Universitatea "Politehnica" București

Facultatea de Automatică și Calculatoare,
Catedra de Calculatoare

LUCRARE DE DISERTAȚIE

ReC^{2}S: Sistem Cloud Computing de Încredere

Conducător Științific:
As. dr. ing. Cătălin Leordeanu
Prof. dr. ing. Valentin Cristea

Autor:
Ing. Alecsandru Pătraşcu

București, 2012

University "Politehnica" of Bucharest

Faculty of Automatic Control and Computer Science,
Computer Science and Engineering Department

MASTER THESIS

ReC^{2}S: Reliable Cloud Computing System

Scientific Adviser:
As. dr. ing. Cătălin Leordeanu
Prof. dr. ing. Valentin Cristea

Author:
Ing. Alecsandru Pătraşcu

Bucharest, 2012

Abstract

Cloud Computing is becoming one of the next computer science buzz words. It joins the ranks of terms including: Grid Computing, Utility Computing, virtualization and clustering. Cloud Computing overlaps some of the concepts of distributed computing. However it does have its own meaning if contextually used correctly. The conceptual overlap is partly due to technology changes, usages and implementations over the years. If Cloud Computing is becoming widespread as interest in it is rapidly growing, on the other part, there has been a decline in general interest of Grid, Utility and Distributed computing. Nevertheless, all these terms will be around in usage for quite a while to come.

In this thesis we focus on providing essential characteristics such as performance, availability, reliability and security for Cloud Computing systems, which are becoming more and more popular. In this document we accurately describe the capabilities that our project provides to its end-users and also detail all the functional and non-functional requirements that the application is implementing. We offer a complete solution for Cloud Computing management, describing also our design choices for the final application, along with how we integrated our solution into existing virtualization and Cloud Computing software.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The term Cloud Computing comes from the use of a Cloud image to represent the Internet or some large network environment. The users are not interested about what is in the Cloud or what goes on there, except the fact that they depend on reliably sending data to and receiving data from it. Cloud Computing is now associated with a higher level abstraction of the existing computing environments. Instead of using data pipes, routers and servers, there are now services. The underlying hardware and software of networking is of course still there but now there are higher level service capabilities available in use for building an application. Behind the services data and compute resources exist. A service user does not necessarily care about how it is implemented, what technologies are used or how it is managed; he only cares about the fact that there is access to it and has a level of reliability necessary to meet the application requirements.

In essence this is **"Distributed Computing"**: an application is built using the resource from multiple services potentially from multiple locations. At this point, typically we still need to know the endpoint to access the services rather than having the Cloud provide the available resources. This is also know as "Software as a Service": behind the service interface is usually a grid of computers to provide the resources. The grid is typically hosted by one company and consists of a homogeneous environment of hardware and software making it easier to support and maintain. Once the user starts paying for the services and the resources utilized, we can call this **"utility computing"**.

Cloud Computing represents in reality a way of accessing resources and services needed to perform functions with dynamically changing needs. An application or service developer requests access from the Cloud rather than a specific endpoint or named resource. What goes on in the Cloud is a process that manages multiple infrastructures across multiple organizations and consists of one or more frameworks overlaid on top of the infrastructures, tying them together. Frameworks provide mechanisms for self-healing, self monitoring, resource registration and discovery, Service Level Agreement definitions and automatic reconfiguration.

There are a lot of discussions around the concept of Cloud Computing from the perspective of a final user. But what exactly does a Cloud user wants to receive from the providers? We can answer this question by explaining some of the points that Cloud Computing is touching as a business model for expenses optimization. This happens from two sides of the story.

The first one is, of course, the consumer of the Cloud services, that do not have to invest into the resources up-front in order to have the capacities and functionality meeting their demands. Neither do they have to over-invest to that the capacities meeting their peak demands, like having extra servers in a rack just to handle holiday sales spikes. Consumers are also saved from the maintenance-related expenses and gain the flexibility of getting rid of the resources, should such a business need arise. Getting rid of a data center is a bit harder, though.

The second one involves the providers of Cloud Computing that could utilize their existing resources more efficiently by shifting them into the Cloud and linking them with flexible pricing strategies. For example, we can have a lot of idle CPU cycles scattered around a datacenter on weekend nights in Romania. Would it

not be more efficient to sell them for the computing purposes, at a flexible rate, instead of just wasting them? This ability to optimize usage of resources at various levels of the picture is one of the primary factors why the Cloud Computing hype is more than a temporary buzzword.

Because of the reasons explained so far, a fair observation can be made: Cloud Computing is a new, raw and tough research domain because it involves knowledge from many other domains like Distributed Computing, algorithms and data structures, networking protocols and many other, in order for it to function properly. The main concerns that are rising are influenced by the transition from the regular basic server infrastructure held by users to a centralized datacenter maintained by a third-party provider.

As a result, we have developed a solution for this problem. Our system can help datacenter owners, administrators and Cloud Computing providers to manage more easily the kind of workloads involved in this new way of doing computing, maintaining also the security to a high level. Using ReC^{2}S, the end-user will benefit from a robust solution that automatically scales vertically and horizontally to its computational needs.

We have organized this thesis as follows. ***Chapter 2*** contains a brief description of the Cloud Computing and Large Scale Distributed Computing domain and paradigm along with a short history of it up to date. ***Chapter 3*** presents the general context in which our system fits and also point out essential data and security issues involved. Also we present some of the most important related work on this field that has influenced our design and application in the form it is implemented. In ***Chapter 4*** we present in detail the system architecture starting from the functional and non-functional requirements and ending with the actual software design details. ***Chapter 5*** is reserved entirely for our core functionality - the scheduler. We presented in detail it's general structure, the functionality, architecture and how it works on real workloads. In ***Chapter 6*** we present briefly the software stack that we used to implement ReC^{2}S along with explanations about it. In ***Chapter 7*** we present some of the results coming from our scheduler and the system as a whole. ***Chapter 8*** contains the conclusions for our thesis and an outline of the directions for future research.

Chapter 2

Cloud Computing

2.1 Roots of Cloud Computing

We can track the roots of clouds computing by observing the advancement of several technologies, especially in hardware (virtualization and multi-core chips), Internet technologies (Web services, service-oriented architectures, Web 2.0), Distributed Computing (clusters and grids), and systems management (Autonomic Computing, datacenter automation). Figure 2.1 shows the convergence of technology fields that significantly advanced and contributed to the advent of Cloud Computing.

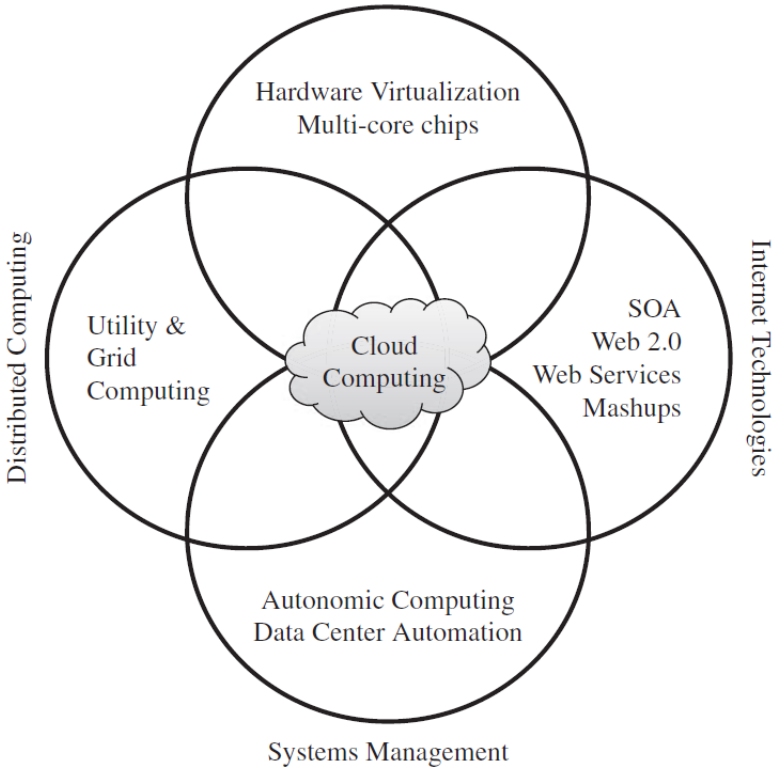

Figure 2.1: *Convergence of various advances leading to the advent of Cloud Computing.*

Some of these technologies have been tagged as hype in their early stages of development; however, they later received significant attention from academia and were sanctioned by major industry players. Consequently,

a specification and standardization process followed, leading to maturity and wide adoption. The emergence of Cloud Computing itself is closely linked to the maturity of such technologies. We present in our thesis a closer look at the technologies that form the base of Cloud Computing, with the aim of providing a clearer picture of the cloud ecosystem as a whole.

2.1.1 From mainframes to clouds

We are currently experiencing a switch in the IT world, from in-house generated computing power into utility-supplied computing resources delivered over the Internet as Web services. This trend is similar to what occurred about a century ago when factories, which used to generate their own electric power, realized that it is was cheaper to just plug-in their machines into the newly formed electric power grid [17].

Computing delivered as a utility can be defined as "on demand delivery of infrastructure, applications, and business processes in a security-rich, shared, scalable, and based computer environment over the Internet for a fee" [5]. This model brings benefits to both consumers and providers of IT services. Consumers can obtain reduction on IT-related costs by choosing to get cheaper services from external providers as opposed to heavily investing on IT infrastructure and personnel hiring. The "on-demand" component of this model allows consumers to adapt their IT usage to rapidly increasing or unpredictable computing needs.

Providers of IT services achieve better operational costs; hardware and software infrastructures are built to provide multiple solutions and serve many users, thus increasing efficiency and ultimately leading to faster Return On Investment (ROI) as well as lowering Total Cost of Ownership (TCO) [6]. Several technologies have, in some way, aimed at turning the utility computing concept into reality. In the 1970s, companies who offered common data processing tasks, such as payroll automation, operated time-shared mainframes as utilities, which could serve dozens of applications and often operated close to 100% of their capacity. In fact, mainframes had to operate at very high utilization rates simply because they were very expensive and costs should be justified by efficient usage [17].

The mainframe era collapsed with the advent of fast and inexpensive microprocessors and IT data centers moved to collections of commodity servers. Apart from its clear advantages, this new model inevitably led to isolation of workload into dedicated servers, mainly due to incompatibilities between software stacks and operating systems [7]. In addition, the unavailability of efficient computer networks meant that IT infrastructure should be hosted in proximity to where it would be consumed. Altogether, these facts have prevented the utility computing reality of taking place on modern computer systems.

Similar to old electricity generation stations, which used to power individual factories, computing servers and desktop computers in a modern organization are often underutilized, since IT infrastructures are configured to handle theoretical demand peaks. In addition, in the early stages of electricity generation, electric current could not travel long distances without significant voltage losses. However, new paradigms emerged culminating on transmission systems able to make electricity available hundreds of kilometers far off from where it is generated. Likewise, the advent of increasingly fast fiber-optics networks has reopened the fire, and new technologies for enabling sharing of computing power over great distances have appeared.

These facts reveal the potential of delivering computing services with the speed and reliability that businesses enjoy with their local machines. The benefits of economies of scale and high utilization allow providers to offer computing services for a fraction of what it costs for a typical company that generates its own computing power [17].

2.1.2 SOA, Web Services, Web 2.0, and mashups

The emergence of Web services (WS) open standards has significantly contributed to advances in the domain of software integration [8]. Web services can glue together applications running on different messaging product platforms, enabling information from one application to be made available to others, and enabling internal applications to be made available over the Internet.

Over the years a rich WS software stack has been specified and standardized, resulting in a multitude of technologies to describe, compose, and orchestrate services, package and transport messages between services, publish and discover services, represent Quality of Service (QoS) parameters, and ensure security in service access [9].

WS standards have been created on top of existing ubiquitous technologies such as HTTP and XML, thus providing a common mechanism for delivering services, making them ideal for implementing a service-oriented architecture (SOA). As seen in Figure 2.2, the purpose of a SOA is to address requirements of loosely coupled, standards-based, and protocol-independent Distributed Computing. In a SOA, software resources are packaged as "services", which are well-defined, self-contained modules that provide standard business functionality and are independent of the state or context of other services. Services are described in a standard definition language and have a published interface [8].

