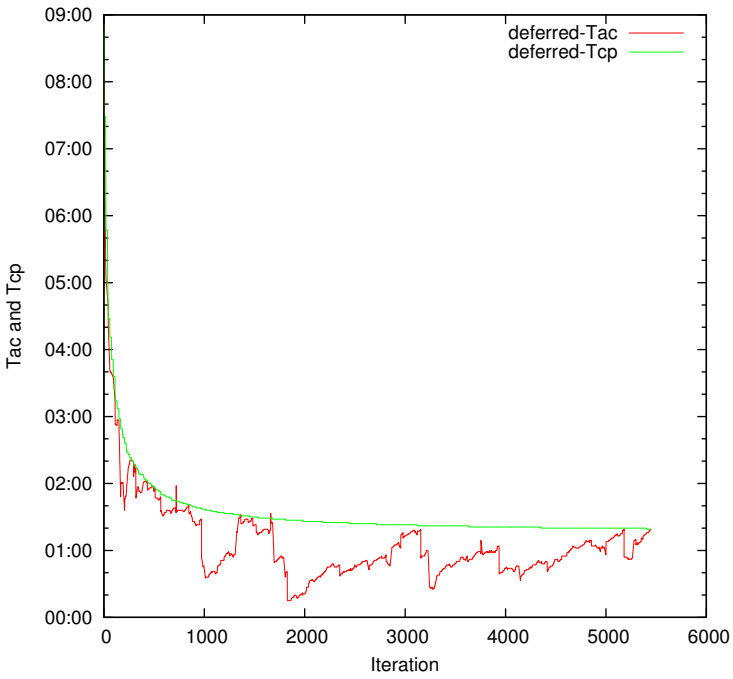


Figure 5.9: The evolution of T_A^{under} and T_{CP}

.

allocation process, for a budget limit of 10 units. In contrast to Figure 5.8, here we have multiple points of convergence for the two values, each of these points represents a valid allocation with a good trade-off between the two. Since in this algorithm we underestimate the cost, there will be a lot more iterations than in the previous. The ridges in the values of T_A^{under} are caused by the difference in price per CPU of the virtual machines from the catalog. As a virtual machine has more CPUs, it's price per hour decreases and so does the value of T_A^{under}.

It is worth noting that the value of T_{CP} becomes more and more flat since the tasks' parallelism starts to become saturated. Here too we have used the T_{CP} *cut-off* strategy in practice.

In a second step, we have to get an estimation of the makespan and total cost that can be achieved with each of these allocations. To obtain these performance indicators, we rely on a classical list scheduling function as shown by Algorithm 6. Tasks are considered by decreasing bottom-level values, *i.e.,* their distance in terms of execution time to the end of the application. For each task, we convert an allocation, *i.e.,* a resource request, into a mapping. This amounts to finding out which set of resource instances the task will be executed on. Two objectives have to be met. First we have to minimize the

finish time of the scheduled task. Second, we have to favor reuse of spare time to reduce the schedule's cost.

To achieve both objectives, we proceed in two steps. First, we estimate the finish time a task will experience by launching only new instances to satisfy its resource request. This set of newly started instances is built so that its cost is minimum, *i.e.,* favor big and cheap instances from the catalog. However, we don't make any assumption about spare time reuse for this mapping. Hence, its cost is computed by rounding up the execution time of the task. This provides us a baseline both in terms of makespan and cost for the current task. Second, we consider all the already started instances, *i.e.,* launched by already scheduled tasks, to see if some spare time can be reused and thus save money. We sort these instances by decreasing amount of spare time (from the current time) and then by decreasing size. Then we select instances from this list in a greedy way until the allocation request is fulfilled, and estimate the finish time of the task on this allocation, as well as the cost of it. This cost is computed as the product of the rounded up execution time of the task by the cost of each instance used minus the cost of the reused spare time.

Now, we have two possible mappings for the current task with different finish times and costs. Our algorithm selects the candidate that leads to the earliest finish time for the task. If the two mappings lead to the same finish time, we select the cheapest option. This is summarized in Algorithm 6.

At the end of a call to Algorithm 6, we have an estimation of the makespan and total cost of the schedule of $\mathcal{G}^i$ using a given allocation. This algorithm is called for each $Allocs^i(k, *)$ as determined by Algorithm 5.

Algorithm 7 details the three stages of our second allocation procedure: (i) Determine a set of candidate allocations for each task (lines 1-3 and Algorithm 5); (ii) Compute the respective makespans and costs achieved by mapping each allocation on a dedicated IaaS cloud (line 7 and Algorithm 6); and (iii) Select the allocation that leads to the best makespan while respecting the budget constraint based on the couples returned by Algorithm 6

Scheduling and workflow execution

It is worth noting that all the previous steps are all static and are performed before runtime. Currently we do not address the problem of workflow execution, as it is not possible to take into consideration the possible state of the Cloud platform and therefore, the resulting schedule would be based on false information. However, by using the allocations selected by our approach we can guarantee that the initial workflow will be run on the Cloud platform given the initial budget, with a certain confidence.

When constructing a schedule by starting from the chosen allocations one should take into consideration the following points: a) as a result of non-determinism, two or more sub-workflows can be ready for scheduling at the same time, yet it is not trivial to find the best order in which they should

Algorithm 6 List-schedule($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), Allocs(*) = Allocs^i(j, *)$)

1: running_instances $\leftarrow \emptyset$
2: **for all** $v \in \mathcal{V}^i$ in decreasing order of bottom-level values **do**
3: $new \leftarrow$ cheapest set of new instances that fulfill $Allocs(v)$
4: $cost(new) = \lceil T(v, Allocs(v)) \rceil \times \sum_{vm_j \in new} cost_j$
5: $finish(new) \leftarrow$ finish time of v on new
6: Sort all $vm_j \in$ running_instances by decreasing spare time and size
7: $reuse \leftarrow$ first set of instances from running_instances that fulfill $Allocsv$)
8: $cost(reuse) = (\lceil T(v, Allocsv)) \rceil -$ reused spare time) $\times \sum_{vm_j \in reuse} cost_j$
9: $finish(reuse) \leftarrow$ finish time of v on $reuse$
10: **if** $finish(reuse) < finish(new)$ **then**
11: $map(v) \leftarrow reuse$
12: **else if** $cost(new) < cost(reuse)$ **then**
13: $map(v) \leftarrow new$
14: **else**
15: $map(v) \leftarrow reuse$
16: **end if**
17: running_instances $\leftarrow$ running_instances $\cup map(v)$
18: **end for**
19: $cost \leftarrow \sum_{vm_j \in VMs} \lceil T_{end_j} - T_{start_j} \rceil \times cost_j$
20: $makespan \leftarrow \max(T_{end_j}) - \min(T_{start_k}), \forall vm_j, vm_k \in$ running_instances
21: **return** $(makespan, cost)$

be scheduled; b) if scheduling is performed offline, there is no possible way of knowing the state of the platform and therefore it is highly likely that the estimations used while scheduling would be false.

5.5 Experimental evaluation

Experimental methodology

We use simulations with synthetic PTGs to evaluate our claims. The synthetic PTGs were generated based on three application models: Fast Fourier Transform (FFT), Strassen matrix multiplication and random workloads that allow us to explore a wider range of possible applications. For more details related to the synthetic workloads and their generation we would like to refer the reader to [115], section V.

Algorithm 7 Find-allocations($\mathcal{G}^i = (\mathcal{V}^i, \mathcal{E}^i), B^i$)

for all $v_j \in \mathcal{V}^i$ **do**
  $Allocs^i \leftarrow Deferred\text{-}allocate(\mathcal{G}^i, B^i)$
end for
$selected\_allocation \leftarrow \emptyset$
$best\_makespan \leftarrow +\infty$
for all $Allocs^i(k, *) \in Allocs^i$ **do**
  $(makespan, cost) \leftarrow$ List-schedule$(\mathcal{G}^i, Allocs^i(k, *))$
  if $(makespan < best\_makespan) \wedge (cost \leq B^i)$ **then**
    $best\_makespan \leftarrow makespan$
    $selected\_allocation \leftarrow Allocs^i(k, *)$
  end if
end for

Platform description

Throughout our experiments we have used Amazon EC2 as our model IaaS platform. This is visible in the virtual resource catalog that we have used, inspired by the the available virtual resource instance types of Amazon EC2 [120] and described in Table 5.1.

Name	#VCPUs	Network performance	Cost / hour
m1.small	1	*moderate*	0.09
m1.med	2	*moderate*	0.18
m1.large	4	*high*	0.36
m1.xlarge	8	*high*	0.72
m2.xlarge	6.5	*moderate*	0.506
m2.2xlarge	13	*high*	1.012
m2.4xlarge	26	*high*	2.024
c1.med	5	*moderate*	0.186
c1.xlarge	20	*high*	0.744
cc1.4xlarge	33.5	10 Gigabit Ethernet	0.186
cc2.8xlarge	88	10 Gigabit Ethernet	0.744

Table 5.1: Amazon EC2's virtual resource types

In our catalog we did not consider instances of type *t1.micro* as it receives virtual CPUs in bursts, which makes it difficult to quantify. We also did not consider GPU cluster instances (*cg1.4xlarge*) as their GPU resources are difficult to quantify in virtual CPUs.

Given that the network bandwidth information for the *m1*, *m2* and *c1* type instances is not given, we have considered *high* network performance as being 10 Gigabit Ethernet and *moderate* network performance as being 1 Gigabit Ethernet.

Comparison of running times

We can consider the running time of the two allocation algorithm on a 16-core Intel Xeon CPU running at 2.93GHz. For convenience's sake we have considered the running time of Eager relative to Deferred for the same PTG and budget. A plot of the relative running time across all the simulation scenarios for each type of application can be seen in Figure 5.10. The first quartile has 25% of the total values smaller or equal to it, the second quartile (median) has 50% and the third quartile has 75%. The range between the first and third quartile is the Inter-Quartile Range (IQR). The whiskers of the plot extend from the ends of the box to 1.5 times the IQR. For convenience's sake, outliers are not show.

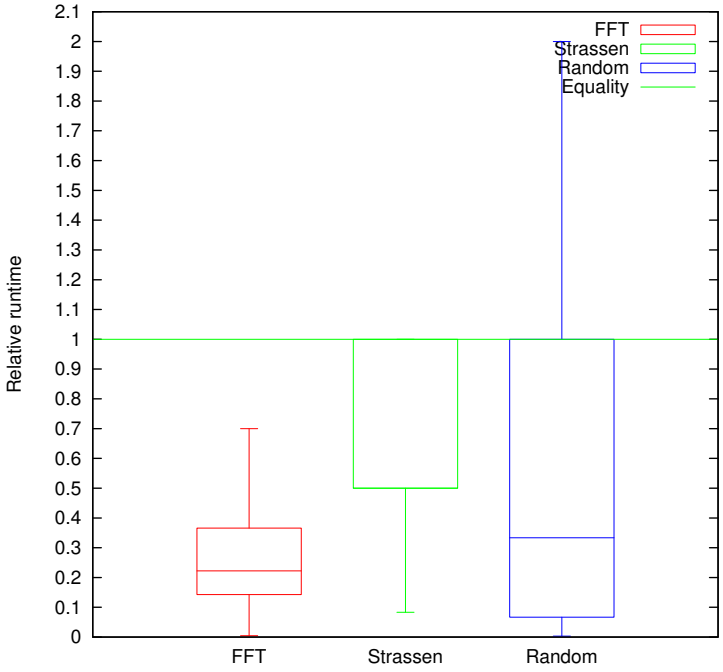


Figure 5.10: Relative runtime of the two allocation algorithms $(\frac{Time\ to\ compute\ Eager}{Time\ to\ compute\ Deferred})$

Deferred's outside iteration over the budget limit has a visible influence, especially for higher values of the maximum budget. Deferred's running time is slower than Eager's by at most an order of magnitude. It is worth noticing that the behavior is as expected, Eager is significantly faster than Deferred for almost all the allocations performed. In the situation of small PTGs, both algorithms run considerably fast and in these situations, the resolution of the internal clock can introduce disturbances, as seen in the case of random PTGs.

Simulation results

We have varied the budget limit for all the input PTGs from 1 unit to 50 units. By considering the cost per hour of the cheapest VM type (0.0084 units per CPU per hour) from the catalog in Table 5.1 gives a testing interval from a minimum of 11 CPU hours to a maximum of 5914 CPU hours. This has the double role of permitting bigger PTG to manifest their influence over time to produce a more general trend and stressing the algorithms in order to find out their best operating parameters.

Figures 5.11, 5.12, 5.13 and 5.14 shows plots of aggregated results of makespan and cost after task mapping, for all three application types. We have used the same semantics for quartiles and whiskers as previously explained.

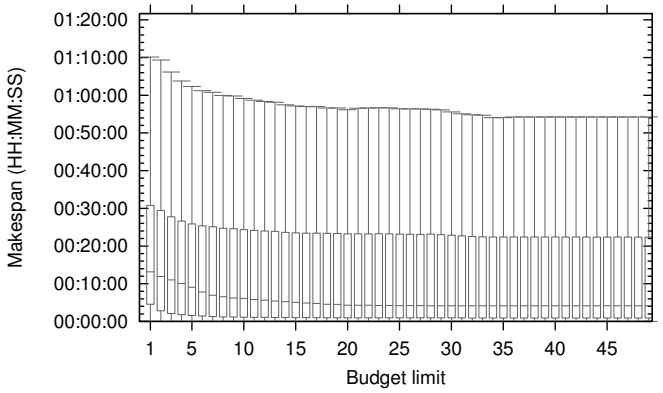


Figure 5.11: Makespan using Eager allocation using all workflow applications

The first observation worth noting is that up to a certain budget value Eager passes the budget limit. This means that our initial assumption of 50% VM spare time reuse is an optimistic one. After a certain budget limit, Eager reaches a point of saturation due to the T_{CP} *cut-off* strategy. This means that after a certain budget limit, the same allocation will be produced by Eager and, consequently, the same task mapping after scheduling.