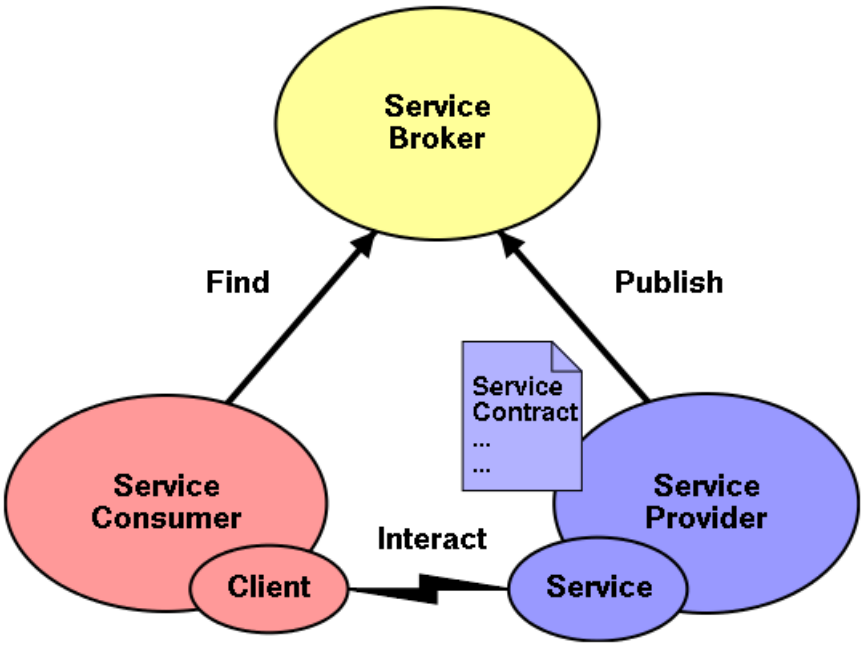

Figure 2.2: *Service Oriented Architecture*

The maturity of WS has enabled the creation of powerful services that can be accessed on-demand, in a uniform way. While some WS are published with the intent of serving end-user applications, their true power resides in its interface being accessible by other services. An enterprise application that follows the SOA paradigm is a collection of services that together perform complex business logic [8].

This concept of gluing services initially focused on the enterprise Web, but gained space in the consumer realm as well, especially with the advent of Web 2.0 (Figure 2.3). In the consumer Web, information and services may be programmatically aggregated, acting as building blocks of complex compositions, called service mashups. Many service providers, such as Amazon, del.icio.us, Facebook, and Google, make their service APIs publicly accessible using standard protocols such as SOAP and REST [10]. Consequently, one can put an idea of a fully functional Web application into practice just by gluing pieces with few lines of code.

In the Software as a Service (SaaS) domain, Cloud applications can be built as compositions of other services from the same or different providers. Services such as user authentication, e-mail, payroll management, and calendars are examples of building blocks that can be reused and combined in a business solution in case a single, ready-made system does not provide all those features.Many building blocks and solutions are now available in public marketplaces. For example, Programmable Web 1.0 is a public repository of service APIs and mashups currently listing thousands of APIs and mashups. Popular APIs such as Google Maps, Flickr, YouTube, Amazon eCommerce, and Twitter, when combined, produce a variety of interesting solutions, from

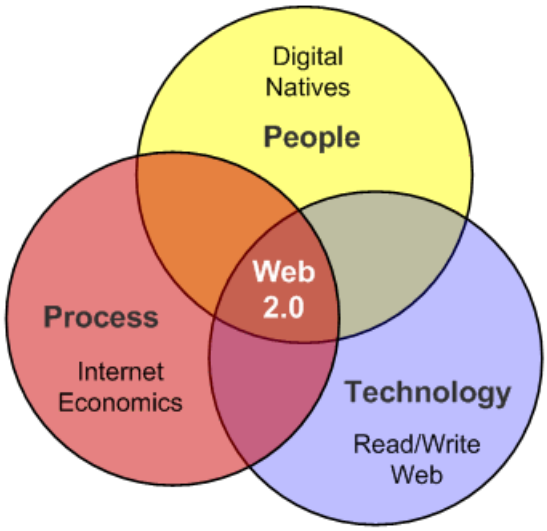

Figure 2.3: *Web 2.0 and service mashup concept*

finding video game retailers to weather maps. Similarly, Salesforce.com's offers AppExchange, which enables the sharing of solutions developed by third-party developers on top of Salesforce.com components.

2.1.3 Grid Computing

Grid Computing enables aggregation of distributed resources and transparently access to them. The architecture is complex (Figure 2.4), and most production grids such as TeraGrid [18] and EGEE [19] seek to share computational and storage resources distributed across different administrative domains, with their main focus being speeding up a broad range of scientific applications, such as climate modeling, drug design, and protein analysis.

A key aspect of the grid vision realization has been building standard WS-based protocols that allow distributed resources to be "discovered, accessed, allocated, monitored, accounted for, and billed for, etc., and in general managed as a single virtual system". The Open Grid Services Architecture (OGSA) addresses this need for standardization by defining a set of core capabilities and behaviors that address key concerns in grid systems.

Globus Toolkit [20] is a middleware that implements several standard Grid services and over the years has aided the deployment of several service-oriented Grid infrastructures and applications. An ecosystem of tools is available to interact with service grids, including grid brokers, which facilitate user interaction with multiple middleware and implement policies to meet QoS needs.

The development of standardized protocols for several Grid Computing activities has contributed theoretically to allow delivery of on-demand computing services over the Internet. However, ensuring QoS in grids has been perceived as a difficult endeavor [21]. Lack of performance isolation has prevented grids adoption in a variety of scenarios, especially on environments where resources are oversubscribed or users are uncooperative. Activities associated with one user or virtual organization (VO) can influence, in an uncontrollable way, the performance perceived by other users using the same platform. Therefore, the impossibility of enforcing QoS and guaranteeing execution time became a problem, especially for time-critical applications [22].

Another issue that has lead to frustration when using grids is the availability of resources with diverse software configurations, including disparate operating systems, libraries, compilers, runtime environments, and so forth. At the same time, user applications would often run only on specially customized environments.

Consequently, a portability barrier has often been present on most grid infrastructures, inhibiting users of adopting grids as utility computing environments [22].

Virtualization technology has been identified as the perfect fit to issues that have caused frustration when using grids, such as hosting many dissimilar software applications on a single physical platform. In this direction, some research projects like the Globus VirtualWorkspaces [22] are aimed at evolving grids to support an additional layer to virtualize computation, storage, and network resources.

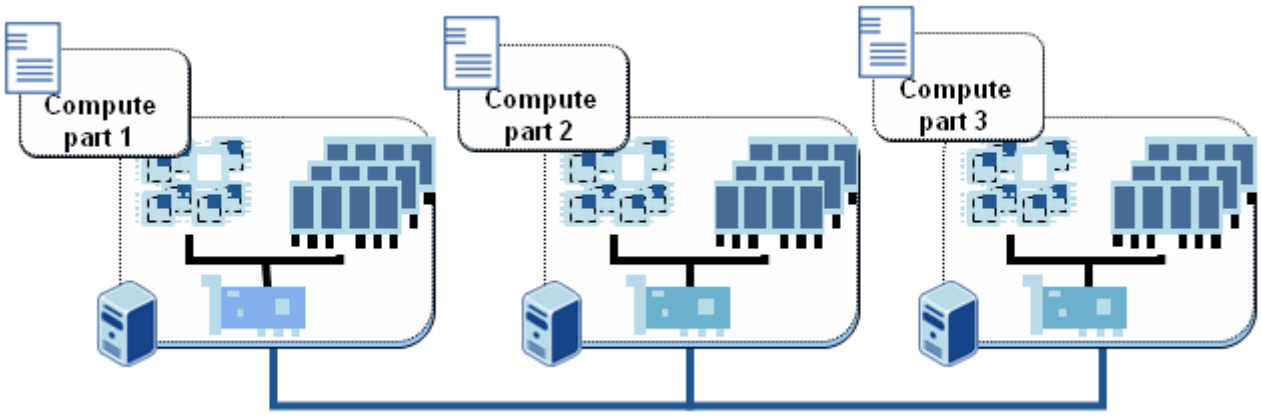

Figure 2.4: *GRID architecture overview*

2.1.4 Utility Computing

With increasing popularity and usage, large grid installations have faced new problems, such as excessive spikes in demand for resources coupled with strategic and adversarial behavior by users. Initially, grid resource management techniques did not ensure fair and equitable access to resources in many systems. Traditional metrics (throughput, waiting time, and slowdown) failed to capture the more subtle requirements of users. There were no real incentives for users to be flexible about resource requirements or job deadlines, nor provisions to accommodate users with urgent work.

In utility computing environments, users assign a "utility" value to their jobs, where utility is a fixed or time-varying valuation that captures various QoS constraints (deadline, importance, satisfaction). The evaluation is the amount they are willing to pay a service provider to satisfy their demands. The service providers then attempt to maximize their own utility, where said utility may directly correlate with their profit. Providers can choose to prioritize high yield (i.e., profit per unit of resource) user jobs, leading to a scenario where shared systems are viewed as a marketplace, where users compete for resources based on the perceived utility or value of their jobs. Further information and comparison of these utility computing environments are available in an extensive survey of these platforms [23].

2.1.5 Hardware virtualization

Cloud Computing services are usually backed by large-scale datacenters composed of thousands of computers. Such datacenters are built to serve many users and host many disparate applications. For this purpose, hardware virtualization can be considered as a perfect fit to overcome most operational issues of data center building and maintenance.

The idea of virtualizing a computer system resources, including processors, memory, and I/O devices, has been well established for decades, aiming at improving sharing and utilization of computer systems [24]. Hardware virtualization allows running multiple operating systems and software stacks on a single physical

platform. As presented in Figure 2.5, a software layer, the virtual machine monitor (VMM), also called a hypervisor, mediates access to the physical hardware presenting to each guest operating system a virtual machine (VM), which is a set of virtual platform interfaces [25].

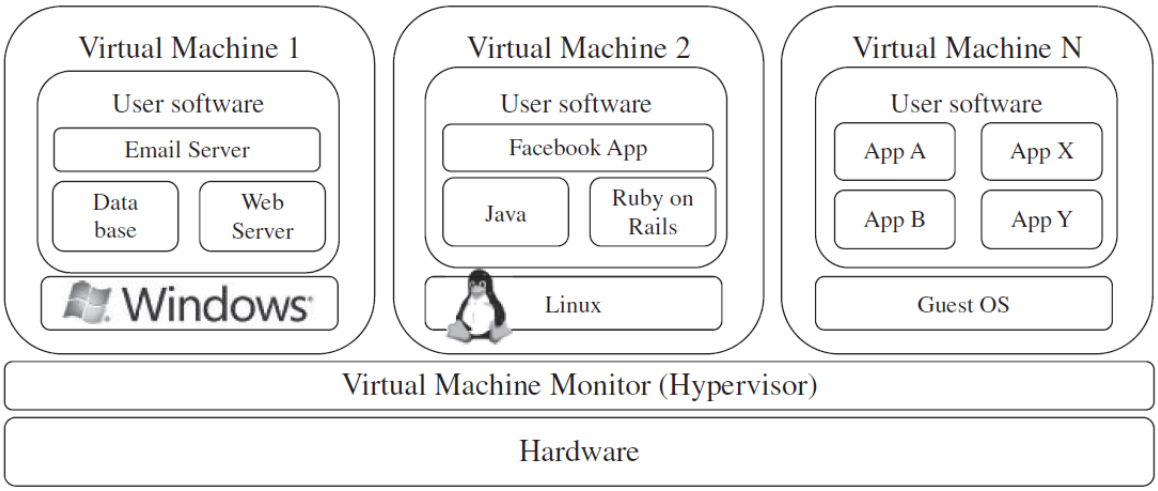

Figure 2.5: *A hardware virtualized server hosting three virtual machines, each one running distinct operating system and user level software stack.*

The advent of several innovative technologies multi-core chips, paravirtualization, hardware-assisted virtualization, and live migration of VMs has contributed to an increasing adoption of virtualization on server systems.

Traditionally, perceived benefits were improvements on sharing and utilization, better manageability and higher reliability. More recently, with the adoption of virtualization on a broad range of server and client systems, researchers and software programmers have been emphasizing three basic capabilities regarding management of workload in a virtualized system, namely isolation, consolidation, and migration [26].

Workload isolation is achieved since all program instructions are fully confined inside a VM, which leads to improvements in security. Better reliability is also achieved because software failures inside one VM do not affect others. Moreover, better performance control is attained since execution of one VM should not affect the performance of another VM [26]. The consolidation of several individual and heterogeneous workloads onto a single physical platform leads to better system utilization. This practice is also employed for overcoming potential software and hardware incompatibilities in case of upgrades, given that it is possible to run legacy and new operation systems concurrently [25].

Workload migration, also referred to as application mobility [26], targets at facilitating hardware maintenance, load balancing, and disaster recovery. It is done by encapsulating a guest OS state within a VM and allowing it to be suspended, fully serialized, migrated to a different platform, and resumed immediately or preserved to be restored at a later date [25]. A VM's state includes a full disk or partition image, configuration files, and an image of its RAM [22].

A number of VMM platforms exist that are the basis of many utility or cloud computing environments. The most notable ones - VMware, Xen, and KVM, are outlined in the following sections.

- **VMware ESX(i).** VMware is a pioneer in the virtualization market. Its ecosystem of tools ranges from server and desktop virtualization to high-level management tools [44]. ESX(i) is a VMM from VMware. It is a bare-metal hypervisor, meaning that it installs directly on the physical server, whereas others may require a host operating system. It provides advanced virtualization techniques of processor, memory, and I/O. Especially, through memory ballooning and page sharing, it can overcommit memory, thus increasing the density of VMs inside a single physical server (Figure 2.6).

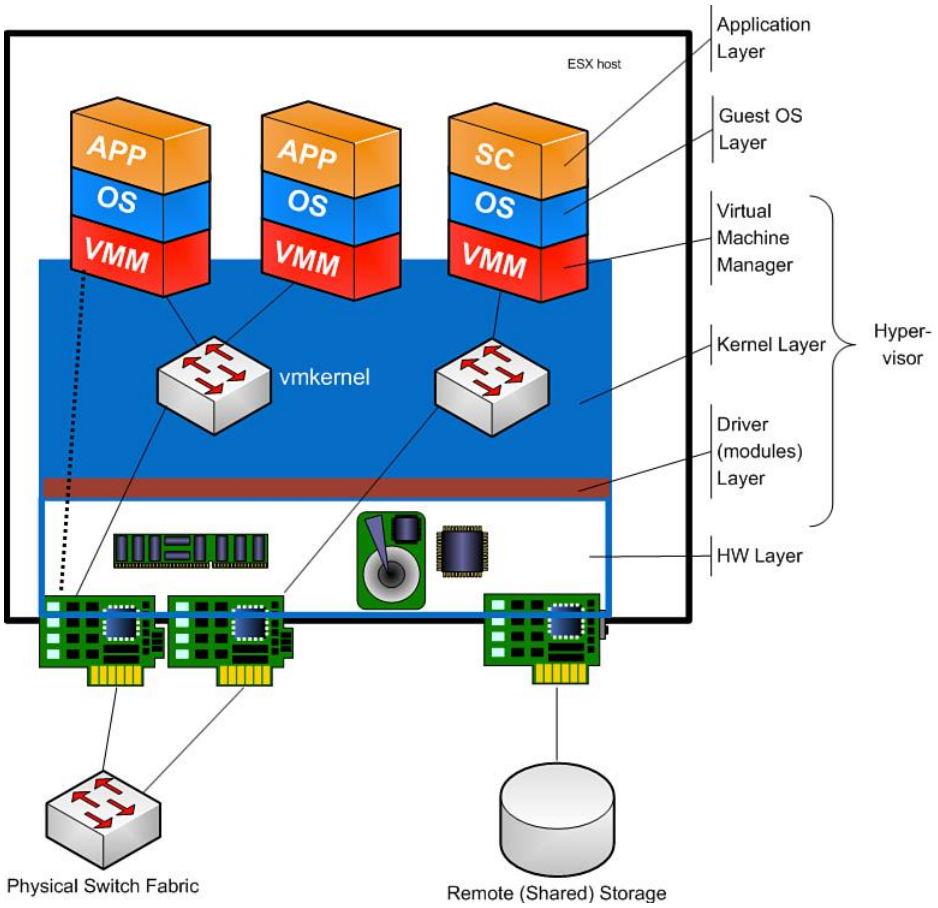

Figure 2.6: *VMware ESX(i) hypervisor architecture*

- **Xen.** The Xen hypervisor started as an open-source project and has served as a base to other virtualization products, both commercial and open-source. It has pioneered the paravirtualization concept, on which the guest operating system, by means of a specialized kernel, can interact with the hypervisor, thus significantly improving performance. In addition to an open-source distribution [27], Xen currently forms the base of commercial hypervisors of a number of vendors, most notably Citrix XenServer [28] and Oracle VM [29] (Figure 2.7).
- **KVM.** The kernel-based virtual machine (KVM) is a Linux virtualization subsystem. Is has been part of the mainline Linux kernel since version 2.6.20, thus being natively supported by several distributions. In addition, activities such as memory management and scheduling are carried out by existing kernel features, thus making KVM simpler and smaller than hypervisors that take control of the entire machine [30]. KVM leverages hardware-assisted virtualization, which improves performance and allows it to support unmodified guest operating systems [31]; currently, it supports several versions of Windows, Linux, and UNIX [30] (Figure 2.8).

2.1.6 Virtual Appliances and the Open Virtualization Format

An application combined with the environment needed to run it (operating system, libraries, compilers, databases, application containers, and so forth) is referred to as a "virtual appliance". Packaging application environments in the shape of virtual appliances eases software customization, configuration, and patching and improves portability. Most commonly, an appliance is shaped as a VM disk image associated with hardware requirements, and

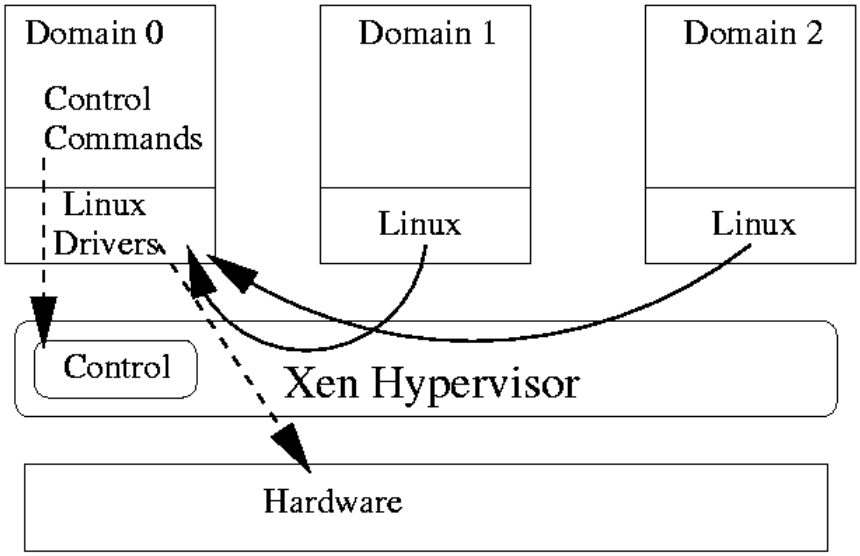

Figure 2.7: *XEN hypervisor architecture*

it can be readily deployed in a hypervisor.

On-line marketplaces have been set up to allow the exchange of ready-made appliances containing popular operating systems and useful software combinations, both commercial and open-source. Most notably, the VMware virtual appliance marketplace allows users to deploy appliances on VMware hypervisors or on partners public clouds [32], and Amazon allows developers to share specialized Amazon Machine Images (AMI) and monetize their usage on Amazon EC2 [33].

In a multitude of hypervisors, where each one supports a different VM image format and the formats are incompatible with one another, a great deal of interoperability issues arises. For instance, Amazon has its Amazon Machine Image (AMI) format, made popular on the Amazon EC2 public cloud. Other formats are used by Citrix XenServer, several Linux distributions that ship with KVM, Microsoft Hyper-V, and VMware ESX(i).

In order to facilitate packing and distribution of software to be run on VMs several vendors, including VMware, IBM, Citrix, Cisco, Microsoft, Dell, and HP have devised the Open Virtualization Format (OVF). It aims at being "open, secure, portable, efficient and extensible" [34]. An OVF package consists of a file, or set of files, describing the VM hardware characteristics (e.g., memory, network cards, and disks), operating system details, startup, and shutdown actions, the virtual disks themselves, and other metadata containing product and licensing information. OVF also supports complex packages composed of multiple VMs [34].

OVF's extensibility has encouraged additions relevant to management of data centers and clouds. Mathews et al. [35] have devised virtual machine contracts (VMC) as an extension to OVF. A VMC aids in communicating and managing the complex expectations that VMs have of their runtime environment and vice versa. A simple example of a VMC is when a cloud consumer wants to specify minimum and maximum amounts of a resource that a VM needs to function; similarly the cloud provider could express resource limits as a way to bound resource consumption and costs.

2.1.7 Autonomic Computing

The increasing complexity of computing systems has motivated research on autonomic computing, which seeks to improve systems by decreasing human involvement in their operation. In other words, systems should manage

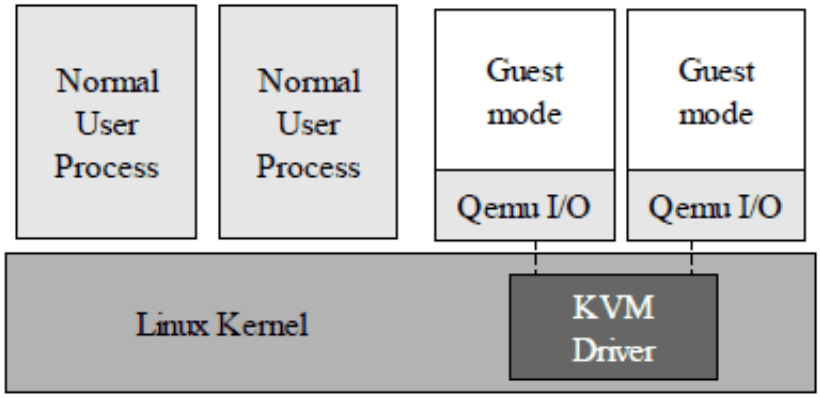

Figure 2.8: *KVM hypervisor architecture*

themselves, with high-level guidance from humans [36].

Autonomic, or self-managing, systems rely on monitoring probes and sensors, on an adaptation engine (autonomic manager) for computing optimizations based on monitoring data, and on effectors to carry out changes on the system. IBM's Autonomic Computing Initiative has contributed to define the four properties of autonomic systems: self-configuration, self-optimization, self-healing, and self-protection. IBM has also suggested a reference model for autonomic control loops of autonomic managers, called MAPE-K (Monitor Analyze Plan Execute-Knowledge) [36], [37].

The large datacenters of Cloud Computing providers must be managed in an efficient way. In this sense, the concepts of autonomic computing inspire software technologies for datacenter automation, which may perform tasks such as: management of service levels of running applications, management of datacenter capacity, proactive disaster recovery and automation of VM provisioning [45].

2.2 Cloud Computing

Cloud Computing to put it simply, means Internet Computing. It is a model for enabling 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.

This cloud model promotes availability and is composed of five essential characteristics (On-demand self-service, Broad network access, Resource pooling, Rapid elasticity, Measured Service); three service models (Cloud Software as a Service (SaaS), Cloud Platform as a Service (PaaS), Cloud Infrastructure as a Service (IaaS)); and, four deployment models (Private cloud, Community cloud, Public cloud, Hybrid cloud). Key enabling technologies include: fast wide-area networks, powerful, inexpensive server computers and high-performance virtualization for commodity hardware [43].

Cloud Computing, as defined by the National Institute of Standards (NIST) [43], is a model for enabling 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. This model promotes availability and is composed of *five essential characteristics* (On-demand self-service, Broad network, access, Resource pooling, Rapid elasticity, Measured Service), *three service models* (Software as a Service - SaaS, Platform as a Service - PaaS, Infrastructure as

a Service - IaaS) and *four deployment models* (Private Cloud, Community Cloud, Public Cloud and Hybrid Cloud).

The Cloud Computing model offers the promise of massive cost savings combined with increased IT agility, meaning how fast the existing or newer business models can adapt to fulfill end-user needs. It is considered critical that government and industry begin adoption of this technology in response to difficult economic constraints. However, Cloud Computing technology challenges many traditional approaches to datacenter and enterprise application design and management. Cloud computing is currently being used; however, security, interoperability, and portability are cited as major barriers to broader adoption. The long term goal is to provide thought leadership and guidance around the Cloud Computing paradigm to catalyze its use within industry and government.