While the T_{CP} *cut-off* strategy also applies to Deferred, it does not try to estimate the costs, it always underestimates them while performing allocations. As a result, the actual costs of the allocations given by Deferred will be a lot higher than the budget limit and the actual saturation level will also be higher. As expected, Deferred in combination with Algorithm 7 will always select an allocation that, after task mapping, is within the budget limit. In combination with a high saturation level this, yields the behavior that we see in Figure 5.12. The only moment when Deferred produced allocations that are not in the budget limit is when the budget limit is too low to accommodate

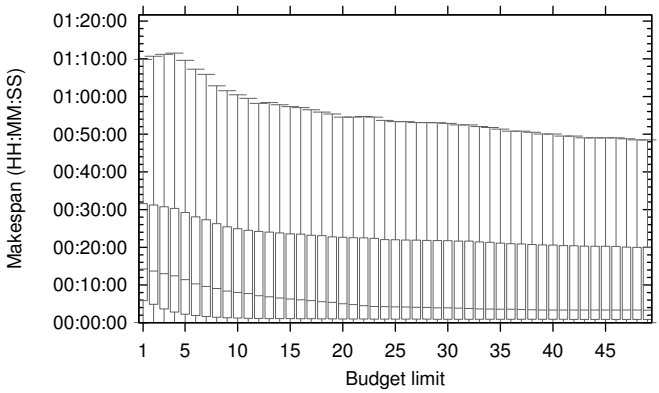


Figure 5.12: Makespan using Deferred allocation using all workflow applications

all the tasks in the workflow.

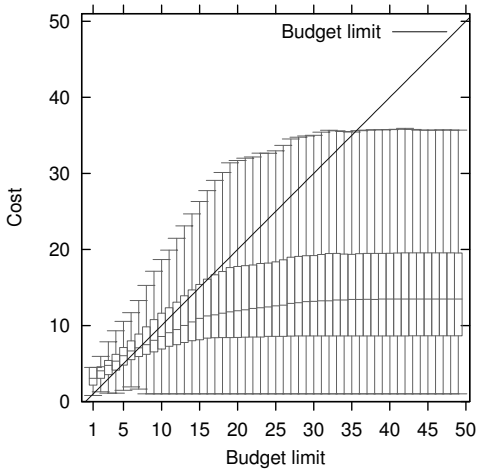


Figure 5.13: Cost using Eager allocation using all workflow applications

To ease the comparison between the two approaches, we can consider the plots in Figures 5.15 and 5.16. It can be seen that, in the beginning, the makespans produced by Eager allocations are shorter than those produced by Deferred allocations and from a cost point of view, Eager produces more costly allocations than Deferred. As the budget increases, the balance shifts slightly in favor of Eager for cost and Deferred for makespan, yet it is not as unbalanced as in the beginning.

For small values of the budget *i.e.,* before task parallelism starts to be-

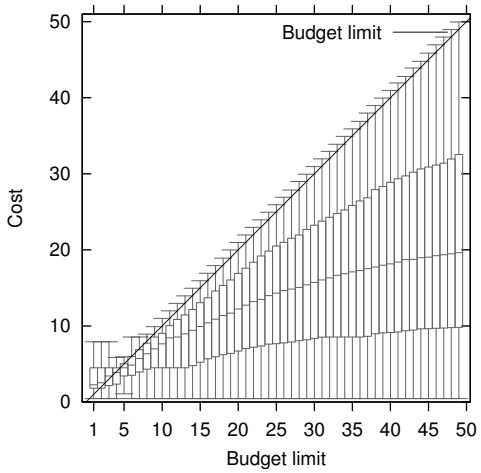


Figure 5.14: Cost using Deferred allocation using all workflow applications

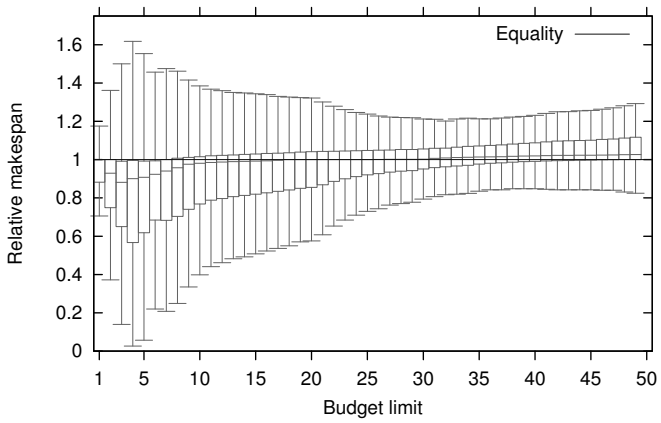


Figure 5.15: Relative makespan ($\frac{Eager}{Deferred}$) for all workflow applications

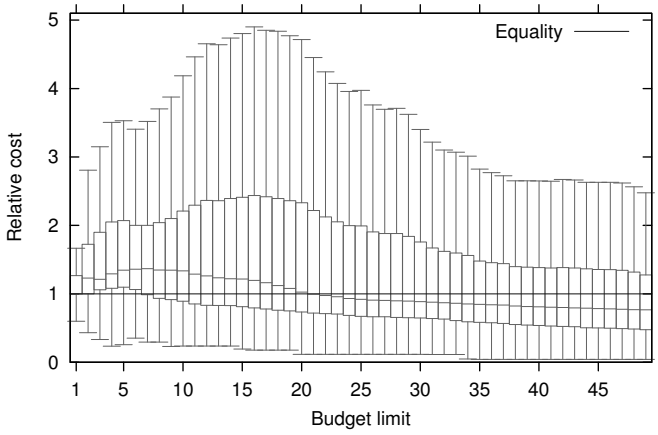


Figure 5.16: Relative cost ($\frac{Eager}{Deferred}$) for all workflow applications

come saturated, Eager outperforms Deferred in terms of resulting makespan by a median of as much as 12%, but Deferred never passes the budget limit and outperforms Eager in terms of budget by a median of as much as 26%. The situation changes once task parallelism begins to appear and the two algorithms yield the same makespan with a median difference of 2%, yet Eager outperforms Deferred in terms of cost by as much as 23%. It it therefore intuitive that for small applications and small budget values one should use Deferred, but when the size of the applications increases significantly or the budget limit approaches task parallelism saturation, using Eager would be the best strategy.

5.6 Scheduling a concrete workflow application

This section will present experiments with real-life workflow applications. The workflow pattern is very common amongst scientific applications. We have focused on a workflow application called RAMSES to test our approach. RAMSES is an *Adaptive Mesh Refinement* (AMR) application that simulates the interaction of dark matter particles in a 3D region of space. The dark matter particles represent the backbone of galaxy formations and allows testing of cosmological models.

The RAMSES workflow (Figure 5.17) is composed of tasks that:

- generate Initial Condition (IC)
- preprocess the current state of the simulation
- simulate dark matter interactions. This task is an MPI parallel application.
- do refinement by looping to the preprocessing step

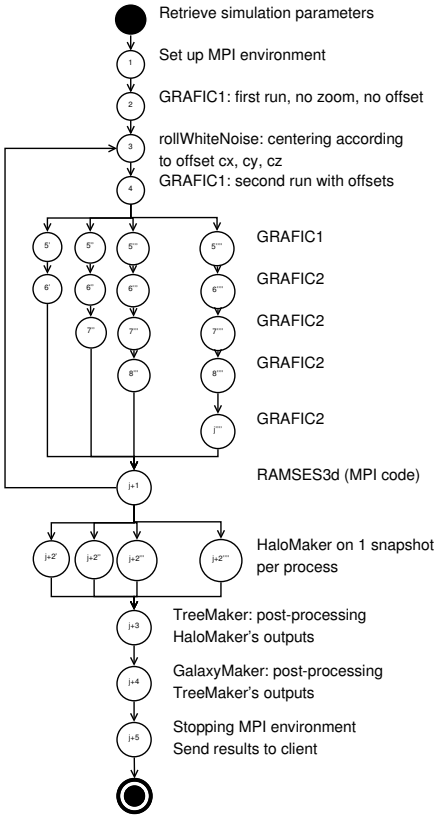


Figure 5.17: The RAMSES workflow

– extract dark matter halos
– build galaxy trees on the extracted halos
– build galaxies on the generated galaxy trees

RAMSES can perform simulations at two different levels of granularity: normal simulations and zoom simulations that concentrate on an "interesting" region of the simulated space.

To facilitate testing we have developed a platform prototype that is able to schedule a workflow in an IaaS platform. We have used DIET MADag as the workflow engine and a FutureGrid installation of Nimbus as the IaaS provider.

The architecture of the prototype platform can be found in Figure 5.18. Its main components are:

Nimbus - the open source IaaS provider. Nimbus implements the Amazon EC2 interface. We have used it as a low level resource provider through a FutureGrid installation.

Phantom - the open source auto-scaling provider that implements a part of the functionality of the Amazon AWS auto-scaling service. We have used it as a high-level resource and availability provider.

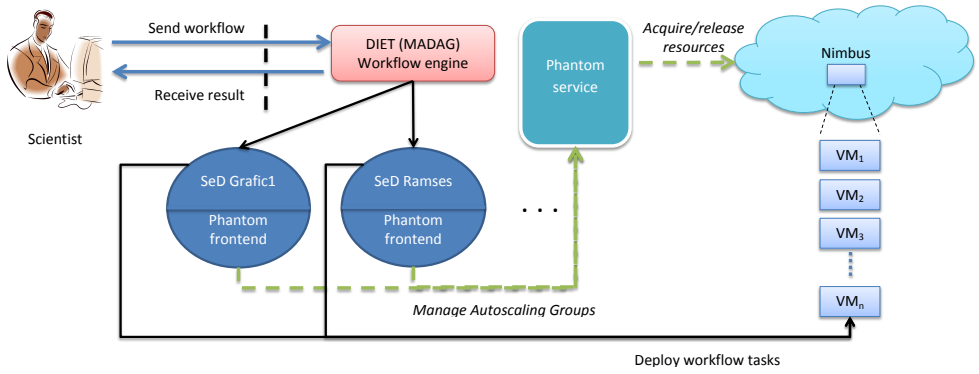


Figure 5.18: System architecture

MADag - workflow engine that is part of the DIET (Distributed Interactive Engineering Toolkit) software. It supports DAG, PTG and functional workflows. We have used one DIET service per workflow task and in our implementation each service launches its afferent task.

Client that describes his workflow in an xml format following the Gwendia [121] language. The client is also responsible for implementing the services and their afferent tasks. The client has the choice of which IaaS provider to use and then calls the workflow engine that will automatically run his workflow.

As a first stage, we have experimentally compared an estimation of costs and running times for static allocations versus dynamic allocations of the RAMSES workflow while varying the size of the allocation in both cases. As experimentation testbed we have used the FutureGrid [122] distributed testbed.

Figure 5.19 shows the real running times for a simulation versus the size of the allocation. The dynamic runtime is composed of two main parts:

- VM reallocation time that represents the delays needed for the virtual platform to allocate new VMs and release unused ones.
- actual work time that represents the running time of the simulation

The component "Runtime (dynamic)" represents the total time, *i.e.* sum of the two above components, for a dynamic platform. As expected, the dynamic allocation yields longer total runtime since it includes a reallocation overhead. An observation worth noting is that the VM reallocation time keeps increasing with the size of the virtual platform, but with each added VM it increases less. In other words it stabilizes and for large platforms it has a constant behavior.

Figure 5.20 contains plots of cost estimations for the two cases of static and dynamic allocation when doing the same simulations. We have considered that the cost is the same for both types of allocations and have estimated it in currency-independent units. What is worth noticing is that the cost for a dynamic allocation is always lower than the cost of a static allocation and for

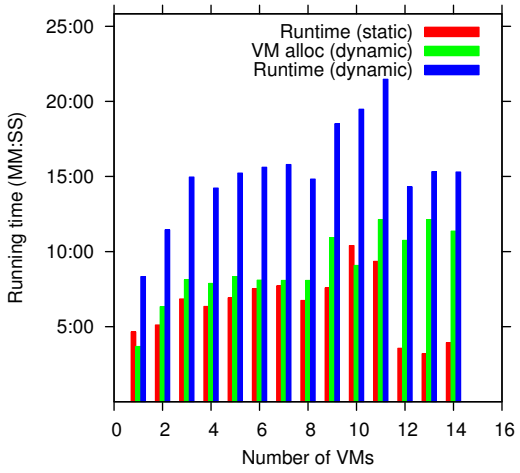


Figure 5.19: Running times for a $2^6 \times 2^6 \times 2^6$ box simulation

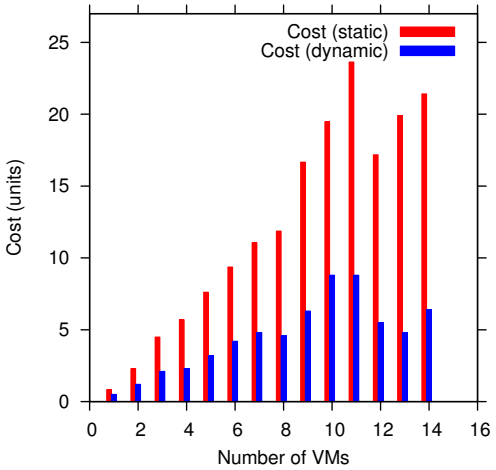


Figure 5.20: Estimated costs for a $2^6 \times 2^6 \times 2^6$ box simulation

large platforms, in our experiments, it can be smaller by a factor of 4x. The difference in runtime when comparing the two platforms is considerably below the price difference factor. The most important conclusion of this experiment is that while the price difference is a linear function of the platform size *i.e.* it increases linearly with the platform size, the runtime difference between the two allocations is sublinear and has a constant behavior for large platforms, essentially becoming a constant overhead.

The second stage consists of testing the two budget constrained scheduling algorithms through an implementation on top of the currently described prototype platform. This work is still ongoing.