As Ambrust et al. talk in their paper [15], information security is the main issue in case of Cloud Computing: "current cloud offerings are essentially public, exposing the system to more attacks". For this reason there are potentially additional challenges to make Cloud Computing environments as secure as in-house IT systems. At the same time, existing, well-understood technologies can be leveraged, such as data encryption, VLANs and firewalls.

The Internet is commonly visualized as clouds; hence the term "cloud computing" for computation done through the Internet. With Cloud Computing users, for example, can access database resources via the Internet from anywhere, for as long as they need, without worrying about any maintenance or management of actual resources. Besides, databases in cloud are very dynamic and scalable. Cloud Computing is unlike Grid Computing, Utility Computing or Autonomic Computing. In fact, it is a very independent platform in terms of computing. The best example of Cloud Computing is Google Apps where any application can be accessed using a browser and it can be deployed on thousands of computers through the Internet.

Security and privacy affect the entire Cloud Computing stack, since there is a massive use of third-party services and infrastructures that are used to host important data or to perform critical operations. Legal and regulatory issues also need attention. When data is moved and used into the cloud, providers may choose to locate them anywhere on the planet in their own datacenters and the physical location of datacenters determines the set of laws that can be applied to the data. For example, specific cryptography techniques could not be used because they are not allowed in some countries.

In order to ensure that data is secure, that it cannot be accessed by unauthorized users or simply lost and that data privacy is maintained, cloud providers touch the following areas:

- **Data protection and reliability.** Data from one customer must be properly separated from that of another; it must be stored and have the possibility to be moved from one location to another. Cloud providers have systems in place to prevent data leaks or access by third parties and proper separation of duties should ensure that auditing or monitoring cannot be defeated, even by privileged users at the cloud provider.
- **Identity management.** Almost every company will have an own identity management system in order to achieve access control to its information and computing resources. Current cloud providers offer at least two ways for this: 1) They can use the customer's identity management system into their own infrastructure (just like Single Sign On technology) and 2) Provide an identity management solution of their own
- **Physical and personnel security.** Cloud providers ensure that the physical machines which host the virtual machine infrastructure are secure and that access to these machines and client data is restricted.
- **Availability.** Cloud providers must offer clients access to their data and applications.
- **Application security.** Cloud providers must make sure that the offered services are secure. This means all the particular applications must be thoroughly tested and appropriate test beds exists for every level of testing, especially in production.
- **Privacy.** Cloud providers must make sure that the customer critical data, such as credit card numbers, are stored in a safe way and only the legal owner can access it. This also means that the digital identities

must be kept safe.

- **Legal issues.** This is an important matter in current cloud providers because they must take in account the laws that exists in the country their service run.

In the next chapters, we are going to talk in detail about the capabilities of the application we developed. The main goal of this thesis is to present the architecture of a unified and self-contained platform and framework on top of which end-users can develop custom application in a secure and reliable mode. The users will benefit from the full power of the Cloud Computing framework in order to reach their desired performance and security degree.

Chapter 3

Context and related work

3.1 General context

Cloud Computing is cost-effective and the cost is greatly reduced as initial expense and recurring expenses are much lower than traditional computing. Maintenance cost is reduced as a third party maintains everything from running the cloud to storing data. The cloud is characterized by features such as platform, location and device independence, which make it easily adoptable for all sizes of businesses, in particular small and mid-sized. However, owing to redundancy of computer system networks and storage system cloud may not be reliable for data, but it scores well as far as security is concerned. In Cloud Computing, security is tremendously improved because of a superior technology security system, which is now easily available and affordable. Yet another important characteristic of cloud is scalability, which is achieved through server virtualization.

The main form of abstracting the hardware resources, and also the main method of providing the scheduler with information is the lease. The concept of leases is also used in other systems available. Basically, a lease is like a renting contract existing between the user that requests certain resources and the system that offers them. This contract specifies the duration of the renting and is based on the fact that the user will be responsible in that for the resources allocated.

The information that is going to be retained in these leases varies from system to system, but mostly they contain details regarding the processor, the memory that is going to be allocated. More exactly, our scheduler will accept leases that contain the following information, which will be joined together under a lease id: processor architecture, processor vendor, processor speed, processor number of cores, memory size, storage capacity, network bandwidth, network protocol, lease start time, lease end time, lease duration.

In the following sections from our thesis we are going to present the resource management and the security context touched by ReC^{2}S. These two form the main concepts around which our system is built and they represent the most important need that a Cloud Computing provider must fulfill to its end-users.

3.2 Resource management

While Cloud Computing provides many new and modern features, it still has some deficiencies such as the relatively high operating cost for both public and private Clouds. The emergent area of "Green Computing" is becoming more and more important in our days when we have limited energy resources and an increased demand for computational power.

In this context, the key issue with productive use of Cloud Computing technologies is given by the effective management of the cloud resources. For broad adoption, Cloud resources must be manageable with the same or similar techniques currently used by administrators of datacenters. In the future, this context will continue

to modify and update at the point in which we can see how "inner" and "outer" Clouds can be integrated. The outer Clouds are those currently offered by commercial providers like Amazon. An inner Cloud would be a private datacenter using Cloud technologies to manage its resources. Seamless integration of these two domains brings unprecedented flexibility to system administrators and to the users of the systems.

In order to cope with these needs ReC^{2}S has been designed and implemented from the ground in the direction of different Cloud Computing providers integration. The first steps towards a large scale distributed Cloud platform was made with the implementation of a distributed spam scanning infrastructure that had the capability of auto-scaling the number of workers according to the system load [2].

Our system is fully capable of interaction with the major players in this field, without user intervention - only an interface to the outer Cloud is needed for a proper functionality.

3.3 Security

Cloud Computing security, sometimes referred to simply as "Cloud security', is an evolving sub-domain of computer security, network security, and, more broadly, information security. It refers to a broad set of policies, technologies, and controls deployed to protect data, applications, and the associated infrastructure of Cloud Computing.

There are a number of security issues/concerns associated with Cloud Computing but these issues fall into two broad categories: security issues faced by cloud providers and security issues faced by their customers. In most cases, the provider must ensure that their infrastructure is secure and that their clients data and applications are protected while the customer must ensure that the provider has taken the proper security measures to protect their information.

As presented in our previous research [4] and [3], in order to ensure data security, meaning that it cannot be accessed by unauthorized users or simply lost, Cloud providers interact with the following areas: data protection and reliability, identity management, physical and personnel security, availability, application security, privacy and legal issues.

ReC^{2}S is also designed with the security need in mind. During development we tried to consider the current evolution of Cloud Computing and Cloud providers to new directions that have emerged over time. The points mentioned bellow are currently in active development and will be thoroughly investigated in our future ReC^{2}S development plan because they need the involving of a team with skills in multiple disciplines like Advanced Mathematics and Number Theory. These point are:

- **Information security.** In order to allow companies to take control of the data being stored in a cloud environment, we proposed a change of perspective: from nowadays data protection, which is external, to inside data protection. Used or stored data must be encrypted, no matter the environment. This can be achieved by creating intelligent algorithms that make use of advances in Artificial Intelligence research.
- **Trust management in remote servers.** Companies avoid switching to cloud services because they must be entirely sure of what will happen to their once stored remotely data. Data audit and specialized third-party security audit companies must be involved to make sure data is not used in an abusive way.
- **Information privacy.** In order to have all the data safe and processed in a reliable way, a different approach must be taken. For example if documents are stored in clear-text mode they can be searched using simple tools just by specifying a certain keyword. Recently, studies regarding schemes that allow computation over the encrypted text were made. In our example, we gave the search engine an encrypted keyword and the latest will answer with documents that match the query, without looking at the clear-text. Newer cryptographic schemes such as (fully and partial) homomorphic encryption [11] and private information retrieval [16] make all computations on encrypted data (without decrypting).

3.4 Related work

Many projects tackle the problem of dynamically overlaying virtual resources on top of physical resources by using virtualization technologies, and do so with different resource models. These models generally consider overhead as part of the virtual resource allocated to the user, or do not manage or attempt to reduce it. A common assumption in related projects is that all necessary images are already deployed on the worker nodes. Our requirements for dynamic deployment of Advanced Reservation (AR) and As-Soon-As-Possible (ASAP) workspaces make it impossible to make this assumption.

Amazon Elastic Compute Cloud (EC2) [46] is a central part of Amazon.com's cloud computing platform, Amazon Web Services (AWS) [33]. EC2 allows users to rent virtual computers on which to run their own computer applications. EC2 allows scalable deployment of applications by providing a web service through which a user can boot an Amazon Machine Image to create a virtual machine, which Amazon calls an "instance", containing any software desired. An user can create, launch, and terminate server instances as needed, paying by the hour for active servers, hence the term "elastic". EC2 provides users with control over the geographical location of instances which allows for latency optimization and high levels of redundancy. For example, to minimize downtime, a user can set up server instances in multiple zones which are insulated from each other for most causes of failure such that one backs up the other.

The XGE [38] project extends SGE so it will use different VMs for serial batch requests and for parallel job requests. The motivation for their work is to improve utilization of a university cluster shared by two user communities with different requirements. By using the suspend/resume capabilities of Xen virtual machines when combining serial and parallel jobs, the XGE project has achieved improved cluster utilization when compared against using backfilling and physical hardware. However, the XGE project assumes two fixed VM images pre-deployed on all cluster nodes.

The VIOLIN and VioCluster [39] projects allow users to overlay a virtual cluster over more than one physical cluster, leveraging VM live migration to perform load balancing between the different clusters. The VioCluster model assumes that VM images are already deployed on potential hosts, and only a "binary diff" file, implemented as a small Copy-On-Write (COW) file, expressing the particular configuration of each instance, is transferred at deploy-time. This approach is less flexible than using image metadata, as COWs can be invalidated by changes in the VM images. Furthermore, our work and system focuses on use cases where multiple image templates might be used in a physical cluster, which makes it impractical to pre-stage all the templates on all the nodes.

The Maestro-VC[40] system also explores the benefits of providing a scheduler with application-specific information that can optimize its decisions and, in fact, also leverages caches to reduce image transfers. However, Maestro-VC focuses on clusters with long lifetimes, and their model does not schedule image transfer overhead in a deadline-sensitive manner, and just assumes that any image staging overhead will be acceptable given the duration of the virtual cluster. Our work includes short-lived workspaces that must perform efficiently under our model.

The Shirako[41] system developed within the Cluster-On-Demand project uses VMs to partition a physical cluster into several virtual clusters. Their interfaces focus on granting leases on resources to users, which can be redeemed at some point in the future. However, their overhead management model absorbs it into resources used for VM deployment and management. As we have shown, this model is not sufficient for AR-style cases.

The In-VIGO[42] project proposes adding three layers of virtualization over grid resources to enable the creation of virtual grids. Our work, which relates to their first layer (creating virtual resources over physical resources), is concerned with finer-grained allocations and enforcements than in the In-VIGO project. Although some exploration of cache-based deployment has also been done with VMPlant, this project focuses on batch as opposed to deadline-sensitive cases.

In [12] Tayal talks about the perspective of Cloud Computing scheduling techniques. He states that the scheduling algorithms in distributed systems usually have goals of spreading the load on processors and maximizing their utilization while minimizing the total task execution time. Tayal considers task scheduling one of the most famous combinatorial optimization problems, which plays a key role in order to improve flexibility

and reliability systems because the purpose is to schedule tasks to the adaptable resources in accordance with adaptable time, which involves finding out a proper sequence in which tasks can be executed under transaction logic constraints.

In [13] Buyya et al talk about the use of High Performance Computing and Cloud Computing in IT applications. This technology has the ability to gain rapid and scalable access to high-end computing capabilities. They state that Cloud Computing promises to deliver such a computing infrastructure using datacenters so that HPC users can access applications and data from a cloud anywhere in the world on demand and pay based on what they use. However, they raise an alarm on the growing demand for the electricity needed to power these kind of systems and datacenters, which has become a critical issue. High energy consumption not only translates to high energy costs, which will reduce the profit margin of cloud providers, but also high carbon emissions which is not environmentally sustainable. Hence, energy-efficient solutions are required that can address the high increase in the energy consumption from the perspective of not only cloud provider but also from the environment. To address this issue they propose a scheduling algorithm that exploits heterogeneity across multiple datacenters for a Cloud provider.

Also, Brandic et al [14] talk about what Cloud Computing means for end users and they provide the architecture for creating Clouds with user-oriented resource allocation by leveraging technologies such as Virtual Machines (VMs). They also present user-based resource management strategies that encompass both customer-driven service management and computational risk management to sustain Service Level Agreement (SLA) oriented resource allocation. Furthermore, they reveal their early thoughts on interconnecting Clouds for dynamically creating global Cloud exchanges and markets and after that some representative Cloud platforms, especially those developed in industries, along with our current work towards realizing user-oriented resource allocation of Clouds as realized in Aneka enterprise Cloud technology. A great care has been taken in order to highlight the difference between High Performance Computing (HPC) workload and Internet-based services workload. The main point in their work in the meta-negotiation infrastructure needed to establish global Cloud exchanges and markets, and illustrate a case study of harnessing "Storage Clouds" for high performance content delivery.

Chapter 4

System architecture

In this chapter we will present in detail the entire application with all its layers. We will start with the software requirements specification, both functional and non-functional. Next we will present each module that forms the application.

4.1 Software requirements specification

4.1.1 Functional requirements

System diagram

The system presented in our thesis has a modular architecture. All modules are described in detail. It is easy to see that the whole ecosystem is actually pluggable and it can be extended with other modules or plugins.

In Figure 4.1 we can see a top view architecture from a functional perspective. The entire system is composed from 6 layers. We present them briefly to offer a general idea of the components and how they fit in.

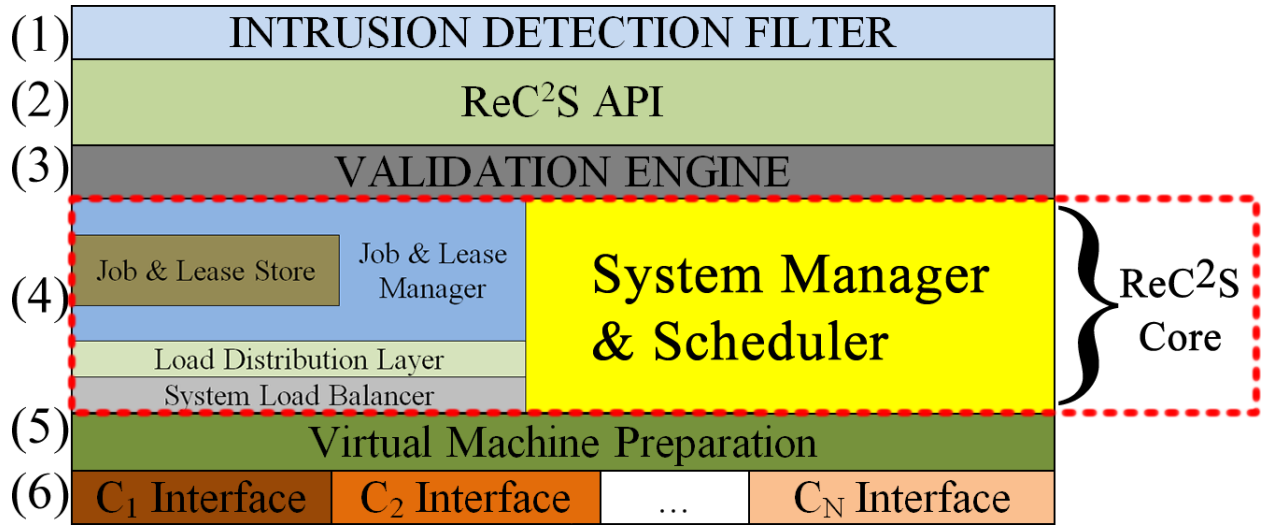

Figure 4.1: *ReC²S functional top view system architecture*

The **first layer** is responsible for filtering the requests that come from outside the Cloud. It basically checks the legality of a certain request to the Cloud system. The **second layer** is the API layer. The API represents a set of primitives that are offered to the user and permits interaction with the Cloud systems. It is presented as a web front-end and a static API. For example an authenticated user can request to start/stop/restart a certain virtual machine, can push a job to the Cloud system or a specific virtual machine, can retrieve answers from the job that he pushed, etc. The **third layer** is responsible for checking that the actions that the user specified or requested are actually eligible for executing. The main goal is security - anomaly request detection. The **fourth layer** and the most important one is the core of the system. It is responsible with interaction with the Cloud systems that he manages. It does a load balancing of the requests it receives, both in the same autonomous system or inter autonomous systems and it runs a lease based scheduler. The **fifth layer** is responsible with the preparation of the virtual machines that are going to be loaded. This includes finding the appropriate virtual image in the content repository, and, if necessary, installing different software stacks inside. The **sixth layer** is responsible for communication directly to a specific Cloud system interface.

Intrusion detection filter

This layer is the entry point in our system and it is responsible with its security. It is designed in such way that it can detect attacks that originate from outside. It detects malicious actions like flooding and DoS/DDoS attacks.

ReC^{2}S API

This layer offers a way to the user to interact with the system. In our current evolution state we provide means to add new leases. The requests are registered and sent to validation, to the proper layer. In order to be more interactive, we provide a graphical user interface, in the form of a webpage.

This layer is split in two separate layers. The first one represents a REST-full API implementation and the second one the proper implementation. In our implementation, the API wrapper must permit the use of the following actions:

- setting the lease details. In this section, the user must provide a series of information about the lease that he's creating, like:
  - processor architecture. The user can choose between a 32 or 64 bit architecture. This is important because in this way he can take advantage of different optimization and speedups available to certain processors;
  - processor vendor. The user can choose between Intel or AMD processor type. Also this is important for certain applications that are optimized for a specific vendor;
  - processor speed. This is the actually speed of the CPU;
  - number of processor cores. This is the number of processor/cores that the virtual machine(s) from this lease will have available;
  - memory size. This is the amount of RAM available to the virtual machine(s);
  - storage capacity. This is the amount of disc space required for the virtual machine(s) to run;
  - network bandwidth. This is the speed of the virtual network card that the virtual machine(s) will have;
  - lease start time. This is important when adding a lease because it impacts the way in which the virtual machine(s) are added and started. This time is used in the different scheduling policies inside the core. The user can choose between a determinate or infinite time for the virtual machine(s). An infinite time is the same as having a persistent lease. The determinate time can be a certain

timestamp in the future or even a certain event (ex: the user wants to start a lease when data is ready to be processed);

  - lease time end. This is important when adding a lease because it impacts the way in which the virtual machine(s) are ran, stopped and preempted. This time is used in the different scheduling policies inside the core. Also the user can choose between a determinate or infinite (persistent) lease;
  - lease duration. This, in conjuncture with the preemptible part can be used in help of the users that are not running very important tasks and they can permit the virtual machine to be paused and resumed for running more important leases. Also, this is a good way of reducing cost and achieving the desired elasticity. The user can choose between a timed or infinite (persistent) lease;
  - preemtible. This offers the user the chance to make his lease prone to pausing and resuming if the scheduler decides to do that.

- setting the virtual machine(s) details. In this section, the user can customize settings for the virtual machine(s) that is (are) going to be run with this lease, like:
  - minimum instances. This is the minimum of virtual machines that the user wants at a certain period, and their number mustn't never get bellow this value;
  - maximum instances. This is the maximum number of the virtual machines that the user wants in heavy load periods. The scheduler must be careful when using this feature to start the machines when a heavy load is detected and stop the extra virtual machines when not needed. This is good for the user because it helps in achieving desired elasticity;
  - the type of the template. This is the operating system that every virtual machine involved in a lease must run. It's implemented as a lease in order to maintain an uniform view of the virtual machines for the scheduler. Currently, the only base template in use is Ubuntu Server 10.04. Later, these templates will be enriched with other Linux operating systems, or even let the user upload their own virtual machine template;
  - network configuration. This is the IP or IP pool of the virtual machine(s) that is going to be used by the user to connect to them.
- setting the packages that the users want to deploy on each virtual machine. This lets the user to auto-install some software on each of the running virtual machine(s). This can include a Java/Python/MPI/etc development framework, a database, etc;
- saving the request for future uses. This is useful for the user because he can save the settings made to the lease, and also the lease itself for future uses.

After adding a lease, the user can access its details and information like:

- lease administration. This permits the user to stop, pause or restart the lease at his desire;
- alerts. This permits the user to have him attention on events like: before starting the lease with T minutes, on lease start, on lease stop, when the scheduler decided when to start the lease.

API action validation

This layer receives requests to add new leases in system. Every new request is checked for consistency and validated. If the request is legit and the user making the request is authenticated, the new lease is transformed in a job for our system and it's properly inserted in the job queue.

Core

This layer is the most important from the entire system. The core is composed from 4 sub-layers, as seen in Figure 4.2. We present each of them briefly in order to make a good idea of the core layer.

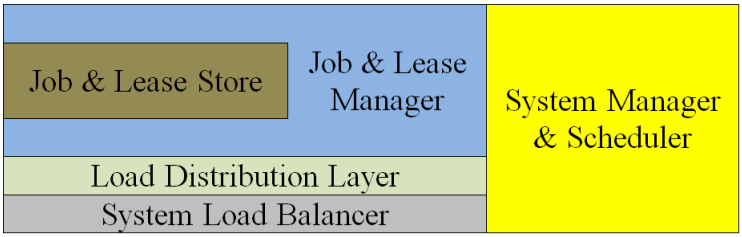

Figure 4.2: *ReC²S system core layer*

The **job and lease manager** mainly consists of a form of data replication and partition. It provides real-time data aware routing, elastic data access and caching. It stores all lease requests from the outside. These requests are stored as objects in the queue. This sub-layer can be implemented in a lot of ways, but at our current stage is implemented as a message queue. When choosing a particular queuing system we analyzed features like: the speed of the sub-system, support for different programming languages (Java, C, C++, etc), standard compliant, data security, data replication, support for different frameworks (Spring, etc), support for different communication protocols (TCP, SSL, UDP, multicast, etc).

The **job and lease store** is basically a sub-layer of the lease manager that is responsible for storing in a reliable way the things that the Cloud system manager and scheduler will process. It connects the entire core part to the job and lease store. This is implemented in a generic form and has to provide access methods like getting an element from the store, putting an element to the store. Because the underneath store can be implemented in different ways and using different architectures, this sub-layer is implemented in a plug-and-play manner, using a pluggable system, so that every access to the store should be made using a specific plugin.

Every plugin must register to the manager. We recommend this approach in order to avoid having to re-deploy and restart the manager every time a store is added, or the store API is changing. If this is not a problem for the environment, the whole sub-layer can be stopped, modified and then started again.

As we have said above, the manager must provide an interface to the outside world. In our current implementation we used the following ones:

- addLease(L). This call will add to the specific store the lease L. This call must return to the caller a proper value that will reflect if the lease is added to the store and it's ready to be processed, or if the adding the lease to the store has failed. This return value is necessary in order to the caller to take proper actions: either inform the user that everything is good and to wait for the access to the virtual machine(s) that he requested or retry adding the lease to the store;
- getNextLease(). This call will return to the caller the next lease that is going to be processed, if it exists. If the store doesn't have any leases the caller must get an empty store alert in order to retry to call this until there are available leases to process.

In order for this manager to work properly it provides real-time data aware routing. The only abstraction being the underneath store, starting from the upper layers, the system has proper knowledge of the leases that are requested and offered to the caller. This is necessary for providing elastic data access, meaning that the sub-layer itself must be capable of auto-balancing in case of high network traffic. Also this shall happen if the stores that it manages are heavily loaded and/or the requests are coming in too fast for one manager to handle.

Caching must be another important feature that this layer has to handle. The leases requested from the underneath store are kept in cache for a certain period of time. We have chosen this approach to fulfill the needs for the persistent leases. This case is rarely found in practice because only few of the users want the lease to be persistent due to the high cost of the resources.

The **load distribution layer** is a sub-layer responsible with horizontal scaling the requests received from the scheduler. It runs an application framework in order to decouple the code from the existing underneath runtime. This is done automatically and in the process of this analysis, the number of workstations is taken in

account. If their number changes over the execution of our system, the entire algorithm is re-run to reflect the current situation.