5.7 Conclusions and Future Work

The elastic allocations that Cloud platforms offer has opened the way for more flexible data models. Notably, parallel task graph applications with a more complex structure than classic DAG workflows are a good match for the elastic allocation model. There has been lots of work around the topic of parallel task graph scheduling on grid or Cloud platforms, yet none of the previous approaches focus on both elastic allocations and non-DAG workflows.

In the current chapter we present our research on the topic of scheduling with budget constraints for non-DAG workflow models that target Cloud platforms. Our approach is to transform the original problem into a set of smaller sub-problems that have been studied before and propose a solution for them. Concretely, we split the input non-DAG workflow into DAG sub-workflows. Next we present two allocation algorithms, Eager and Deferred, built on the specifics of a typical IaaS Cloud platform and provide an algorithm for selecting the most interesting of these allocations such that the budget limit is not reached. Eager is designed to be a fast allocation algorithm and uses a heuristic approach for estimating the real cost of the allocation it produces. Deferred, on the other hand, is slower in running time, but it produces a set of allocations, each with a good trade-off between the time on the critical path and the total work area (in cost). It does not try to estimate the real cost of the allocations, but underestimates it instead and delays the decision of which allocation to choose until scheduling time. The two algorithms differ in terms of running time by as much as an order of magnitude in favor of Eager. Under tight budget constraints, Eager leads to shorter, yet more expensive schedules and usually passes the budget limit. In contrast, Deferred always results in schedules that are in the budget limit and longer as makespan. The conclusion is that for small applications or small budget limit sizes, Deferred yields the best results and for large applications or large budget limit sizes Eager outperforms Deferred.

As long term goal we plan on integrating the current work into an existing Open Source IaaS Cloud platform. A good improvement will be to determine per application which is the tipping point up to which Deferred should be used and after which Eager would be the best fit.

Acknowledgments

Experiments presented in this chapter were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

Chapter 6

Overall conclusions and perspectives

In recent years we have witnessed an increase in services labeled as "Cloud". Infrastructure as a service Cloud platforms offer virtual machines from a predefined catalog of resource types. Cloud computing brings the ability to dynamically scale a virtual platform automatically, without any prior contract and with a low overhead. This leads to cost saving by only paying for the resources that are used and only using the resources that are needed. From a scientific point of view, this raises questions related to scheduling strategies adapted for this type of dynamic platforms.

The current Thesis focuses on determining efficient mechanisms for resource management in Cloud platforms. This includes both resource provider point of view and Cloud user point of view.

In order to clarify the context of the current work, in Chapter 2 we introduce the reader into the general topic of Cloud computing and we present a detailed study of the state-of-the-art in the field. We focus on the most relevant of Cloud platform features: automatic scaling, load balancing and platform monitoring. We examine the presence and capabilities of these features in commercial and open-source platforms. We also explore how these features can be implemented by examining research efforts being done around these topics.

The "killer feature" that Cloud platforms offer is ability to automatically scale a virtual platform without any prior contract, which leads to saving costs on unused resources as they can be automatically released from the virtual platform. Automatic scaling or "autoscaling" is achieved by using a scaling strategy. Determining a good strategy is not a trivial task and there are many

approaches to this problem. In Chapter 3 we present the problem of automatic scaling in more detail and we present an algorithm that helps in determining a good scaling decision. The algorithm that we propose determines repetitive usage patterns in the Cloud client's platform and uses these to extrapolate what the future platform usage will be. The patterns do not have to be periodic, just repetitive, which is documented behavior in web traffic and is inherent to certain applications and usage scenarios. To test our approach we have used traces from one Cloud client application and several grid workloads. Our results show that the algorithm we propose is capable of identifying nonperiodic repetitive behavior with a high accuracy and that the accuracy of the prediction can be improved if we increase the size of the historic database or if we present the algorithm with a historic database closer to the domain of the application that we are trying to predict.

A common use-case of Cloud platforms is to extend another, exiting platform with virtual resources in a transparent way. In this type of scenario the platform manager tries to share resources fairly amongst its users. In Chapter 4 we study the problem of fair virtual resources sharing by extending the DIET grid middleware. We propose a model of resource sharing based on the free flow of value given by markets. We propose these mechanisms as an extension to DIET where the DIET users have a limited amount of virtual currency that they use for running their services on virtual resources. Running a service is done in three steps, following the *tender / contract-net* model: first the DIET user sends a request to the DIET platform, then each service responds with an estimation of the user's utility when running the service and finally the user chooses one of the offers to run his service. The user's utility is a user-defined function that reflects a metric that is important for him. One example is a function that minimizes cost and running time of the service. This model is useful for fair resource sharing.

Many scientific applications can be described through workflow structures. The reasons for this may vary from building applications on top of legacy code to modeling phenomena that have an inherent workflow structure. In Chapter 5 we analyze and describe in details the workflow class of applications. We present an approach for virtual resource allocation for this class of applications. Given that Cloud platforms are advertised as having an infinite number of resources, the classic approach of resource allocation and scheduling needs to be adapted. In the current situation we do not have a fixed number of resources and therefore when performing allocation, both makespan and allocation cost must be taken into consideration. This leads to a bi-criteria optimization problem. To address this problem, we have developed two algorithms that perform resource allocation while keeping a budget limit. Both algorithms are based on the bi-CPA algorithm.

The first algorithms, called "eager" determines only one allocation that represents a good trade-off between the length of the critical path of the graph and the average cost-area per task. Cost estimation is done at allocation time,

which leads to a fast algorithm, but an inaccurate cost estimation, because the real costs of virtual resource usage are only known after scheduling is performed.

The second algorithm is called "deferred". It does not try to estimate the final cost of the used resources, deferring the decision until scheduling time. As a result it returns a set of possible allocations, each representing a good trade-off between the length of the critical path and the average cost-area. This second algorithms is slower than the first, but cost estimations are more accurate since they are delayed until scheduling is performed. The decision of which allocation to choose is only taken when scheduling is performed, thus guaranteeing that the budget limit is not passed.

In order to validate our claims we have simulated the two algorithms with synthetic traces that model three application classes. Our results confirmed the difference in running time of the two algorithms as well as the expected behavior. Deferred gives allocations that lead to more accurate schedules and always keeps the budget limit, while Eager is faster in running time, but the resulting schedules can sometimes pass the budget limit. After a certain budget limit the two algorithms converge and yield close allocations. The conclusion here is that Deferred can be used for small budgets and Eager can be used when budget limit increases.

Perspectives

The work in this Thesis can be extended in several ways. Related to the Cloud client auto-scaling algorithm that we proposed in Chapter 3 there are a few open question. First, determining the appropriate length of the search pattern is a problem in itself. The pattern length is the length of the sliding window used for prediction. There is no single length that gives accurate predictions for all application types, in fact the length of the pattern might not be constant even for the same application. There are many factors that influence this length, most notable are the granularity of the historic database samples and the volatility of the application's use. This makes the problem nontrivial and interesting.

Another possible direction of improvement is to isolate per-application historic usage databases and determine how these databases can be composed to describe a new application type in a relevant manner.

Continuing in the field of resource allocation, the market based resource allocation mechanism in Chapter 4 can be extended with co-allocation capabilities. There are numerous scenarios where a user needs more than one resource at the same time in order to complete a task that built on top of all the resources. This requires coordination in obtaining the multiple resources at the same time and is know as *co-allocation*. Achieving this can be done by adding a layer in between the SeD offers and the user. This layer would

coordinate co-allocations and possible delays between the start time of the different resources.

For the more specialized VM allocation algorithms presented in Chapter 5, a possible direction of improvement is to determine formally when an application should pass from using the "deferred" algorithm to using the "eager" algorithm.

Another direction of further research is to automatically determine workflow application profiles. This can be done by gathering statistics on application executions and automatically building a probability distribution model for the nondeterministic transition inside the workflow. Once done this will improve the accuracy of the allocations and in general the performance of the two algorithms.

Appendix

Appendix A

Publications

Note: author names are in alphabetic order or in no particular order.

International reviewed journals

[J1] Luis Rodero-Merino, Luis M. Vaquero, Eddy Caron, Frédéric Desprez, Adrian Muresan. **Building safe PaaS clouds: A survey on security in multitenant software platforms.** *Elsevier Journal of Computers & Security.* Volume 31, Number 1. 2012, Pages 96-108

[J2] Luis Rodero-Merino, Eddy Caron, Adrian Muresan, and Frédéric Desprez. **Using clouds to scale grid resources: An economic model.** *Future Generation Comp. Syst., 646.* 2012.

[J3] Eddy Caron, Frédéric Desprez and Adrian Muresan. **Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients.** *Journal of Grid Computing.* Vol. 9, Issue 1, pages 49-64, 2011

Book chapters

[B4] Frédéric Desprez, Eddy Caron, Luis Rodero-Merino and Adrian Muresan. **Auto-scaling, load balancing and monitoring in commercial and open-source clouds**. *CRC Press, "Cloud computing: methodology, system and applications"*, DOI doi:10.1201/b11149-17, Chapter number 14, Pages 301-323, 2011

International reviewed conferences

[C5] Eddy Caron, Frédéric Desprez, Adrian Muresan and Frédéric Suter. **Budget Constrained Resource Allocation for Non-Deterministic Workflows on a IaaS Cloud.** *12th International Conference on Algorithms and Architectures for Parallel Processing. (ICA3PP-12)*, Fukuoka, Japan, September 04 - 07, 2012

[C6] Adrian Muresan, Kate Keahey. **Outsourcing computations for galaxy simulations.** *In eXtreme Science and Engineering Discovery Environment 2012 - XSEDE12*, Chicago, Illinois, USA, June 15 - 19 2012. Poster session.

[C7] Eddy Caron, Frédéric Desprez and Adrian Muresan. **Forecasting for Grid and Cloud Computing On-Demand Resources Based on Pattern Matching.** *2nd International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2010).* Indianapolis, USA, Nov 30 - Dec 3, 2010

[C8] Eddy Caron, Frédéric Desprez, David Loureiro, Adrian Muresan. **Cloud Computing Resource Management through a Grid Middleware: A Case Study with DIET and Eucalyptus**. *IEEE CLOUD-II*, Bangalore, India, June 2009

National reviewed conferences

[N9] Adrian Muresan. **Prédiction d'allocation de ressources pour les grilles et les Clouds basée sur la recherche de motifs.** *Rencontres francophones du Parallélisme (RenPar'20).* Saint-Malo, France, May, 2011.

Press articles

[P10] Eddy Caron, Frédéric Desprez, Luis Rodero-Merino and Adrian Muresan. **Recent developments in DIET: from Grid to Cloud.** *ERCIM News, Special Theme: "Cloud Computing Platforms, Software and Applications"*, No. 83, Pages 25-26, October 2010.

Research reports

[R11] Eddy Caron, Frédéric Desprez, Adrian Muresan, and Frédéric Suter. **Budget Constrained Resource Allocation for Non-Deterministic Workflows on a IaaS Cloud**. *Research Report RR-7962, INRIA*, May 2012.

[R12] Eddy Caron, Luis Rodero-Merino, Frédéric Desprez, and Adrian Muresan. **Auto-Scaling, Load Balancing and Monitoring in Commer-**

cial and Open-Source Clouds. *Research Report RR-7857, INRIA*, February 2012.

[R13] Luis Rodero-Merino, Eddy Caron, Frédéric Desprez, and Adrian Muresan. **Using Clouds to Scale Grid Resources: An Economic Model**. *Research Report RR-7837, INRIA*, November 2011.

[R14] Luis Rodero-Merino, Luis Vaquero, Eddy Caron, Frédéric Desprez, and Adrian Muresan. **Building Safe PaaS Clouds: a Survey on Security in Multitenant Software Platforms**. *Research Report RR-7838, INRIA*, November 2011.

[R15] Eddy Caron, Frédéric Desprez and Adrian Muresan. **Forecasting for Cloud Computing On-Demand Resources Based on Pattern Matching.** *Research report RR-7217, INRIA*, February 2010.

[R16] Eddy Caron, Frédéric Desprez, David Loureiro, and Adrian Muresan. **Cloud Computing Resource Management through a Grid Middleware: A Case Study with DIET and Eucalyptus**. *Research Report RR-7096, INRIA*, 2009.

Appendix B

Building Safe PaaS Clouds: a Survey on Security in Multitenant Software Platforms

This chapter surveys the risks brought by multitenancy in software platforms, along with the most prominent solutions proposed to address them. A multitenant platform hosts and executes software from several users (tenants). The platform must ensure that no malicious or faulty code from any tenant can interfere with the normal execution of other users' code or with the platform itself. This security requirement is specially relevant in Platform-as-a-Service *(PaaS) clouds. PaaS clouds offer an execution environment based on some software platform. Unless PaaS systems are deemed as safe environments users will be reluctant to trust them to run any relevant application. This requires to take into account how multitenancy is handled by the software platform used as the basis of the PaaS offer. This survey focuses on two technologies that are or will be the platform-of-choice in many PaaS clouds: Java and .NET. We describe the security mechanisms they provide, study their limitations as multitenant platforms and analyze the research works that try to solve those limitations. We include in this analysis some standard container technologies (such as Enterprise Java Beans) that can be used to standardize the hosting environment of PaaS clouds. Also we include a brief discussion of Operating Systems (OSs) traditional security capacities and why OSs are unlikely to be chosen as the basis of PaaS offers. Finally, we describe some research initiatives that reinforce security by monitoring the execution of untrusted code, whose results can be of interest in multitenant systems.*

B.1 Introduction

The term *multitenancy* refers to the ability of a platform to run software from different users *in a safe manner*. To some degree, multitenancy is supported in many software platforms such as OSs or Virtual Platforms (VPs) such as Java and .NET. However, as this survey shows, none of these platforms offer a fully secured hosting environment. This problem is relevant even in controlled environments where only code from trusted users will be run: faulty code can stall its container for example by allocating too many objects (so the system runs out of memory). Security concerns are even more pressing if code from unknown users is hosted.

This work depicts how malicious code can interfere with the container platform that executes it, or with other software also hosted in the same container. Also, it presents the research works that try to solve the security limitations of standard platforms regarding multitenancy. As we will see this problem has not been neglected by the research community, but arguably it has not received as much attention as other security-related problems so far (e.g. Web attacks such as denial of service, cross-site scripting or SQL injections have been deeply studied). This is likely to change due to the growing importance of cloud systems [11] where multitenancy is specially relevant.

Cloud systems allow organizations to outsource the operation of IT infrastructure, both hardware and software. Much attention has been payed to them due to the potential benefits and business opportunities that clouds could bring [123]. However, there are several concerns that could impede the adoption of cloud-based solutions [124]. Some of them are uncertain reliability (low availability and/or performance dropouts), vulnerability to network attacks (e.g. Denial of Service attacks), or potential vendor lock-in (users not being able to migrate their software to other clouds). Those are not addressed here as they are outside the scope of this work. Another relevant factor to be considered by potential cloud users is security: if clouds are perceived as risky environments users will be very reluctant to migrate their systems there [125]. Unfortunately, securing clouds is not a trivial task as they must face several threats. This survey focuses on the risks induced by multitenancy in *Platform-as-a-Service* (PaaS) clouds. A PaaS cloud provides a container platform where users deploy and run their components. A well known example is Google App Engine (GAE) [1], which runs Java servlets. In a PaaS cloud components from different users can be run in the same platform or container system. As we will see, this implies that malicious users have several straightforward ways to interfere with the normal execution of other components or with the container itself. This is emphasized in [126], where the authors specify that providers are responsible for isolating components so that no user software can interfere with other users. This chapter further explores this requirement by surveying

1. `http://code.google.com/appengine`

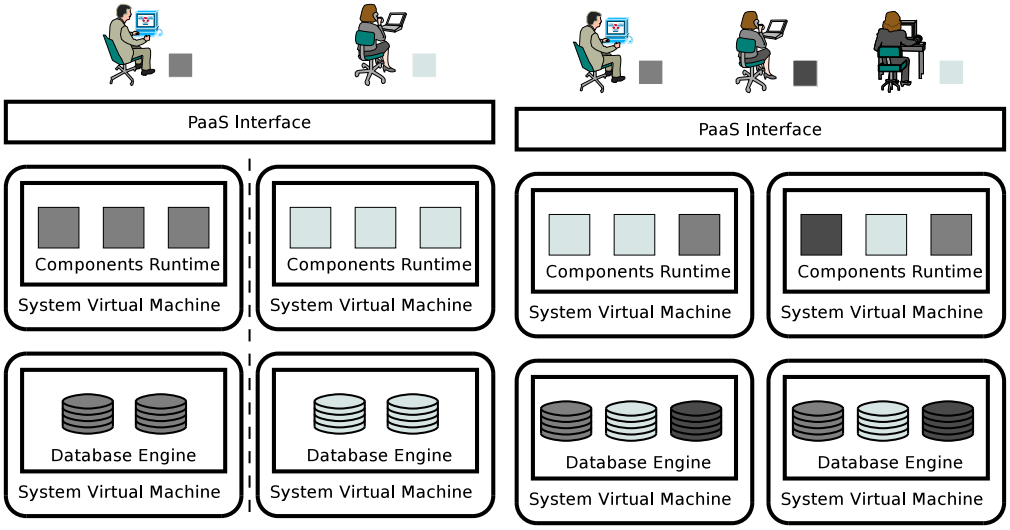
the isolation capabilities of potential PaaS platforms. This analysis is due at three levels representing three possible container systems: Operating System (OS) level, Virtual Platform (VP) level (i.e. Java and .NET) and container level. Most emphasis is put on the VP and container levels as, as we will see, these are more relevant for PaaS clouds.

To avoid confusion, we should clarify that there are some systems also denoted PaaS clouds that build a unique environment per user which is hosted in not shared machines, e.g. provided by an *Infrastructure-as-a-Service* (IaaS) cloud. This is the case for example of Stax.net [2] that offers pre-packaged disk images with the software stack that the user demands and where the deployment and monitoring process is eased thanks to the custom tools provided. Fig. B.1(a) shows an example of such layout, where the PaaS system deploys each user's components in different Virtual Machines. In these systems it is the provider of the VMs (an IaaS provider) who is in charge of implementing proper isolation (which has its own challenges, see [127]). Hence, this chapter does not deal with such PaaS systems as they delegate the implementation of secure isolation to the VM level.

In this chapter we focus instead on PaaS clouds that host and run applications from several different users in the same platform [128] in a safe manner. Tenants share PaaS platform resources (hardware, libraries, supporting services, IT management, etc.), but this is totally transparent to them. This way, the provider can host more users' applications in the same resources. Fig. B.1(b) depicts such a PaaS system, where components from different users are deployed in the same container systems. To achieve safe multitenancy in PaaS platform each application must run isolated from the rest, so a malicious or faulty application cannot impact others. Also, as the code executed by the PaaS system may be untrusted, it is necessary to find mechanisms that can enforce security policies to decrease the risks involved in running such code.

The rest of this chapter is organized as follows. In Section B.2 we explore security mechanisms at OS level, while at the same time we discuss the limitations of the OS as a hosting environment for PaaS applications. Such limitations seem to signal VPs such as Java as more adequate to build PaaS systems. The most well-known container systems are based on Java, so emphasis is put on this platform. Thus, Section B.3 focuses on studying standard Java features and its limitations as a PaaS container platform from the point of view of security, while Section B.4 discusses security on Java container systems. Section B.5 switches focus to the .NET platform, whose security characteristics are also analyzed. With a more general approach, Section B.6 comments *external code monitoring* as a security solution to be applied in PaaS platforms. Finally, Section B.7 presents some conclusions resulting from this survey.

2. `http://stax.net`



(a) Non-shared Resources PaaS (b) Shared Resources PaaS

Figure B.1: PaaS Systems

B.2 Safe Multitenancy through Process Isolation at Operating System Level

In [129] a complete definition of a secure Operating System is given: "*A secure OS provides security mechanisms that ensure that the system's security goals are enforced despite the threats faced by the system.*". An OS deals with resources such as devices, network, data, memory and processors. Each resource type has different security issues related with it, but to implement safe resource sharing in a multitenant OS five main security areas can be considered. 1) *Access control*, an access mechanism should be available to authorize requests from users or processes to perform OS operations as read, write, etc., on OS objects such as files, sockets, etc. The most well-known solution is based on ACL (for Access Control List), where each object has a list of permissions associated; 2) *Integrated firewall* functionality, like IP Filter, IPsec and VPN techniques; 3) *Data encryption* for data in transit or stored in the file system; 4) *Prevention of execution* of memory zones, using the *No execute* (Nx) page flag; 5) *Isolation* is finally the last (but not least) security area OSs must provide. Process isolation has been a basic feature of most OSs for decades. Proper isolation prevents any process to interfere with others or to access protected resources. This is achieved through well known protection mechanisms (memory segmentation and page mapping) that build a separated address space for each process. A process cannot access memory regions out-

side its address space. Although other ways to implement process isolation have been proposed [130], this is by far the most common in modern OSs. [131] shows how to take benefit of virtual machines to secure an OS. Also, [132] propose a Mandatory Access Control (MAC) based system to ensure integrity in OS and VMs.

Other menaces are present, but they are not so related with multitenancy (which is the main focus of this chapter) or are already dealt with by the areas depicted above. Techniques like intrusion prevention, authentication and availability deal with *external attacks*; *data integrity* is preserved by ACLs and data encryption; *hidden information flows*, which occur when some user' software propagates information that should remain confidential, can happen if users share critical data (e.g. the same database), but in PaaS systems users do not share application-level data, which should prevent this kind of risks if data integrity is properly implemented.

However, typical PaaS systems do not host applications that run right on top of the OS. Although this is technically feasible, PaaS providers prefer to offer other abstractions to users. Reasons may vary:

- Platform standardization and portability. If a PaaS player allowed users to deploy applications to run on top of the OS, she/he would have to decide which OS(s) to offer, which version, and which dynamically linked libraries (and versions) should be available for applications. This is far from trivial and could constrain the set of applications that could be run in the cloud.
- Simplified abstractions. Also, given the domain of the applications that run in PaaS clouds, providers may prefer to offer simplified abstractions that ease the development tasks.
- Dominance of interpreted languages and virtualized platforms in Web development. Finally, it is foreseeable that future developments in PaaS clouds will be strongly Web-oriented (as they will be accessed through the Web). In Web development, scripting languages (Ruby, Python and others) and virtualized platforms (Java or .NET) are dominant.

There are also several concerns that could be raised if PaaS platforms are allowed to run applications from several tenants on the same OS, [133] enumerates some of them: administration, installation, fault and attack isolation; along with crash recovery.

Furthermore, as noted in [134], general purpose OSs do not allow for an appropriate control of scheduling policies and resource management. These authors already advocated for the utilization of *containers*, although with an important difference from present container systems: those containers were an abstraction provided at the OS level. Each container encompassed the resources associated to a particular task. Each application could use one or many containers, and through them the OS was able to monitor and manage the resources (CPU, memory, bandwidth) consumed by each task executed by the application.

This same idea of 'container' is present in many systems, however they are implemented at the application level (where any resource management and tenant isolation task must be implemented).

B.3 Security and Multitenancy in the Java Platform

Arguably, the best well-known container systems are based on the Java platform. The Enterprise Java Bean [135] (EJB) and Servlets [136] specifications (part of the J2EE specification [137]), and the OSGi [3] specification [138] are the most relevant Java container technologies and they can be expected to have a prominent role in future PaaS platforms. For example, the GAE PaaS system already provides a runtime engine for Java servlets.

Standard Security Capabilities of Java

This section presents a brief summary of the main security features of the standard Java platform (for more information on this topic see [139, 140]). The Java specification includes the *Java security model* [4], a set of features that intend to make Java a safe environment. They include: sandbox execution so potential risks for the hosting system are limited; bytecode verification so the runtime is not corrupted; and cryptography, PKI, and secure transport APIs for communications protection. Also, Java implements a *class loading* mechanism that can be used to control which classes can be instantiated by each thread. Typically, in cloud platforms untrusted code will be run by special threads with specific class loaders that limit which classes can be accessed.

Furthermore, Java implements strong *access control* capabilities to limit access to resources such as network, files, system properties, or any logical entity that the container must protect. The class loader sets for each class the *protection domain* it belongs to. This domain carries 1) a set of *permissions*; 2) the *code source*, an entity that contains the public certificates used to sign the code (if any). The security *policy*, which is set when the platform starts, is used to determine which permissions can be assigned to each class depending on its code source. Finally, the *security manager* is the entity that enforces security.

Resources are usually "wrapped" by specific classes. When some functionality needs a resource it will call the corresponding class. That class protects the resource by asking to the service manager to check if a calling thread has the corresponding permissions by traversing back the method stack. For each method in the stack, the security manager checks if the permissions carried

3. The term OSGi was originally the acronym of *Open Services Gateway initiative*, but today that name is obsolete.

4. `http://java.sun.com/javase/technologies/security/`

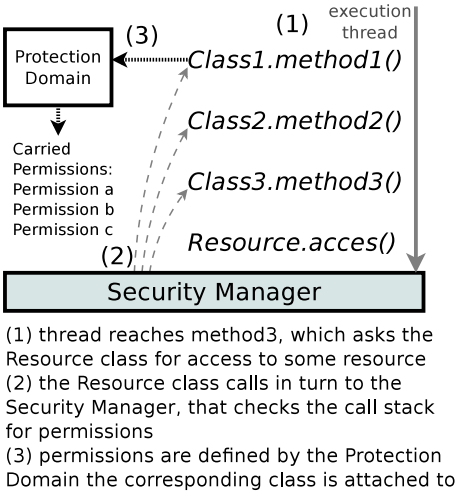


Figure B.2: Checking of Permissions in Execution Stack

by the protection domain the method class belongs to are enough to grant the requested access. If it finds a method in that stack belonging to a class that does not have the required permission, an exception is thrown. This is depicted in Fig. B.2.

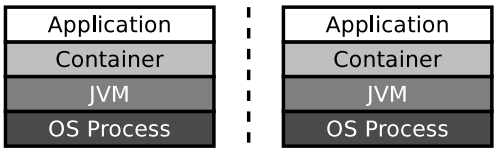
Previous control is code-centric, but can also be user-centric by using the standard *Java Authentication and Authorization Service* (JAAS) APIs. Once a user is authenticated through JAAS, one or more *principals* are associated to her. The security policy used determines which permissions are assigned to each principal when running a certain code. A more complex authorization solution (both role-based and hierarchical) oriented to multitenant clouds is presented in [141].

Security Hazards in Java

Unfortunately, the Java platform also presents certain limitations that hinder the construction of secure multitenant environments. In [142] and [143] the authors analyze the problems and threats to be taken into account when using Java as a multitenant platform. In [142] the authors also study the problems derived from running multitenant software as Java threads. As they explained, even if newer Java versions include protection mechanisms [140] so that no thread could neither modify nor stop other threads, still many issues remain:

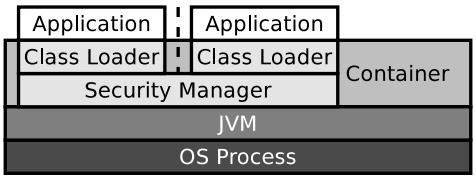
- **Isolation**. A proper isolation mechanism must ensure that one tenant cannot access to components of other tenants. Figure B.3 shows three different isolation solutions that PaaS platforms can use, ranging from isolating applications by running them on their own OS process, going through using already available security devices such as class loaders,

or using last advances on virtual platforms to provide full applications isolation in the same container.



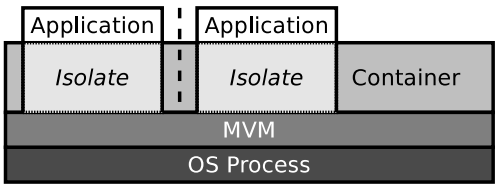
Isolation by OS processes on Virtual Platforms is expensive in terms of resource consumption.