The **system load balancer** is responsible for vertically scaling the requests received from the scheduler. This is done also automatically. There is a connection between this layer and the load distribution layer. To be clearer, we give a simple example of how the two of them work together. Let's assume that we run our system over an infrastructure consisting in four servers, all the same. The system load balancer detects that the hardware capabilities are the same and will report it to the load distribution layer. Then, the load distribution layer sees that it has four servers available, and after receiving updates from the system load balancer, it will split the entire work load in four parts and will submit them to each server. Now, let's assume that a server will be replaced by one, twice as powerful. The load distribution layer will still see four servers, but it will get a notification from the system load balancer that one of the servers is two times more powerful than the remaining three. Therefore, it will split the work load into five parts and it will submit two parts to the newer and powerful server and to the rest of the others, one part for each.

The **system manager and scheduler** is the most important sub-layer. Its main purpose is efficiently scheduling the jobs and leases between the virtual machines. It also discovers new instances of new services and virtual machines managers, load balancers, load distributions. It has a pluggable interface in order to be scalable. We detail this layer in depth in Chapter 5.

Virtual Machine Preparation

This layer is intended as a layer between the core and the actual Cloud specific API. The most important tasks that it does are:

- get the decision made by the scheduler;
- analyze it and create a deployment plan. The deployment plan must contain the virtual machine image that is going to be loaded, the software stack that is going to be installed and the appropriate installation script, if such option is chosen by the user;
- preload using the Cloud API the image to the chosen location;
- run the installation script. When choosing a certain framework to run our script we analyzed features like: system independent implementation, the ability to run all kind of tasks (creating a folder, deleting a folder, calling certain specific commands). All these requirements were filled by the open-source project Apache Ant

Cloud Specific Interface

This layer is intended as an abstraction between our system and the Cloud platforms existing on the market. In order to be more scalable and also maintain a high degree of abstraction, an interface is provided and every implementation of the specific API must implement this. This layer acts as a plugin manager, with the proper API being inserted at run-time in a plug-and-play way.

ReC^{2}S is a sum of parts that are designed to work together. The application is composed from 7 modules, as it can be seen in Figure 4.3. The "GUI" module is put outside because the goal of our thesis is the presentation of our framework in details. Nevertheless we will briefly present it in the "Software design specification".

4.2 Software design specification

As we mentioned in the "System diagram' section, our application is composed from 7 modules that work together, each having it's own role. We will present each of the module and there will be detailed a series of

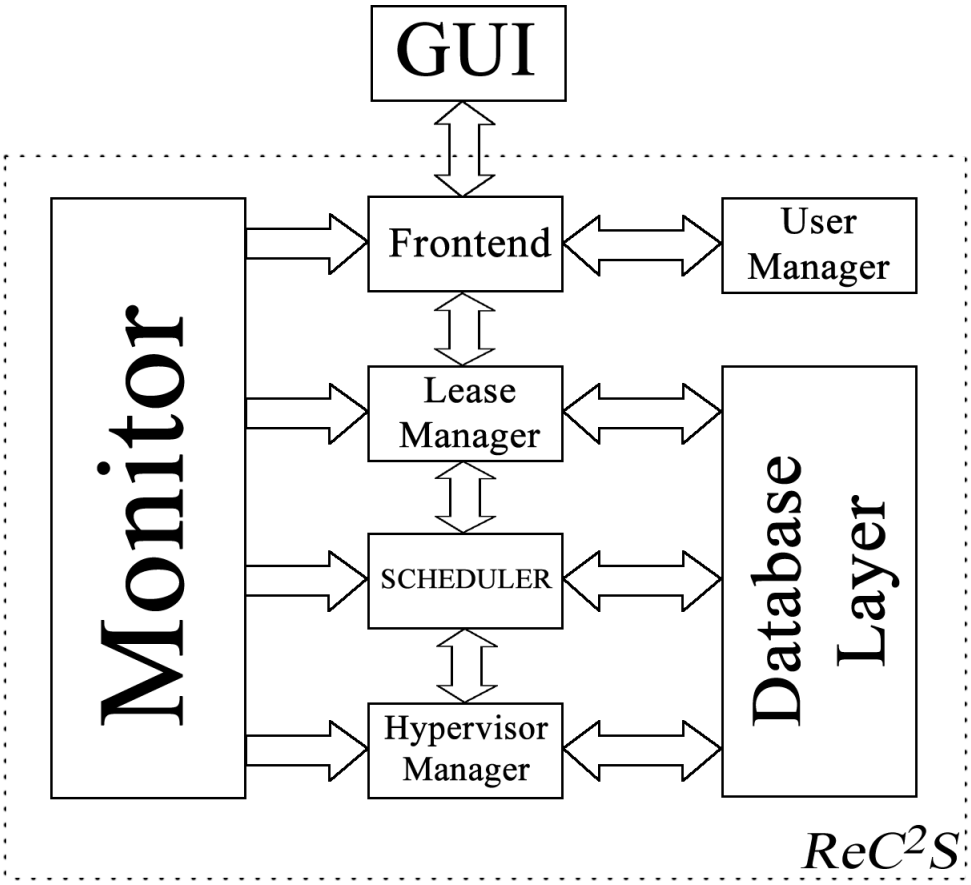

Figure 4.3: *ReC²S System Architecture*

essential sections for each module individually, in special the main data structures used, the design templates and the the module description. These modules are:

1. Frontend
2. User Manager
3. Lease Manager
4. Scheduler
5. Hypervisor Manager
6. Monitor
7. Database Layer

The actual software was designed with a combined architectural template in mind. It brings benefits from both the **"Client-Server"** model and also the **"Distributed-Computing"** model. The *Client-Server* model is a computing model that acts as a distributed application which partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients, that communicate over a computer network, each on separate or same hardware. A *Distributed System* consists of multiple autonomous computers that communicate also through a computer network. Therefore we can say that the *Distributed-Computing* model has been used because the system is composed from a series of software components that run on different machines and communicate through the network in order to supply the answer to the user in a short time.

The actual implementation of the information presented in the "Functional requirements" section is made using a multi-tier architecture, more exactly a three-tier architecture. In our case the presentation, the application processing and the data management are logically separate processes. This kind of application architecture provides a model to create a flexible and reusable application. By breaking up an system into tiers, we only have to modify or add a specific layer, rather than have to rewrite the entire application over again.

More exactly, a three-tier architecture has the following three tiers:

- **Presentation tier.** As we can see in Figure 4.4, this is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing, and shopping cart contents. It communicates with the other tiers by outputting results to the client tier and all other tiers in the network.

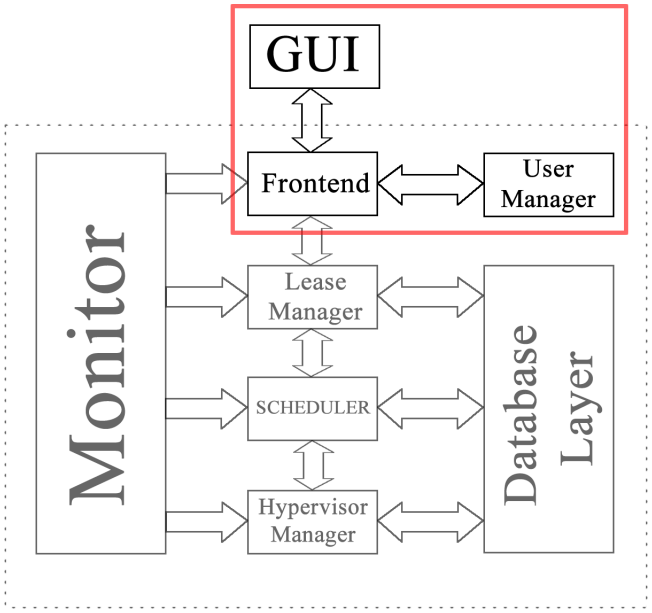

Figure 4.4: *ReC²S Presentation tier*

- **Application tier.** This is also called *business logic tier*, *data access tier* or *middle tier*. It is pulled out from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing. See Figure 4.5 for a graphical description of it.
- **Data tier.** As we can see in Figure 4.6, this tier is responsible for database connectivity. Here information is stored and retrieved. This tier keeps data neutral and independent from application servers or business logic. Giving data its own tier also improves scalability and performance.

On the following sections we will present each of the ReC²S modules and how they map to the top view architecture. We will start with the external module called "GUI" and we will continue in order with the modules that compose each tier.

4.2.1 "GUI" module

The **"GUI"** module is responsible for user interaction. It is presented as a web interface. In Figure 4.7 we presented the most important screen from it, the one from which the user can submit a new lease in the system. We can see that a user can choose a number of virtual machines - a minimum and a maximum instance count. Also he can select each virtual machine template properties and software that is going to be installed, when to start and stop the lease. At this step he can also choose if he wants a preemtible lease or a regular lease. Beside

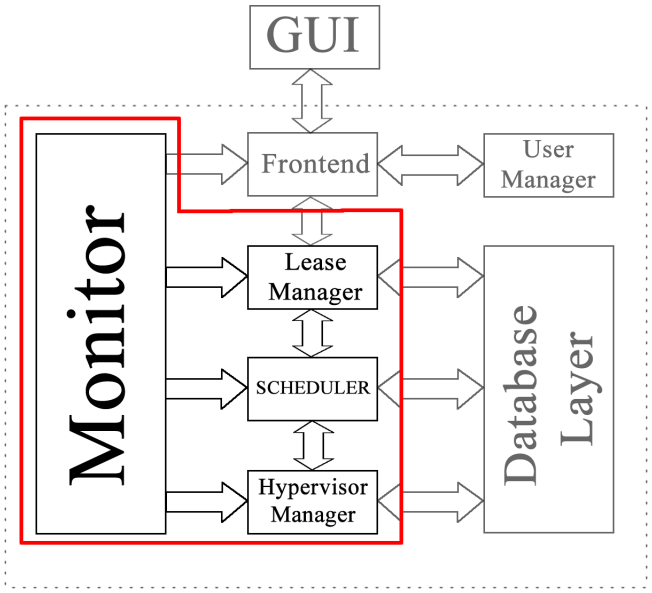

Figure 4.5: *ReC²S Application tier*

this functionality, an user can register itself as a ReC²S user, see the status of each lease submitted by him or set specific alerts to each lease.

4.2.2 "Frontend" module

This module maps over the "Intrusion detection filter" and "ReC²S API" layers and has the functionality described to them. Also it receives authentication requests from the "GUI" module and if the requests are legit it passes them to the "User Manager" module.

4.2.3 "User Manager" module

This module is responsible for storing the details for the system users, like username and password. It has authentication and lease validation purposes and it maps over the "Validation engine" layer.

4.2.4 "Lease Manager" module

This module implements the "Job and Lease Manager" layer. It is responsible for getting the leases from the user, transform them into proper jobs and saving them both on local physical memory and a remote database. We have chosen this approach all over the modules in order to fulfill the availability needs that our system must provide.

More precisely, we created a general framework based on a cluster of processes, that we are modeling for each module needs. We will explain further this decision and how it works.

Today's modern processors are based on multi-core technology. This means we have to find new and better ways to deal with this environments. One approach can be made using threads, but it is hard to use them *correctly* in order to be efficient. The alternative approach is to use process parallelism which uses processes instead of threads. In this case the user will want to launch a cluster of processes to handle an incoming load.