(a) Isolation at OS Level



Isolation with standard Java security features (class loaders, access control). This does not prevent problems such as leaked references, thread termination issues, etc.

(b) Isolation by Standard Java Security



Research targetting advanced isolation abstractions provided for example by MVM.

(c) Isolation at VM Level

Figure B.3: Isolation Options in PaaS Platforms

Fig.B.3(a) shows the most straightforward option, to create a new JVM per user application. This is a safe approach as it uses OS processes to isolate different applications. However it is expensive in terms of resources.

Fig.B.3(b) depicts an approach that enforces security by means of standard Java technologies. Isolation is reinforced by class loaders. Through class loaders, a Java runtime can prevent malicious tenants from loading (and running) not allowed classes or corrupting classes used by other tenants. However, this is not enough to ensure proper isolation among tenants running code in the same JVM. Potential problems vary: visibility of object references from mutable parts of classes (specially static ones), and the possibility for malicious tenants to block other tenants through shared data structures (such as queues) or static synchronized methods [142, 144, 143, 145].

Certain research works have tried to implement the option depicted in Fig.B.3(c)) by providing better isolation mechanisms to Java. In [144] the authors introduce the Multitasking Virtual Machine (MVM), a modified JVM that implements the concept of *isolates*. Each isolate runs a different application (also denoted *task* by the authors) with its own threads in such a way that the application has the illusion of being executed in a non-shared VP. In MVM each task has its own memory heap and so there are no shared objects. Communication between isolates must use other mechanisms such as sockets. Depending on the amount of calls among isolates this can induce a considerable overhead. At the same time, MVM promotes the sharing of as much resources as possible to enhance performance (e.g. core native methods are shared).
Also, static variables are considered by MVM. In a typical JVM, static variables of any class are shared by all threads. In MVM, each isolate keeps its own copy of the static variables, only shared by the threads inside that isolate. Static synchronized methods in each class can be another source of trouble. The monitor associated to those methods is kept by the corresponding instance of `java.lang.Class` (in fact it is the own monitor of the instance). But in the JVM there is only one single instance of `Class` per class, shared by all threads. Hence, the monitor of the `Class` instance is also shared, so if a thread gets the monitor (by synchronizing on the `Class` instance or by calling a static synchronized method of the class) it can block any other thread trying to access it. To avoid this, MVM keeps for the same class different instances of `Class` in each isolate.
Later on, an evolution of the MVM was developed so the same MVM could support applications of different users at OS level [146]. This is implemented by controlling access to private files, allowing the safe execution of native code and adding a mechanism to ensure the correct operation of core native libraries by replicating the global state of shared core classes. Note that this work refers to users at OS level, not to be confused with the tenants of PaaS systems that will try to run their code in the VP. In a PaaS environment, it is safe to assume that the platform will always be started by a single OS level user (admin).
These and other works influenced the Java Specification Request [5] (JSR) 121 *Java Isolation API* [147], which enables Java applications to start other applications in an isolated manner. This specification defines a set of interfaces for the creation and control of isolated containers for components. However, it does not impose any implementation strategy so each isolated component could be implemented by a whole JVM running on an OS process of its own, or all isolations could share the same JVM

5. Java Specification Requests are the standard process to define and propose new additions to the Java platform.

(as in the case of MVM).
On the other hand, the JSR 121 has not been included yet in any standard release of the Java platform, and in fact it seems to be a dormant specification. Also, research project *Barcelona*[6], that hosted the development of the MVM, is no longer active[7].
KaffeOS is another interesting proposal developed by Back and Hsieh [148]. KaffeOS is a new JVM that implements support for isolated *processes* inside the runtime and manages the CPU and memory resources available to each process. These processes are similar to the ones given by typical OSs. They claim that they provide better isolation capacities that the *isolates* given by the MVM.
Geoffray et al. [149, 150] also apply the concept of *isolates* originated by the work on MVMs. However, they transform them so that they are not associated to a running task (i.e. threads can migrate among domains in contrast to isolates) but to class loaders (classes loaded by the same class loader are in the same isolation). With this approach they avoid the overhead caused by inter-task communication in the MVM. As in the case of MVM, each isolate keeps its own copy of static variables and instances of `Class`. In [150] the authors introduce *I-JVM*, a modified JVM that implements their concept of isolates. I-JVM is based on VMKit [151], a software framework to speed up the creation of VPs.
Finally, Sun et al. [145] focus on solving the problems originated by the sharing of the heap memory, such as memory leaks from faulty software that can consume all available memory. The heap is split in logical partitions, so the memory faults caused by a component only affect the partition it resides in. The partition can be repaired without rebooting the whole system.

– **Resource Accounting**. As commented before, the security manager and protection domains are the foundation of the Java environment to implement and assign custom security policies that control access to resources by code (depending, for example, on the origin of that code). Unfortunately, once access is granted to some code, that code can use the resource without limitations. There is no accounting of resource usage by threads in the Java platform, and, so, there is no way to enforce a limited utilization of resources. Therefore, a malevolent tenant can, for example, try to exhaust all available memory just by creating many instances of objects.
The (somewhat old) *Java Virtual Machine Profiling Interface*[8] (JVMPI) and its more recent replacement the *Java Virtual Machine Tooling In-*

6. `http://labs.oracle.com/projects/barcelona/`
7. We tried to get in touch with Sun/Oracle to access to the last version of the MVM. We were notified that, although there is a more recent and stable version based on JDK 7, access to the MVM has been restricted since Oracle acquired Sun.
8. `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`

terface[9] (JVMTI) can be used to support resource accounting as they allow to inspect the state of applications and the JVM. However, these interfaces must be used by software written in native code, breaking Java portability. Also, they introduce a considerable overhead that can make them unusable in many production environments. Finally, these interfaces do not aim at accounting of generic resources.
There have been several approaches trying to solve this for different single resources. For example [152] proposes a system able to account memory usage by using a modified garbage collector that computes the total size of objects reachable by each task as it looks for unreachable objects. They are deemed to be imprecise due to shared references [148].
Other works apply bytecode rewriting (also called program transformations) to inject some kind of accounting capabilities to the Java platform in a portable way. This manner, the platform should be able to prevent threads from using too many resources. The most prominent efforts using this approach are JRes [153] and JRAF-2 [154, 155, 156, 157].
As a result of this concern about the lack of a proper resource control mechanism in Java, Czajkowski and others started to work in a new Resource Management (RM) API [158]. This work and the MVM [144] (discussed above) are strongly related. The RM uses MVM's idea of *isolates* as the basic accounting entity that can demand or dispense resources, and [158] introduces an implementation of the RM API on top of the MVM.
Eventually [158] leaded to the creation of the JSR 284 *Resource Consumption Management API* [159]. This JSR, which has been recently approved, defines a set of interfaces that enable the programming of resource management policies. This API *"will be a framework through which resources can be uniformly exposed to client programs as entities subject to management"*. Also, JSR 284 includes a set of core resources that all compliant implementations will have to expose by default. An implementation is already available, but it is unknown if this API will be included in future releases of any of the flavors (J2ME, J2SE, J2EE) of the standard Java platform.
On the other hand, KaffeOS implements per process resource accounting and bounds setting (CPU and memory). It does not provide accounting of other resources neither from the platform nor handled by the users.
Regarding I-JVM, it implements per-isolate accounting of CPU time, threads created, I/O r/w operations and memory. But as KaffeOS, it does not have a general accounting framework for a generic resource.

– **Safe Thread Termination**. This problem is due to the lack of a safe way to enforce the termination of a Java thread. The `java.lang.Thread.stop()` method was intended for that, but:

9. `http://java.sun.com/javase/6/docs/technotes/guides/jvmti`

- It is *deprecated* because it is deemed unsafe: the terminated thread would release all its monitors, which could leave some objects in an inconsistent state.
- The method triggers a `java.lang.ThreadDeath` exception in the thread to stop it. The thread can just catch that exception and ignore it to keep running.

Hence malicious threads can remain alive forever, consuming resources trying to monopolize resources, block other threads, etc. Another problem could be caused by the platform trying to run a safe shutdown, which implies that all threads running inside the platform must be stopped first. If the platform waits for a malicious thread to terminate then it could be brought to a stall state. Some solutions [160] propose to modify the untrusted software bytecode to inject termination checks at certain execution points. These solutions have a drawback: they incur heavy performance penalties.

MVM does solve this problem. A MVM-aware application can create, execute, pause, resume and stop other applications. Also, KaffeOS allows to stop the *processes* it is based on.

Finally, in I-JVM, when one isolate is terminated all the threads originated by it are stopped by a special `StoppedIsolateException` exception that can only be caught by objects outside the terminated isolate (so the exception cannot be ignored by the isolate being stopped).

But I-JVM, on the other hand, does not totally implement safe thread termination. The problem is that in I-JVM the same thread can traverse different domains regardless its origin (this cannot happen in MVM nor in KaffeOS) as isolations are not based on threads unlike MVM isolates or KaffeOS processes. When one thread is stopped all the monitors locked by it are released, which could leave objects synchronized by those locks in an inconsistent state. In I-JVM this could happen when releasing the locks of objects in isolates other than the one being stopped. This is the same reason because the standard `java.Thread.stop()` method was deprecated in Java. The creators of I-JVM estimate that the benefits of light inter-isolation communication outweigh this problem.

B.4 Security in Java Application Containers

It is to be expected that future PaaS clouds will not run user components right on top of the JVM. It seems more likely they will use container technologies to provide added standard services. In [161], the authors identify the security threats that multitenant containers must address and enumerate the security requirements they must fulfill:

- *Availability*: an application shall not use local or connected resources that prevent other applications from running due to resource starvation.

The container should have mechanisms to enforce different resource sharing policies. Also, the container must be available regardless of the state of the applications running inside.
- *Confidentiality and Integrity*: an application shall not explore or modify the platform of other applications if not authorized. Access to other applications and their data must be controlled.

It is straightforward to see that these requirements would be achieved by properly addressing the issues listed in Section B.3. Container availability can be brought by safe thread termination and resource accounting, while confidentiality and integrity would be implemented by full isolation of components.

The remainder of this section focuses on the security features of J2EE and OSGi technologies, as they are the most prominent relevant Java container solutions today. Also, the works that try to bring stronger security capabilities to each container technology are listed.

J2EE Containers

The EJB specification [135], as part of the contract between the EJB and the container, imposes strong restrictions and limitations to what EJBs can do. EJBs cannot create threads (to avoid interferences with the container's ability to control components' life cycle), manipulate files (files are not transactional resources and could also limit the application distributability), modify class loaders, access non final static fields (such fields would make a bean difficult to distribute), etc.

These restrictions are enforced by the EJB container through the standard Java security model (see Section B.3), and all together build an interesting security mechanism. EJB containers combine these constraints with the application of class loaders to achieve proper EJBs isolation. Unfortunately, these restrictions impose a somewhat limited programming model which may not be appropriate for many development needs. And, more important, they are not enough to fully achieve the requirements listed in Section B.3.

On the other hand, the Servlet specification, which is also part of the J2EE platform (as the EJB specification), does not stress isolation among servlets, nor imposes strict restrictions for servlet programming. In this specification, security is concerned only with authentication and authorization of servlets' clients.

It is possible, of course, to apply the standard security Java mechanisms (such as access control and PKI APIs) to the development of servlets and EJBs based systems. There are texts available that address this topic [162, 163]. But even in this case, proper *Isolation*, *Resource Accounting* and *Safe Thread Termination* (Section B.3) would remain as open issues.

Some research works [164, 165] have tried to use MVM (see Section B.3) to achieve proper isolation among users applications on J2EE environments. In [164] the authors discuss how to apply MVM's isolates in a J2EE server.

They propose using *application domain* isolates, where one application domain encapsulates one or more user J2EE applications, including its required servers. Later on, in [165] the authors used a MVM extended with the Resource Management API (defined in [158], see Section B.3) and combine it with application domain isolates, so they can easily monitor the resources used by each application.

A Servlets-Based PaaS: Google App Engine

Being a prominent PaaS platform, based totally in the Java Virtual Platform, it is worth to discuss how GAE has addressed the security problems of standard Java.

First, they limit the possible actions that users can perform applying the Java security model, i.e. they apply custom class loaders and security policies enforced by the Security Manager. For example, tenants cannot create new threads, instantiate certain classes, modify system properties or read files that do not belong to the user application (a GAE application is basically a set of Java servlets, Javascript code, configuration files and static content like images or HTML pages).

Regarding isolation, GAE solves it in a quite "naive" manner: users do not share servers. Each user application runs on its own JVM instance (as depicted in Fig. B.3(a)).

GAE offers accounting data of certain resources: CPU, network bandwidth and stored data size. Users are billed depending on the amount of resources used. However, it is not explained how GAE performs this accounting (using a custom JVM, using the JVM Tooling Interface, at OS level, etc.).

Finally, GAE uses thread termination to control how long it takes to attend each request. A request in GAE can last up to 30 seconds. When that limit expires, an exception is thrown by the platform to the servlet processing the request. If the exception is not caught, the thread will finish and a `HTTP 500 server error` message will be sent in response to the HTTP request that triggered the thread execution. If the exception is caught the runtime engine will give *"the request handler a little bit more time (less than a second) after raising the exception to prepare a custom response"*. After that, the thread is terminated by force. Google claims that the thread is shutdown "gracefully", other threads in the same server are not affected. In fact, the whole container is stopped. To make sure that other threads are not affected, the load balancer in front of the container stops sending requests to it when a thread is to be stopped. Then, when no more threads are running, the whole container server can be stopped. This implies that programmers should develop servlets taking into account that requests should be attended by stateless processes (there is no concept of session affinity per user) as consecutive requests from the same user can be forwarded to different server instances.

OSGi Containers

The OSGi framework defines a platform where loosely coupled software modules (denoted *bundles*) can expose and use services; OSGi enforces some isolation through its *Module Layer*. This layer defines a modularization model so that packages included in each bundle are shared (exported) or hidden to other bundles as declared by the developer. Again, this isolation is implemented through class loaders.

Extra security is provided by controlling whether bundles can export/import certain packages, access resources, etc. Still, OSGi carries several potential security hazards. In [166] the authors enumerate 25 different security flaws in different OSGi implementations. And, while 17 of them can be fixed programmatically by setting proper security measures, there are still 8 vulnerabilities that need to be addressed at JVM level. All of them are related with the security limitations of Java mentioned in Section B.3: poor isolation (e.g. a bundle can modify shared static variables), need for resource accounting (e.g. a bundle could use all of the memory available) and lack of support for thread termination (e.g. a bundle can ignore signals to stop and catch all `ThreadDeath` exceptions).

Some works try to improve the OSGi framework robustness by providing better isolation: Gama and Donsez [167] patch an OSGi implementation using the Isolation API (JSR 121) on MVM to provide service level isolation.

In [150] the authors modify an OSGi implementation to run with I-JVM. They show how applying I-JVM this new OSGi platform solves the 8 risks described in [166] tied to the JVM.

Other works try to enhance the tolerance to faulty software, for example in [168] the authors use light proxies to route calls between bundles that wrap service objects and handle failures when they occur.

B.5 Security Considerations about the .NET Platform as a PaaS Enabler Technology

No any other VP has been as intensively studied as the Java platform. Also, no other VP has reached the same popularity. But Java is not the only candidate VP that can be used to build a PaaS system. This section will introduce the main security features of the .NET platform, which can be regarded as an alternative to the Java platform.

Standard Security Capabilities of .NET

The .NET platform is a development environment created by Microsoft with several similarities with the Java platform. The *Common Language Runtime* (CLR), which would be the equivalent to the JVM in .NET settings, implements the main security aspects of this platform.

In .NET, software is contained in libraries denoted *assemblies*, which are grouped in *code groups*. Membership of code groups is ruled by the *evidences* that each assembly carries (for example who signs the code). Each code group has an associated set of *permissions*. If some assembly belongs to more than one group, its associated permissions are the union of all the permissions of all groups it belongs to.

The mapping between code groups and permissions is done through *security policies*. Policies are organized in a hierarchy with 4 levels (top-down order): enterprise, machine, user and application domain. Usually, the permission associated to each code group is given by the intersection of the permissions at all levels it belongs to, although more complex settings are possible.

Permissions are used for granting access to resources or to other code. They have a stack walking semantics very similar to the one found in Java. If a method demands a certain permission, then all the methods higher than the current one in the call stack are checked for that permission. This prevents attacks in which some untrusted software tries to use a trusted piece of code to run a protected operation.

We can see that the CLR access control mechanism has similarities with the one used in Java, although it is considered by some [169] as easier to use.

Security Hazards in .NET

- **Isolation**. The CLR implements the concept of *Application Domains* (ADs). Each application is assigned an AD when is run by the CLR (the same CLR instance can run several ADs with several instances of different applications). ADs are isolated, so code running in one AD neither can call, nor can be called from code running in other AD. If several application instances use the same code, the CLR will handle one copy of that code per AD where it is used. For intra-process isolation in .NET, using different application domains is recommended because they can be dynamically loaded and unloaded during the runtime of the application.

  An interesting feature of the CLR is that it keeps a separate copy of the static variables maintained for each domain, thus preventing object references from being leaked across domains as static variables. We can conclude that, by default, the CLR has more complete (and thus safer) isolation capabilities than the standard JVM. However, although the *application domain* concept provides a straightforward way to achieve tenancy isolation, the fact is that CLR still suffers some other limitations of the Java platform.
- **Resource Accounting** Just like in the case of Java, .NET does not implement any generic resource accounting functionality. It does have a profiling mechanism, but it provides information about the state of the CLR through events (load/unload of classes, threads creation, and oth-

ers), it cannot be used by components developers to control the resources they offer.
There has been some works around resource accounting that target Windows applications. Notably [170] have described a framework that allows resource accounting. This framework allows the dynamic assignment of resources to tasks and task management to a fine granularity that includes bounding the running context of tasks (for example in CPU and memory usage) therefore creating a sandboxed context for the task. The framework described here targets *unmanaged code* (code that does not target the .NET framework and is not run by the CLR) but the authors stated it was being extended to allow .NET remote resources to be used. As such the presented framework is a viable solution for resource accounting for the .NET framework.

– **Safe Thread Termination**
CLR's thread termination solution is based on a C#'s method (`System. Threading. Thread. Abort()`) that injects an exception in the aborted thread, as the `java. lang. Thread. stop()` does in Java. The `Abort()` method is not deprecated (as the `stop()` method is), yet it is recommended to avoid it [10]. But even more important is the fact that .NET does not guarantee that the thread on which `Abort()` was called is stopped. In fact it is easy for the thread to continue its execution by handling the exception and calling `System.Threading.Thread.ResetAbort()` or by having unbound computations in its `catch` or `finally` statements. Thus, like Java, .NET does not provide a safe mechanism for thread termination.
This impacts ADs management. Before unloading an application domain all its threads must be stopped, which is implemented by using the `Thread.Abort()` method. Note that, given the fact that thread stopping is not guaranteed neither is the successful unloading of an application domain.

Security in .NET Application Containers

Regarding container architectures, no container system similar to Java's EJBs or OSGi exists in .NET. The closest technologies could be *ASP.net* [11] and *Component Object Model* [12] (COM). ASP.net provides a Web framework, but as in the case of the Servlets specification there is no special reinforcement of isolation among components (although it uses the concept of ADs). Regarding the COM platform, it is not built on top of .NET and is not part of the .NET framework. Also, COM is not a container technology *per se*, it is more oriented to enabling the connection of components. COM+ has been developed

10. `http://msdn.microsoft.com/en-us/library/system.threading.thread.abort.aspx`
11. `http://www.asp.net`
12. `http://www.microsoft.com/com`

as an improvement of COM. Recent versions of COM+ add private / public component isolation mechanisms whereas previous versions only offered role-based authorization. For its use in the .NET framework, a wrapper library has been built under the name of .NET Enterprise Services [13].

The compliance and possible implementation of an OSGi-like platform on the .NET framework has been studied by [171]. To enforce OSGi-like containers in .NET, the authors recommend applying ADs. They can provide the necessary isolation mechanisms, yet the only way to communicate between two non-shared application domains is by using interprocess communication solutions such as .NET remoting. These communication mechanisms come with a considerable time overhead which would make some applications impractical, yet the possibility of an OSGi-like platform implemented on top of the .NET framework exists.

There have been a few projects that aim towards the development of a PaaS cloud based on the .NET framework. One such project is the Aneka Cloud Platform described in [172]. The goal of the Aneka project is to provide a PaaS cloud that enables the deployment of public, private or hybrid clouds. The Aneka platform is based on Aneka containers. They provide the services required for platform management and the runtime necessary for the execution of applications.

Security inside the Aneka platform is handled by providers of authentication and authorization. The providers have the role of abstracting the concrete mechanisms that perform the task. As such, Aneka is able to use the underlying authentication and authorization mechanisms of the environment in which it was deployed if required and also to provide custom ones.

Although the general mechanisms used for application isolation in current cloud environments have been presented, the specifics implemented in Aneka related to this domain have not been detailed in the referenced work. As a result, the reader is unsure if Aneka contains implicit isolation or sandboxing for its deployed applications or if the Aneka user is responsible for developing her/his own isolation mechanisms.

In a previous work [173] Aneka has been described as an enterprise grid platform. In addition to the membership-based security approach described above, [173] also presents the possibility of using a certification-based approach with X.509 certificates for authentication. No further details related to the application isolation mechanisms used are given.

B.6 Monitoring External Code Execution to Enforce Security

The security features of VPs and the related research efforts studied so far try to build a safe environment by addressing the platform characteristics

13. http://msdn.microsoft.com/en-us/library/Aa286569

that can be used by malevolent code (e.g. not proper thread termination mechanisms).

Another complementary approach is to *monitor untrusted code execution* to ensure that security policies are fulfilled by tenants' code. For example a security policy that could be enforced in PaaS systems is to impose tenants code to apply SSL connections when sockets are used. Relevant research works related with this approach are analyzed in this section.

The components that monitor code execution and take actions when some policy is violated are denoted *Reference Monitors*. Schneider in [174] presents 1) a formalism to determine which security policies can be reinforced by what he denotes *Execution Monitoring* (EM); 2) an automata-based mechanism to define such policies. The formalism uses a set of restrictions: EM only uses the information obtained by observing the code execution, it does not modify the code observed. It truncates the code execution when some security policy is violated.

On the other hand, although Schneider's definition of EM does not include any mechanism that modifies the executed code, such solutions are also considered by other authors as EM. Schneider himself states that nothing prevents using such approach with arbitrary security automata [174].

Security monitors that modify the untrusted code are denoted *Inline Reference Monitors* (IRM). Some examples of IRM based solutions are

- SASI [175], it adds code that 1) simulates an automaton that enforces a certain security policy and 2) it is executed before each untrusted code instruction.
- Java-MaC [176], an implementation in Java of the Monitoring and Checking architecture, which ensures that the code runs correctly with regards to a formal specification of requirements.
- Polymer [177], it allows to define monitors in the Polymer language and translates them to Java bytecode, which is then used to rewrite the untrusted code.

The idea of weaving security enforcement code inside untrusted modules is clearly related with *Aspect Oriented Programming* (AOP). AOP [178] intends to provide mechanisms to define "crosscutting concerns", or aspects, that are present in different components of the same system. Security is one of such concerns, as many components (if not all) must take into account security policies and constraints.

Through AOP a PaaS platform could reinforce security rules in a transparent manner [179], like for example log relevant data, implement protection techniques against buffer overflows, etc. The Polymer system is in fact using an approach similar to AOP. Java-MOP [172] also applies AOP to monitor formal specifications in programs. In a recent work [180] the authors present an XML-based language to express security rules as automata whose edge labels (i.e. transitions) become AOP *pointcuts*, that is, places in the code affected by a certain aspect and where the IRMs will be injected. A more straightforward

application of AOP to security is found in [181]. Here the authors apply AOP to add role-based access control to a CORBA access control system. Also, users could apply AOP to point out in which parts of the service some security policies must be checked.

Rather than injecting extra code to untrusted applications, other solutions are oriented to the static analysis of software before execution to ensure that it does not break any security police. For example, *Proof Carrying Code* [182] (PCC) carries static information that can be examined before execution to prove that the code is safe. It is unlikely however that in PaaS systems such extra information will be available.

Feature	JVM	CLR	MVM	I-JVM	KaffeOS
Access control mechanisms	Based on Permissions and Policies	Based on Permissions and Policies	Similar to JVM	Similar to JVM	Similar to JVM
Reference leak	Not fixed	Fixed with ADs	Fixed with Isolations	Fixed with Isolations	Fixed with Processes
Shared static references	Not fixed	Fixed with ADs	Fixed with Isolations	Fixed with Isolations	Fixed with Processes
Block by synchronized static components	Not fixed	Fixed with ADs	Fixed with Isolations	Fixed with Isolations	Fixed with Processes
Thread termination	Not fixed	Not fixed	Fixed with Isolations	Not Fixed	Fixed with Processes
Resource accounting	Profiling through JVMTI. Resource accounting specified by JSR 284	Profiling mechanism	Generic resource management API	CPU, memory, #threads, I/O	CPU and memory

Table B.1: Summary of Security Features of Virtual Platforms

B.7 Discussion and Conclusions

As cloud adoption grows, also there will be an increasing demand for multitenant platforms that allow to run, in a safe manner, untrusted code from different users in the same container system.

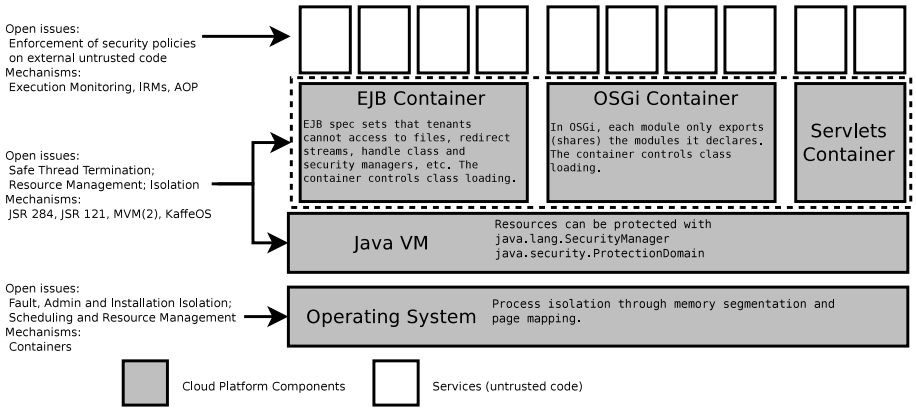


Figure B.4: Summary of PaaS Security Issues and Solutions at Different Layers.

But present standard VPs, that could be used as the basic building blocks of PaaS clouds, still suffer from some important security flaws that must be taken into account when designing a PaaS system. Figure B.4 summarizes the main open security issues at each level of a Java PaaS platform. Also, for each level the figure briefly enumerates both the solutions presented in this survey to address those issues, along with the security mechanisms already implemented.

Table B.1 summarizes the security features discussed in this chapter for different VPs. The *access control mechanism* security feature in that table refers to the standard security mechanisms explained in Sections B.3 and B.5. The *reference leak*, *shared static references*, *block by synchronized static components*, *thread termination* and *resource accounting* features are discussed in Sections B.3 (for the JVM, MVM, I-JVM and KaffeOS VPs) and B.5 (for the CLR VP). From the analysis carried out in those sections it can be concluded that the standard Java platform still has some limitations that hinder the safe execution of untrusted code, a capability that we deem necessary for the construction of PaaS systems. The CLR on the other hand implements more powerful isolation characteristics that solve some of the problems present in Java. However it seems that Java is better positioned as a base platform for building PaaS clouds. First, the CLR still lacks a safe mechanism for thread termination and a generic resource accounting framework (which is addressed in Java by JSR 284). Also, remarkable container technologies are based on the Java platform (J2EE and OSGi) and it is reasonable to expect them to be the basis of several PaaS platforms (as they are already). Furthermore, much research effort has been put on the JVM to address its security limitations (MVM, KaffeOS, I-JVM). Of all these works, MVM seems the more complete solution as it answers all open security issues. I-JVM, on the other

hand, takes a different approach to isolation, so they allow threads to traverse different isolates. This way they solve the high costs of inter-isolate communication present in MVM and KaffeOS. However, due precisely to its design, I-JVM does not solve the thread termination issue. Designers of secure PaaS systems should decide which approach better suits their needs.