In our implementation we currently have a local master-slave architecture. This means that we have at least one master and at least a slave (worker) that does the job. Our framework recognizes when a process

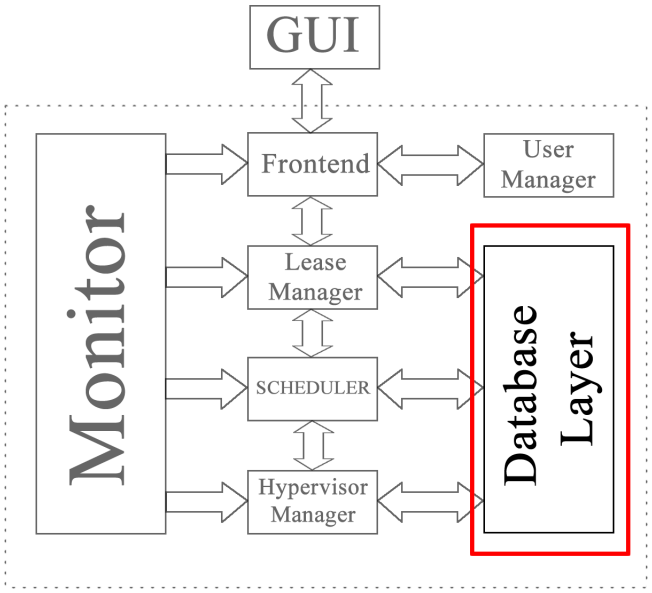

Figure 4.6: *ReC²S Data tier*

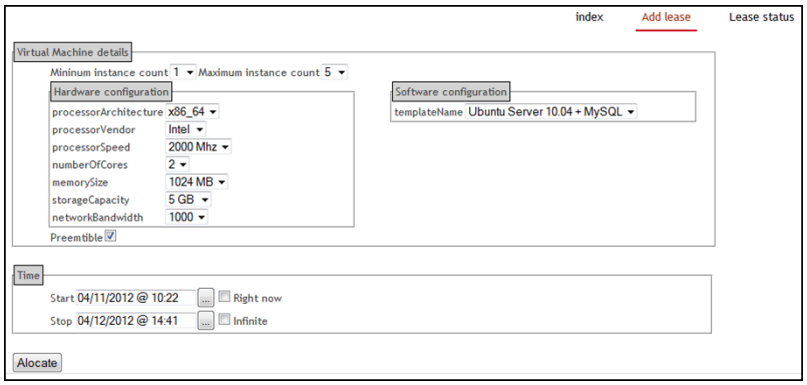

Figure 4.7: *ReC²S User GUI*

is first ran and automatically promotes it to *master* and every children that is forked from it, for each CPU existing on the system or virtual machine, is promoted to *worker*. A great advantage of our framework is that the master can share a single network port, making load balancing implicit. Furthermore, if a worker crashes the master instantly detects it and forks another fresh process to take its place. To avoid the single-point-of-failure problem, our framework automatically starts another shadow process that backs up the master in case of failure. We represented our basic framework functionality in Figure 4.8.

For this specific module we added on top of this framework the possibility to save a lease both on local memory and a remote database. In Figure 4.9 we can observe this more clearly. It is easy to see that even if the database connectivity is broken the module keeps running and will store the leases only in local memory. We also added the possibility that when the module starts up it loads the existing leases from the database into the memory. In this way we keep the entire consistency of the system in case of a crash.

This module is implemented using a plug-an-play architecture, meaning that we can add different storage capabilities. For example, instead of a single database, the system administrator can use a cluster of databases for redundancy. Even more, the driver registration is made automatically and does not create downtime.

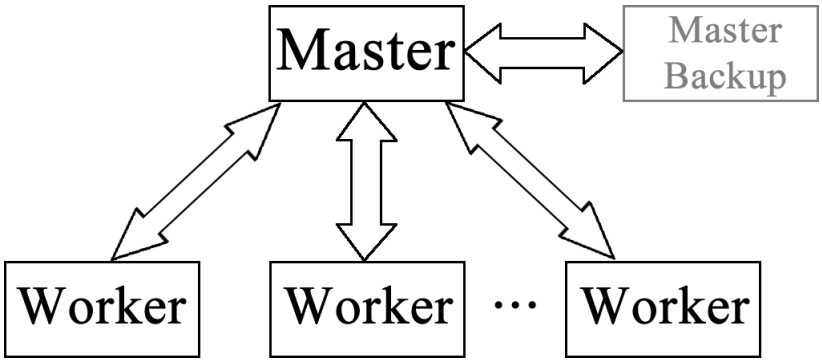

Figure 4.8: *ReC²S process cluster framework*

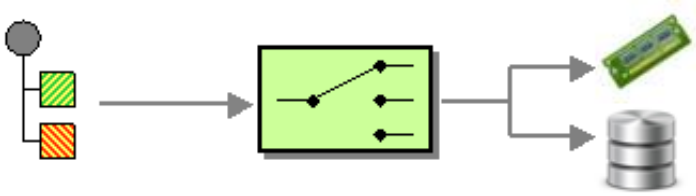

Figure 4.9: *ReC²S lease manager simultaneous store*

4.2.5 "Scheduler" module

This module is the implementation of the "Core" layer and it is presented in detail in Chapter 5

4.2.6 "Hypervisor Manager" module

This module is responsible for the hypervisors existing in the system and it maps over the "Virtual Machine Preparation" and "Cloud Specific Interface" layers, having a double functionality. It is also implemented as plug-and-play and the system administrator can add hypervisors by writing specific drivers. It can manage both individual hypervisors, like VMware ESX, and existing Cloud Computing environments, like OpenNebula or Eucalyptus. It also monitors the state of each of the attached hypervisors and the load of each virtual machine running on each hypervisor.

Furthermore, it receives requests from the "Scheduler" to start a new lease or to check the physical resources needed to run a new lease. In order to manage the virtual machines from the lease it uses a standard interface that permits it, for example, to start/stop/restart a particular virtual machine.

4.2.7 "Monitor" module

This module is responsible for monitoring the entire activity from the system. It knows the leases that are marked as *preemptible* and decides when to preempt them. Also it decides if a running lease needs virtual machine instances increased up to the maximum count specified by the user or decreased to the minimum count.

In order to do this properly he knows what preemptible leases are running at a particular moment in time and also knows the load of the virtual machines running inside the lease. If the load is bigger than a point, it commands the start of a fresh virtual machine, up to the maximum instance count. If the load is smaller that a point, and the system is running more virtual machines than the minimum instance count, it commands

the stop of virtual machines. To monitor the virtual machine load it must communicate with the "Hypervisor Manager".

4.2.8 "Database Layer" module

This module is responsible with the underneath database(s) (DB). Since it is implemented separate from the entire system it can be easily adapted to work with different DB softwares. It is also implemented using the same plug-and-play architecture and the only thing that is needed to do from the part of the system administrator is writing the proper driver to interface with the particular DB vendor.

Chapter 5

ReC^{2}S Scheduler

5.1 General structure

The main form of abstracting the hardware resources, and also the main method of providing the scheduler with information is the lease. The concept of leases is also used in other systems available. Basically, a lease is like a renting contract existing between the user that requests certain resources and the system that offers them. This contract specifies the duration of the renting and is based on the fact that the user will be responsible in that for the resources allocated.

As presented before, the information that is going to be saved in these leases varies, but mostly they contain details such as processor architecture, processor vendor, processor speed, processor number of cores, memory size, storage capacity, network bandwidth, network protocol, lease start time, lease end time, lease duration.

5.2 ReC^{2}S scheduler architecture

5.2.1 Functionality

In order for the scheduler to function properly, a scheduling scheme has been composed. Depending on the leases and the task, they are split in the following scheduling types and the scheduler will function in three ways:

1. Using advanced reservations. This forms the base of the leasing strategies. It reflects fixed resources that are going to be allocated.
2. Using a best-effort strategy. This is formed from another four sub-strategies:
   - **Pre-emptible.**

     The virtual machines can be started, stopped, paused and resumed at a given time. This process can be somehow compared with the preemption of processes made in the operating system, only that in our case the processes are replaced with virtual machines. To maintain a good system consistency an external clock must be used, to prevent messages loosing between the virtual machines when they are stopped.

     To be more exactly, in the following lines we will talk about this strategy and comparing it with a process. Let's assume that we have a virtual machine running, not exchanging data with any other virtual machine. When the scheduler decides to preempt it, for later reloading or to move it to

another network node, it just calls the properly "Pause Virtual Machine" function. Then, the virtual machine is stopped and it's ready for restore.

But if we have a machine that exchanges data with another virtual machine and it depends on it to function properly, when the scheduler decides to preempt it, all this chain of virtual machines must be stopped in the same time to avoid data loss. The same happens when they are restored - the scheduler must restore them in the same time, again to avoid data loss.

- **Non-pre-emptible**

  This type of scheduling is also known as "right now allocation". In order to achieve this, a series of information regarding the virtual machine and the place in which the virtual machine will run is required. The scheduler needs data like the actual size of the files that compose the virtual machine, the transfer speed to the destination network and the time needed for the virtual machine to start. For example, if the user wants a lease of one machine, starting in 2 minute, the scheduler must find the appropriate free server to start the virtual machine, and start to transfer the virtual machine template as soon as possible so that at the end of the 2 minute, a virtual machine will be ready for deployment. Also, if the template is copied in place sooner than the starting point, it is no problem because it can wait for the user to start using it. A problem appears if the starting point of the lease is sooner than the time needed to transfer the virtual machine image from the repository to the destination. This can be resolved by using virtual machines caches. This part will be resolved in the future by a different module of the system.

- **Deadline.**

  This type of scheduling is also known as "you can allocate anytime, but no longer than T", where T is a fixed time. In order to achieve this, the scheduler must know how much time will the lease last so that he can allocate it before this.

- **Negociated.**

  Depending on the moment in time when you will allocate (Now: 100$, after 1h: 50$, after 2h: 20$, etc)

3. Using an urgent lease. This is going to be used when instant resources must be allocated

In order to deploy efficiently through different virtual machines operating systems a process driver must be used. To make this task easier we will use Apache Ant because of its support for multiple architectures and operating systems.

5.2.2 Architecture

The whole system is seen as in Figure 5.1. We will detail each of the modules in the next paragraphs.

I've mentioned the "System input" layer because in order to work, our system must have entries that contain leases to process. The input is presented more thoroughly in another paper [1] and previuos chapter, and mainly, this is a connection layer between the core and the underneath lease storage.

When the input reaches the system it is analyzed and sent to the **Resource Checker** module. This is responsible with checking the availability of resources specified in the lease on the physical systems. It also checks if the system can sustain this requirements on the whole lease duration. This decision is influenced by the current state of the managed systems. It must also keep a safe guard for the required elastic expansion of the virtual machines involved in the allocation of the lease. If this module decides that the lease is safe to be allocated it saves the lease on a special queue, implemented as a priority queue, the **"Timed start queue"** from the above. If it decides that the physical system cannot hold the lease it saved the lease on another special queue, also implemented as a priority queue, the **"Infinite start queue"** from the above. The thing that differences the two queues is that in the moment the lease is inserted in the latest, the lease start time is

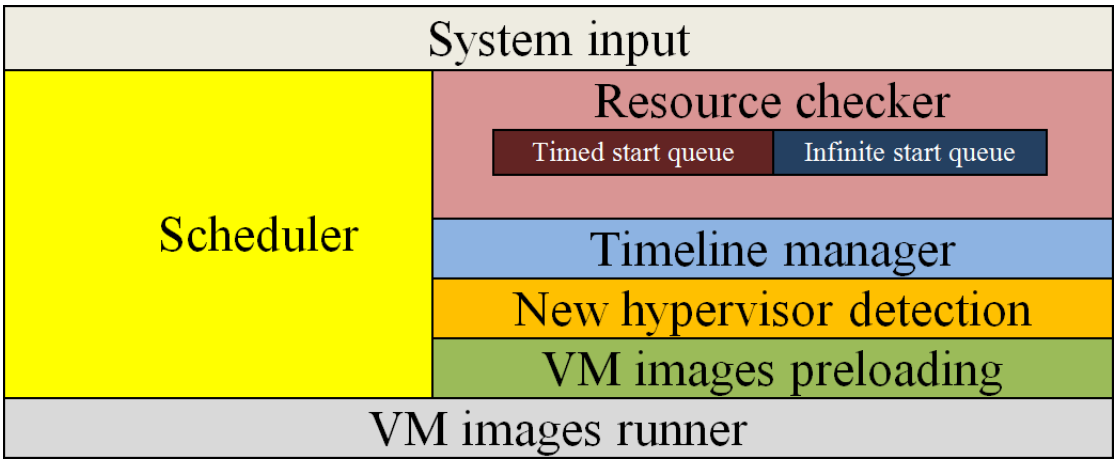

Figure 5.1: *General ReC²Sscheduler architecture.*

modified to infinite, so that it can start as soon as the system has free resources. We will explain later how this thing is achieved.

Another thing that this module uses is the **"Timeline manager"**, which acts as a global clock and it manages the two queues mentioned above. Mainly its functionality is that, at every defined moment in time, in our case at every minute, to check first for existing leases in the timed start queue. If it founds one, it is automatically deployed on a destination hypervisor chosen from the one attached to the virtual machine runner layer. It also checks if resources are available to run the lease from the infinite start queue.

The **"New hypervisor detection"** layer is responsible with detection of new hypervisor that are going to be attached and used when running the leases.

The **"VM image preloading"** layer is responsible with the initialization of the virtual machine image, or cloning an existing one. The initialization creates a new virtual machine from an existing configuration and installs the software stacks chosen by the user. The cloning function just duplicates an existing configuration.