Besides the security guarantees achieved by the platform, security in PaaS clouds must address other aspects. First they must try to enforce security policies so users do not build applications that are themselves prone to attack. This can be done through the enforcement of security policies by the code monitoring techniques studied above. A survey of research in this area shows that most proposals are based on AOP in the Java platform, which further positions Java as a good candidate to build secure PaaS clouds.

In any case, future work on VPs and container systems (which will impact on the security of PaaS clouds) should take into account the risks brought by multitenancy outlined in this work. They should use or develop artifacts that bring full isolation among components, blocking access to external references. Also, it must be possible to stop non-trusted threads without affecting the platform, and mechanisms that allow to implements resource sharing policies should be available.

Acknowledgments

This research has been partially funded by the EC under project CumuloNimbo FP7-257993, by the Madrid Research Council (CAM) under project CLOUDS S2009TIC-1692 and by the Spanish Science Foundation under project CloudStorm TIN2010-19077.

Bibliography

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

[2] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, number 3779 in LNCS, pages 2–13. Springer-Verlag, 2005.

[3] Jim Almond and Dave Snelling. Unicore: uniform access to supercomputing as an element of electronic commerce. *Future Gener. Comput. Syst.*, 15(5-6):539–548, October 1999.

[4] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Technical report, GridRPC Working Group, July 2005.

[5] Henri Casanova and Jack Dongarra. Netsolve: A network server for solving computational science problems. In *The International Journal of Supercomputer Applications and High Performance Computing*, pages 212–223, 1995.

[6] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In Louis O. Hertzberger and Peter M. A. Sloot, editors, *HPCN Europe*, volume 1225 of *Lecture Notes in Computer Science*, pages 491–502. Springer, 1997.

[7] E. Caron and F. Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[8] Eddy Caron. *Contribution to the management of large scale platforms: the DIET experience*. HDR (Habilitation à Diriger les Recherches), École Normale Supérieure de Lyon, October6 2010.

[9] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[10] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, Upper Saddle River, NJ, 1999.

[11] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[12] Amazon Web Services LLC. Amazon elastic compute cloud (amazon ec2) `http://aws.amazon.com/ec2/`, September 2012.

[13] Inc. Rackspace. Rackspace open cloud `http://www.rackspace.com/cloud/`, September 2012.

[14] GoGrid.com. Gogrid cloud hosting `http://www.gogrid.com/products/cloud-hosting`, September 2012.

[15] Microsoft. Windows azure `http://www.windowsazure.com/en-us/`, September 2012.

[16] Inc. Eucalyptus Systems. Eucalyptus cloud `http://www.eucalyptus.com/eucalyptus-cloud`, September 2012.

[17] University of Chicago. Nimbus cloud `http://www.nimbusproject.org/`, September 2012.

[18] OpenNebula Project (OpenNebula.org). Opennebula open-source cloud `http://www.opennebula.org/`, September 2012.

[19] OpenStack.org. Openstack: the open source cloud operating system `http://www.openstack.org/`, September 2012.

[20] Google Inc. Google appengine `https://appengine.google.com/`, September 2012.

[21] SalesForce.com Inc. Force.com platform `http://www.force.com/`, September 2012.

[22] Heroku.com. Heroku cloud application platform `http://www.heroku.com/`, September 2012.

[23] UCSB. Appscale framework http://appscale.cs.ucsb.edu/, September 2012.

[24] TyphoonAE. Typhoonae http://code.google.com/p/typhoonae/, September 2012.

[25] Red Hat Inc. Openshift auto-scaling paas for applications https://openshift.redhat.com/app/, September 2012.

[26] Dustin Owens. Securing elasticity in the cloud. *Queue*, 8(5):10–16, 2010.

[27] Luis Rodero-Merino, Luis M. Vaquero, Víctor Gil, Fermín Galán, Javier Fontán, Rubén Montero, and Ignacio M. Llorente. From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26:1226–1240, October 2010.

[28] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. *IEEE International Symposium on Cluster Computing and the Grid*, 0:43–52, 2010.

[29] Microsoft. Microsoft entreprise library integration pack for azure http://msdn.microsoft.com/en-us/library/hh680918(v=pandp.50).aspx, September 2012.

[30] Paraleap Technologies LLC. Paraleap windows azure dynamic scaling http://www.paraleap.com/, September 2012.

[31] GoGrid.com. Gogrid exchange http://exchange.gogrid.com/, September 2012.

[32] OpenNebula project. OpenNebula Sservice Management Project http://dev.opennebula.org/projects/oneservice, September 2012.

[33] RightScale Inc. Rightscale cloud management http://www.rightscale.com/, September 2012.

[34] enStratus Networks Inc. enstratus http://www.enstratus.com, September 2012.

[35] Scalr Inc. Scalr cloud management http://scalr.net/, September 2012.

[36] T.C.K. Chou and J.A. Abraham. Load balancing in distributed systems. *Software Engineering, IEEE Transactions on*, SE-8(4):401 – 412, 1982.

[37] Amazon Web Services LLC. Aws elastic load balancing http://aws.amazon.com/elasticloadbalancing, September 2012.

[38] Microsoft. Windows azure load balancing virtual machines http://www.windowsazure.com/en-us/manage/windows/common-tasks/how-to-load-balance-virtual-machines/, September 2012.

[39] GoGrid.com. Gogrid load balancing services http://www.gogrid.com/cloud-hosting/load-balancers.php, September 2012.

[40] Rackspace US Inc. Rackspace cloud load balancers www.rackspace.com/cloud/public/loadbalancers/, September 2012.

[41] OpenNebula Project (OpenNebula.org). Opennebula scheduling policies http://www.opennebula.org/documentation:rel2.0:schg, September 2012.

[42] Amazon Web Services LLC. Amazon cloudwatch http://aws.amazon.com/cloudwatch/, September 2012.

[43] Microsoft. Monitoring windows azure applications http://msdn.microsoft.com/en-us/library/windowsazure/gg676009.aspx, September 2012.

[44] Rackspace US Inc. Rackspace cloud tools http://www.rackspace.com/cloud/tools/, September 2012.

[45] Cloudkick. Cloudkick cloud monitoring https://www.cloudkick.com/, September 2012.

[46] Nagios Entreprises LLC. Nagios infrastructure monitoring http://nagios.org/, September 2012.

[47] John Bresnahan, Tim Freeman, David LaBissoniere, and Kate Keahey. Managing appliance launches in infrastructure clouds. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, pages 12:1–12:7, New York, NY, USA, 2011. ACM.

[48] Eucalyptus.com. Eucalyptus monitoring http://open.eucalyptus.com/wiki/EucalyptusMonitoring_v1.6, September 2012.

[49] Federico D. Sacerdoti, Mason J. Katz, Matthew L. Massie, and David E. Culler. Wide area cluster monitoring with ganglia. *Cluster Computing, IEEE International Conference on*, 0:289, 2003.

[50] OpenNebula Project (OpenNebula.org). Opennebula information manager http://www.opennebula.org/documentation:rel2.0:img, September 2012.

[51] Rich Wolski, Neil T Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757 – 768, 1999.

[52] Matthew J. Sottile and Ronald G. Minnich. Supermon: A high-speed cluster monitoring system. *Cluster Computing, IEEE International Conference on*, 0:39, 2002.

[53] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[54] Tiago C. Ferreto, Cesar A. F. de Rose, and Luiz de Rose. Rvision: An open and high configurable tool for cluster monitoring. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:75, 2002.

[55] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, July 2009.

[56] Apache Software Foundation. delta-cloud `http://deltacloud.apache.org/`, September 2012.

[57] Andy Edmonds, Thijs Metsch, Alexander Papaspyrou, and Alexis Richardson. Open Cloud Computing Interface: Open Community Leading Cloud Standards. *ERCIM News*, (83):23–24, October 2010.

[58] Mark Crovella and Azer Bestavros. Explaining world wide web traffic self-similarity. Technical report, Boston, MA, USA, 1995.

[59] Jonathan Kupferman, Jeff Silverman, Patricio Jara, and Jeff Browne. Scaling into the cloud. Technical report, 552 University Road, CA 93106, 2009.

[60] Fermín Galán, Americo Sampaio, Luis Rodero-Merino, Irit Loy, Victor Gil, and Luis M. Vaquero. Service specification in cloud environments based on extensions to open standards. In *COMSWARE '09: Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, pages 1–12, New York, NY, USA, 2009. ACM.

[61] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Proceedings of the 10 th IEEE/ACM International Conference on Grid Computing (Grid 2009)*, pages 50 – 57, 2009.

[62] Alexandru Iosup, Dick H. J. Epema, Carsten Franke, Alexander Papaspyrou, Lars Schley, Baiyi Song, and Ramin Yahyapour. On grid performance evaluation using synthetic workloads. In *JSSPP'06: Proceedings of the 12th international conference on Job scheduling strategies for parallel processing*, pages 232–255, Berlin, Heidelberg, 2007. Springer-Verlag.

[63] Alexandru Iosup and Dick Epema. Grenchmark `http://grenchmark.st.ewi.tudelft.nl/`, September 2012.

[64] Alexandru Iosup, , Ru Iosup, Dick H. J. Epema, Jason Maassen, and Rob Van Nieuwpoort. Synthetic grid workloads with ibis, koala, and grenchmark. In *In Proceedigs of the CoreGRID Integrated Research in Grid Computing*, 2005.

[65] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, Rutger F. H. Hofman, Ceriel J. H. Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient java-based grid programming environment: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(7-8):1079–1107, 2005.

[66] Hashim Mohamed and Dick Epema. Koala: a co-allocating grid scheduler. *Concurr. Comput. : Pract. Exper.*, 20(16):1851–1876, 2008.

[67] Nikolaos Doulamis, Anastasios Doulamis, Antonios Litke, Athanasios Panagakis, Theodora Varvarigou, and Emmanuel Varvarigos. Adjusted fair scheduling and non-linear workload prediction for qos guarantees in grid computing. *Computer Communications*, 30(3):499 – 515, 2007. Special Issue: Emerging Middleware for Next Generation Networks.

[68] Radu Prodan and Vlad Nae. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems*, 25(7):785 – 793, 2009.

[69] William I. Chang and Thomas G. Marr. Approximate String Matching and Local Similarity. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 259–273, London, UK, 1994. Springer-Verlag.

[70] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms, Chapter 32: String Matching.* McGraw-Hill Higher Education, 2001.

[71] TUDelft. The grid workload archive `http://gwa.ewi.tudelft.nl/pmwiki/`, September 2012.

[72] WLCG. Worldwide lhc computing grid `http://wlcg.web.cern.ch/`, September 2012.

[73] The Nordugrid Collaboration. Nordugrid `http://www.nordugrid.org/`, September 2012.

[74] The shared Hierarchical Academic Research Computing Network. Sharcnet `https://www.sharcnet.ca/my/front/`, September 2012.

[75] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in grid computing. pages 1507–1542. Wiley Press, 2002.

[76] Aminul Haque, Saadat M. Alhashmi, and Rajendran Parthiban. A survey of economic models in grid computing. *Future Gener. Comput. Syst.*, 27(8):1056–1069, October 2011.

[77] Rajkumar Buyya and Manzur M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *CoRR*, cs.DC/0203019, 2002.

[78] Eddy Caron, Frederic Desprez, David Loureiro, and Adrian Muresan. Cloud computing resource management through a grid middleware: A case study with DIET and EUCALYPTUS. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09, pages 151–154, Washington, DC, USA, 2009. IEEE Computer Society.

[79] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 19–24, New York, NY, USA, 2009. ACM.

[80] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[81] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky computing. *IEEE Internet Computing*, 13:43–51, 2009.

[82] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 7–7, Berkeley, CA, USA, 2005. USENIX Association.

[83] James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Market-oriented Grids and Utility Computing: The State-of-the-art and Future Directions. *Journal of Grid Computing*, 6(3):255–276, September 2008.

[84] Brent N. Chun, Philip Buonadonna, Alvin Auyoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *In Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, 2005.

[85] L.M. Minga, Yu-Qiang Feng, and Yi-Jun Li. Dynamic pricing: ecommerce - oriented price setting algorithm. In *Machine Learning and Cybernetics, 2003 International Conference on*, volume 2, pages 893 – 898 Vol.2, nov. 2003.

[86] Chee Shin Yeo and Rajkumar Buyya. Pricing for utility-driven resource management and allocation in clusters. *Int. J. High Perform. Comput. Appl.*, 21(4):405–418, November 2007.

[87] Arutyun I. Avetisyan, Roy Campbell, Indranil Gupta, Michael T. Heath, Steven Y. Ko, Gregory R. Ganger, Michael A. Kozuch, David O'Hallaron, Marcel Kunze, Thomas T. Kwan, Kevin Lai, Martha Lyons, Dejan S.

Milojicic, Hing Yan Lee, Yeng Chai Soh, Ng Kwang Ming, Jing-Yuan Luke, and Han Namgoong. Open cirrus: A global cloud computing testbed. *Computer*, 43(4):35–43, April 2010.

[88] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán. The reservoir model and architecture for open federated cloud computing. *IBM J. Res. Dev.*, 53(4):535–545, July 2009.

[89] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. *CoRR*, abs/1003.3920, 2010.

[90] Shantenu Jha, Andre Merzky, and Geoffrey Fox. Using clouds to provide grids with higher levels of abstraction and explicit support for usage modes. *Concurr. Comput. : Pract. Exper.*, 21(8):1087–1108, June 2009.

[91] Michael A. Murphy and Sebastien Goasguen. Virtual organization clusters: Self-provisioned clouds on the grid. *Future Gener. Comput. Syst.*, 26(8):1271–1281, October 2010.

[92] Manuel Rodríguez, Daniel Tapiador, Javier Fontán, Eduardo Huedo, Rubén S. Montero, and Ignacio M. Llorente. Euro-par 2008 workshops - parallel processing. chapter Dynamic Provisioning of Virtual Clusters for Grid Computing, pages 23–32. Springer-Verlag, Berlin, Heidelberg, 2009.

[93] Constantino Vázquez, Eduardo Huedo, Rubén S. Montero, and Ignacio M. Llorente. On the use of clouds for grid resource provisioning. *Future Gener. Comput. Syst.*, 27(5):600–605, May 2011.

[94] R. Buyya, D. Abramson, and S. Venugopal. The Grid Economy. In M. Parashar and C. Lee, editors, *Proceedings of the IEEE*, volume 93 of *Special Issue on Grid Computing*, pages 698–714. IEEE Press, New Jersey, USA, Mar 2005.

[95] Jahanzeb Sherwani, Nosheen Ali, Nausheen Lotia, Zahra Hayat, and Rajkumar Buyya. Libra: a computational economy-based job scheduling system for clusters. *Softw. Pract. Exper.*, 34(6):573–590, May 2004.

[96] Richard Wolski, James S. Plank, John Brevik, and Todd Bryan. G-commerce: Market formulations controlling resource allocation on the computational grid. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 46–, Washington, DC, USA, 2001. IEEE Computer Society.

[97] 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, 2004.

[98] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, August 2005.

[99] Brent N. Chun and David E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *In 2nd IEEE International Symposium on Cluster Computing and the Grid*, pages 30–38, 2002.

[100] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04, pages 160–169, Washington, DC, USA, 2004. IEEE Computer Society.

[101] Florentina I. Popovici and John Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 36–, Washington, DC, USA, 2005. IEEE Computer Society.

[102] Alvin Auyoung, Laura Grit, Janet Wiener, and John Wilkes. Service contracts and aggregate utility functions. In *In Proceedings of the IEEE Symposium on High Performance Distributed Computing*, pages 119–131, 2006.

[103] Peter Couvares, Tevik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. Workflow Management in Condor. In Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shields, editors, *Workflows for e-Science*, pages 357–375. Springer, 2007.

[104] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph Jacob, and Daniel Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[105] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Jr., and Hong Linh Truong. ASKALON: a Tool Set for Cluster and Grid Computing. *Concurrency - Practice and Experience*, 17(2-4):143–169, 2005.

[106] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and Efficient Workflow Deployment of Data-Intensive Applications on GRIDS with MOTEUR. *Intl. Journal of High Performance Computing Applications*, 3(22):347–360, 2008. Special issue on Workflow Systems in Grid Environments.

[107] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[108] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[109] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The Triana Workflow Environment: Architecture and Applications. *Workflows for e-Science*, pages 320–339, 2007.

[110] Ilkay Altintas, Sangeeta Bhagwanani, David Buttler, Sandeep Chandra, Zhengang Cheng, Matthew Coleman, Terence Critchlow, Amarnath Gupta, Wei Han, Ling Liu, Bertram Ludäscher, Calton Pu, Reagan Moore, Arie Shoshani, and Mladen A. Vouk. A modeling and execution environment for distributed scientific workflows. In *Proc. of the 15th Intl. Conference on Scientific and Statistical Database Management*, pages 247–250, 2003.

[111] T Werner. Target Gene Identification from Expression Array Data by Promoter Analysis. *Biomolecular Engineering*, 17(3):87–94, 2001.

[112] Anthony Mayer, Steve McGough, Nathalie Furmento, William Lee, Steven Newhouse, and John Darlington. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In *UK e-Science All Hands Meeting*, pages 627–634. IOP Publishing Ltd, 2003.

[113] Emir M. Bahsi, Emrah Ceyhan, and Tevfik Kosar. Conditional Workflow Management: A Survey and Analysis. *Scientific Programming*, 15(4):283–297, 2007.

[114] Andrei Radulescu and Arjan van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proc. of the 15th Intl. Conference on Parallel Processing (ICPP)*, pages 69–76, 2001.

[115] Frédéric Desprez and Frédéric Suter. A Bi-Criteria Algorithm for Scheduling Parallel Task Graphs on Clusters. In *Proc. of the 10th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*, pages 243–252, 2010.

[116] Hector Fernandez, Cédric Tedeschi, and Thierry Priol. A Chemistry Inspired Workflow Management System for Scientific Applications in Clouds. In *Proc. of the 7th Intl. Conference on E-Science*, pages 39–46, 2011.

[117] Mirko Viroli and Franco Zambonelli. A Biochemical Approach to Adaptive Service Ecosystems. *Information Sciences*, 180(10):1876–1892, 2010.

[118] Ming Mao and Marty Humphrey. Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis, (SC'11)*, 2011.

[119] Wil van der Aalst, Alistair Barros, Arthur ter Hofstede, and Bartek Kiepuszewski. Advanced Workflow Patterns. In *Proc. of the 7th Intl. Conference on Cooperative Information Systems*, pages 18–29, 2000.

[120] Amazon Web Services LLC. Amazon ec2 instance types `http://aws.amazon.com/ec2/instance-types/`, September 2012.

[121] Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay Fornarino. A data-driven workflow language for grids based on array programming principles. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.

[122] G. von Laszewski, G.C. Fox, Fugang Wang, A.J. Younge, A. Kulshrestha, G.G. Pike, W. Smith, J. Vöckler, R.J. Figueiredo, J. Fortes, and K. Keahey. Design of the futuregrid experiment management framework. In *Gateway Computing Environments Workshop (GCE), 2010*, pages 1 – 10, nov. 2010.

[123] The Economist. Clash of the clouds `http://www.economist.com/node/14637206`, September 2012.

[124] Neal Leavitt. Is cloud computing really ready for prime time? *Computer*, 42(1):15–20, January 2009.

[125] John Viega. Cloud computing and the common man. *Computer*, 42(8):106–108, August 2009.

[126] Hassan Takabi, James B. D. Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security and Privacy*, 8(6):24–31, November 2010.

[127] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[128] Craig D. Weissman and Steve Bobrowski. The design of the force.com multitenant internet application development platform. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 889–896, New York, NY, USA, 2009. ACM.

[129] Trent Jaeger. *Operating System Security*. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan & Claypool Publishers, 2008.

[130] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[131] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '03, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association.

[132] Patrice Clemente, Jonathan Rouzaud-Cornabas, and Christian Toinard. Transactions on computational science xi. chapter From a generic framework for expressing integrity properties to a dynamic MAC enforcement for operating systems, pages 131–161. Springer-Verlag, Berlin, Heidelberg, 2010.

[133] Xuxian Jiang and Dongyan Xu. Protection of application service hosting platforms: an operating system perspective. Technical Report 03-010, Purdue University, 2003.

[134] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[135] Java Community Process. Java Specification Request 220: Enterprise Java Beans 3.0 `http://www.jcp.org/en/jsr/detail?id=220`.

[136] Java Community Process. Java specification request 154: Java Servlet 2.5 Specification `http://www.jcp.org/en/jsr/detail?id=154`.

[137] Java Community Process. Java specification requests jsr 244: Java Platform Enterprise Edition 5 (Java EE Specification) `http://www.jcp.org/en/jsr/detail?id=244`.

[138] OSGI Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.2. Technical report, OSGI Alliance, September 2009.

[139] Scott Oaks. *Java security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[140] Li Gong. *Inside Java 2 platform security architecture, API design, and implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[141] Jose M. Alcaraz Calero, Nigel Edwards, Johannes Kirschnick, Lawrence Wilcock, and Mike Wray. Toward a multi-tenancy authorization system for cloud services. *IEEE Security and Privacy*, 8(6):48–55, November 2010.

[142] Almut Herzog and Nahid Shahmehri. Problems running untrusted services as java threads. In Enrico Nardelli and Maurizio Talamo, editors, *Certification and Security in Inter-Organizational E-Service*, volume 177 of *IFIP On-Line Library in Computer Science*, pages 19–32. Springer US, 2005.

[143] Walter Binder. Secure and reliable java-based middleware - challenges and solutions. In *Proceedings of the First International Conference on Availability, Reliability and Security*, ARES '06, pages 662–669, Washington, DC, USA, 2006. IEEE Computer Society.

[144] Grzegorz Czajkowski and Laurent Daynés. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 125–138, New York, NY, USA, 2001. ACM.

[145] Kewei Sun, Ying Li, Matt Hogstrom, and Ying Chen. Sizing multi-space in heap for application isolation. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 647–648, New York, NY, USA, 2006. ACM.

[146] Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer. A multi-user virtual machine. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '03, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[147] Java Community Process. Java specification requests jsr 121: Application isolation api specification `http://www.jcp.org/en/jsr/detail?id=121`.

[148] Godmar Back and Wilson C. Hsieh. The kaffeos java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, July 2005.

[149] Nicolas Geoffray, Gaël Thomas, Bertil Folliot, and Charles Clément. Towards a new isolation abstraction for osgi. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 41–45, New York, NY, USA, 2008. ACM.

[150] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java Virtual Machine for Component Isolation in OSGi. In *International Conference on Dependable Systems and Networks (DSN 2009)*, Estoril, Portugal, June 2009. IEEE Computer Society.

[151] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. Vmkit: a substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference*

on Virtual execution environments, VEE '10, pages 51–62, New York, NY, USA, 2010. ACM.

[152] David W. Price, Algis Rudys, and Dan S. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 263–, Washington, DC, USA, 2003. IEEE Computer Society.

[153] Grzegorz Czajkowski and Thorsten von Eicken. Jres: a resource accounting interface for java. *SIGPLAN Not.*, 33(10):21–35, October 1998.

[154] Walter Binder and Jarle G. Hulaas. Portable resource control in java - the j-seal2 approach. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, pages 139–155. ACM Press, 2001.

[155] Vladimir Omar Calderon Yaksic. J-raf – the java resource accounting facility. Master's thesis, Université de Genève, Département d'informatique, 24 rue du Général-Dufour, Geneva, Switzerland, 2002.

[156] Jarle Hulaas and Walter Binder. Program transformations for portable cpu accounting and control in java. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '04, pages 169–177, New York, NY, USA, 2004. ACM.

[157] Walter Binder and Jarle Hulaas. Exact and portable profiling for the jvm using bytecode instruction counting. *Electr. Notes Theor. Comput. Sci.*, 164(3):45–64, 2006.

[158] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. A resource management interface for the java™ platform. Technical report, Mountain View, CA, USA, 2003.

[159] Java Community Process. Java specification requests jsr 284: Resource Consumption Management API `http://www.jcp.org/en/jsr/detail?id=284`.

[160] Algis Rudys and Dan S. Wallach. Termination in language-based systems. *ACM Trans. Inf. Syst. Secur.*, 5(2):138–168, May 2002.

[161] Almut Herzog and Nahid Shahmehri. An evaluation of java application containers according to security requirements. In *Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, WETICE '05, pages 178–186, Washington, DC, USA, 2005. IEEE Computer Society.

[162] Pankaj Kumar. *J2ee; security for servlets, ejbs and web services: applying theory and standards to practice.* Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2003.

[163] Marco Pistoia, Nataraj Nagaratnam, Larry Koved, and Anthony Nadalin. *Enterprise Java 2 Security: Building Secure and Robust J2EE Applications*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[164] Mick Jordan, Laurent Daynès, Marcin Jarzab, Ciarán Bryce, and Grzegorz Czajkowski. Scaling j2ee application servers with the multi-tasking virtual machine. *Softw. Pract. Exper.*, 36(6):557–580, May 2006.

[165] Mick Jordan, Grzegorz Czajkowski, Kirill Kouklinski, and Glenn Skinner. Extending a j2ee server with dynamic and flexible resource management. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 439–458, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[166] P. Parrend and S. Frenot. Security benchmarks of osgi platforms: toward hardened osgi. *Softw. Pract. Exper.*, 39(5):471–499, April 2009.

[167] Kiev Gama and Didier Donsez. Towards dynamic component isolation in a service oriented platform. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 104–120, Berlin, Heidelberg, 2009. Springer-Verlag.

[168] Heejune Ahn, Hyukjun Oh, and Chang Oan Sung. Towards reliable osgi framework and applications. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1456–1461, New York, NY, USA, 2006. ACM.

[169] Scott Oaks. *Java vs. .NET Security*. O'Reilly & Associates, Inc., May 2004.

[170] Nikos Parlavantzas, Geoff Coulson, and Gordon Blair. A resource adaptation framework for reflective middleware. In *Proc. 2nd Intl. Workshop on Reflective and Adaptive Middleware (located with ACM/IFIP/USENIX Middleware 2003), Rio de*, 2003.

[171] C. Escoffier, D. Donsez, and R.S. Hall. Developing an osgi-like service platform for .net. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 1, pages 213 – 217, jan. 2006.

[172] Nicolas Halbwachs and Lenore D. Zuck, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*. Springer, 2005.

[173] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: A software platform for .net-based cloud computing. *CoRR*, abs/0907.4622, 2009.

[174] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[175] Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, NSPW '99, pages 87–95, New York, NY, USA, 2000. ACM.

[176] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Form. Methods Syst. Des.*, 24(2):129–155, March 2004.

[177] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. *SIGPLAN Not.*, 40(6):305–314, June 2005.

[178] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, October 2001.

[179] J. Viega, J. T. Bloch, and P. Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14(2):31–39, February 2001.

[180] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 11–20, New York, NY, USA, 2008. ACM.

[181] Shu Gao, Yi Deng, Huiqun Yu, Xudong He, Konstantin Beznosov, and Kendra Cooper. Applying aspect-orientation in designing security systems: A case study. In *Proceedings of International Conference of Software Engineering and Knowledge Engineering*, 2004.

[182] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 333–344, New York, NY, USA, 1998. ACM.