The **"VM image runner"** is responsible with running the previously created and configured virtual image, on the destination hypervisor. The user will be granted access to the virtual machine at this step.

The core is composed from the **"Scheduler"** layer.

5.2.3 Scheduling

Historically, scheduling was and it's going to be a hard and tricky problem in computer science [47]. Scheduling is concerned with the allocation of scarce resources to activities with the objective of optimizing one or more performance measures. Depending on the situation, resources and activities can take on many different forms. Resources may be machines in an assembly plant, CPU, memory, I/O devices, etc. This is usually done to load balance a system effectively or achieve a target quality of service. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously).

The scheduler is concerned mainly with:

- **Throughput** - number of processes that complete their execution per time unit.
- **Latency**, specifically:
  - *Turnaround* - total time between submission of a process and its completion.

  - *Response time* - amount of time it takes from when a request was submitted until the first response is produced.

- **Fairness / Waiting Time** - Equal CPU time to each process (or more generally appropriate times according to each process' priority)

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the above mentioned concerns depending upon the user's needs and objectives.

In real-time environments, such as embedded systems for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable. Scheduled tasks are sent to mobile devices and managed through an administrative back end.

Types of schedulers

Computing systems may feature up to 3 distinct types of scheduler, a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler and a short-term scheduler. The names suggest the relative frequency with which these functions are performed. The scheduler is also a system module that selects the next jobs to be admitted into the system and the next process to run.

1. **Long-term scheduling**

   The long-term, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue (in the Main Memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - i.e.: whether a high or low amount of processes are to be executed concurrently, and how the split between IO intensive and CPU intensive processes is to be handled. In modern operating systems, this is used to make sure that real time processes get enough CPU time to finish their tasks. Without proper real time scheduling, modern GUI interfaces would seem sluggish. The long term queue exists in the Hard Disk or the "Virtual Memory".

   Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers and render farms. In these cases, special purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system.

2. **Medium-term scheduling**

   The medium-term scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.

   In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the medium-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded"

3. **Short-term scheduling**

   The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt, an IO interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions

much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU. In most cases short-term scheduler is written in assembly because it is a critical part of the operating system.

5.2.4 Online scheduling

The form that we are going to study in deeper and also implemented in my system is called **online scheduling**. In our online scheduling form, the scheduler receives jobs during different periods of times. The most interesting part is that it must take decisions without knowing some of the details of the jobs or knowing what will happen in the future. This means that the decisions he is going to make will not be optimal, but with proper algorithms and heuristics this entire process will tend to that.

In the online-time paradigm, the scheduler must decide at each time t which job to run at time t. Problems within the online-time model typically have release dates, and the scheduler is not aware of the existence of a job until its release date. Once a job is released, we assume that the scheduler learns the processing time of a job. For example, a web server serving static documents might reasonably be modeled by the online-time model since the web server can know the size of the requested file. In contrast, in the online-time-nclv model, the scheduler is given no information about the processing time of a job at its release date. For example, the process scheduling component of an operating system is better modeled by the online-time-nclv model than the online-time model since the operating system typically will not know the execution time of a process. This lack of knowledge of the processing time is called ***nonclairvoyance***.

If preemption is not allowed for problems in either the online-time or online-time-nclv model, and jobs can have arbitrary processing time, then there is usually a trivial example that shows that any online scheduler will produce schedules that are far from optimal. This is why server systems, such as operating systems and web servers, generally allow preemption. Thus most research in online scheduling assumes preemption unless all jobs have similar processing times (as could be the case for a name server, for example). In the online setting, there is another possibility, which is meaningless for offline algorithms. Namely, a running job can be stopped and later restarted from the beginning on the same or different machine(s). Thus in order to finish, a job has to be assigned to the same machine(s) for its whole running time without an interruption; in the offline case this is equivalent to nonpreemptive scheduling as the unfinished parts of a job can simply be removed from the schedule. This possibility will be denoted by pmtn-restart in the middle (job) field of the three-field notation.

In the online-list paradigm, the jobs are ordered in a list/sequence. As soon as the job is presented, we know all its characteristics, including the processing time. The job has to be assigned to some machine and time slots (consistent with the restrictions of the given problem) before the next job is seen. The scheduling algorithm cannot change this assignment once it has been made. In the online-list model, in contrast to the online-time and online-time-nclv models, the time between when jobs are assigned is irrelevant or meaningless. The online-list model might be an appropriate model for a load balancer sitting in front of a server farm.

In our research for the most suitable scheduling algorithms we studied a couple of them. We will present them in the following section, together with our observations:

- **randomized algorithms.** These algorithms are based on a Monte-Carlo approach and take their decision based on a random choice
- **semi-online algorithms:**
  - ***Shortest Job First.*** Is a scheduling policy that selects the waiting lease with the smallest execution time to execute next. This is advantageous because it's simple to use and implement and because it maximizes the average amount of time each lease has to wait until its execution is complete. However, it has the potential for what we call "leases starvation", for leases which will require a long time to complete if short leases are continually added. This policy can be effectively used with interactive

processes which generally follow a pattern of alternating between waiting for a lease release and executing it. If the execution of a command is regarded as a separate entity, past behavior can indicate which lease to run next, based on the estimate of its running time. Shortest job first is rarely used outside of specialized environments because it requires accurate estimations of the time of all leases that are waiting to execute. Estimating the running time of queued leases is sometimes done using a technique called aging.

- ***Shortest Remaining Processing Time.*** Is a scheduling method that is a preemptive version of the Shortest Job Next scheduling. In this algorithm, the lease with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progress, leases will always run until they complete or a new process is added that requires a smaller amount of time. This policy is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a lease completes or a new lease is added, and when a new lease is added the algorithm only needs to compare the currently executing lease with the new lease, ignoring all other leases currently waiting to execute.
- ***First In First Out.*** Is basically an abstraction in ways of organizing and manipulation of data relative to time and prioritization. This expression describes the principle of a queue processing technique or servicing conflicting demands by ordering process by first-come, first-served (FCFS) behavior: what comes in first is handled first, what comes in next waits until the first is finished, etc.
- ***High Density First.*** The algorithm Highest Density First always runs the job with the highest density, which is the weight of the job divided by the initial work of the job.
- ***Round Robin.*** This is one of the simplest scheduling algorithms for leases, which assigns time slices to each lease in equal portions and in circular order, handling all leases without priority (also known as cyclic executive). Round-robin scheduling is both simple and easy to implement, and starvation-free. This kind of policy may not be desirable if the sizes of the jobs or tasks are highly variable. A lease that produces large jobs would be favored over other leases. This problem may be solved by time-sharing, i.e. by giving each job a time slot or quantum (its allowance of CPU time), and interrupt the job if it is not completed by then. The job is resumed next time a time slot is assigned to that lease.
- ***Shortest Elapsed Time First.*** This algorithm devotes all the resources to the job that has been processed the least. In the case of ties, this amounts to round robin on the jobs that have been processed the least. While round robin perhaps most intuitively captures the notion of fairness, shortest elapsed time first can be seen as fair in an affirmative action sense of fairness

From all the above scheduling algorithms we have chosen to implement FIFO and SJF. In completion, we also have implemented a custom scheduling algorithm, called ReC2Sched. We will present them bellow.

The first engine is FIFO. This is a simplistic implementation and it processes leases as soon as it arrives. SJF additionally does a lease sort depending on the time of running. In our tests, the times obtained with these two engines were not sufficient to satisfy our need for speed. So we started implementing a custom algorithm, specially created for our purpose. It is different because it does fast lookups into the lease storage unit and it analyses them in advance (we have used in our implementation a 2 minute processing time in advance). Also, our algorithm is a hybrid between the previous two and also is an adaptive algorithm. As a novelty, we are using a 0.5 second computation step. In each step a number of leases is choses to be scheduled.

Chapter 6

Software implementation

The system was implemented in a modular way. For this we used a new paradigm of parallel and distributed programing: an event loop. We will further explain this concept.

Historically, there are two main ways in which we can program in a parallel and distributed way, each having advantages and disadvantages: a) using threads b) using an event loop. A threaded model will spawn a new thread for every request. This means that we get some overhead in terms of computation and memory. An event loop runs in a single thread, which means we don't get the overhead.

The result of this is that we must change our programming model. Because all these different things are happening in the same thread, we cannot block. This means we cannot wait for something to happen because that would block the whole thread. Instead we define a callback that is called once the action is complete. This is usually referred to as non-blocking I/O.

Pseudo example for blocking I/O:

```
row = db_query('SELECT * FROM some_table');
print(row);
```

Pseudo example for non-blocking I/O:

```
db_query('SELECT * FROM some_table',
function (row) {
print(row);
});
```

This example uses lambdas (anonymous functions) like they are used in JavaScript (JS) all the time. JS makes heavy use of events, and that's exactly what callbacks are about. Once the action is complete, an event is fired which triggers the callback. This is why it is often referred to as an *evented model* or also *asynchronous model*. The implementation of this model uses a loop that processes and fires these events. That's why it is called an *event queue* or *event loop*.

As the programming language we have used server-side JavaScript. JS is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. It was formalized in the ECMAScript language standard and is primarily used in the form of client-side JS, implemented as part of a Web browser in order to provide enhanced user interfaces and dynamic websites. This enables programmatic access to computational objects within a host environment.

JavaScript's use in applications outside Web pages - for example in PDF documents, site-specific browsers,

and desktop widgets - is also significant. Newer and faster JavaScript VMs and frameworks built upon them, notably "Node.js" - the one used in our implementation, have also increased the popularity of JS for server-side web applications.

JavaScript uses syntax influenced by that of C. It copies many names and naming conventions from Java, but the two languages are otherwise unrelated and have very different semantics. The key design principles within JS are taken from the Self and Scheme programming languages.

"Node.js" is a software system designed for writing highly-scalable internet applications, notably web servers. Programs are written in JS, using event-driven, asynchronous I/O to minimize overhead and maximize scalability. Node.js consists of Google's V8 JavaScript engine plus several built-in libraries. Similar environments written in other programming languages include Twisted for Python, Perl Object Environment for Perl, libevent for C and EventMachine for Ruby. Unlike most JS programs, it is not executed in a web browser, but is instead a server-side JS application.

In our current ReC^{2}S implementation we have all the modules detailed in Chapter 4 written in server-side JS. The front-end was written using Grails framework. Grails is an open source web application framework which uses the Groovy programming language, which is in turn based on the Java platform. It is intended to be a high-productivity framework by following the "coding by convention" paradigm, providing a stand-alone development environment and hiding much of the configuration detail from the developer.

The lease manager was split in two parts: an API wrapper, accessible to the external world and the actual lease containers. Both of them are written in a clustered way - both the master process and children processes are backed up by replicas. The scheduler was implemented also in JS. We implemented its running behavior to provide more throughput. First of all, when starting the scheduler module we check the database for existing leases. If are found, we load them in the scheduler's running leases cache. After that, at certain periods of time we take a lease from the lease manager module and we check it - if it is an "URGENT" one we start in on a certain hypervisor and save it in the running leases cache. If it is not, we assume it is a delayable one and compute a delay to start it. We have chosen a lazy paradigm of allocating the virtual machines from the lease.

The hypervisor manager currently implements ways to communicate with VMware ESXi version 5 and VirtualBox version 4: creating, starting, stopping, pausing, resuming and cloning a new or existing virtual machine.

Chapter 7

Results

7.1 Testbed configuration

For testing, the modules have been implemented and split across multiple workstations, as can be seen in Figure 7.1.

They are represented as a cluster of servers, each having the functionality presented in detail in Chapter 4. As it can be seen, the entire modules found in the dotted perimeter, called "Management modules", can also be ran all on one workstation. Elements like network switches are not represented in order not to burden the graphic, but the IP address of the hosts are kept. For the "Hypervisor / Cloud Interface" three distinct hypervisor servers have been used, each having its own security and load policies.

7.2 Experimental results

7.2.1 Scheduler experimental results

To optimize the speed of deployment of the virtual machines requested on a specific lease, we configured a list of virtual machines with predefined software stacks. For example we have in the repository virtual machines templates containing already-installed databases, and different other frameworks. This has a great impact on the entire system because deployment is made faster. This time is won from the time used by the virtual machines images to install the specific software stacks.

To test the scheduler and the system as a whole we have tested all three scheduling algorithms. The scenario was as follows. Suppose that we have 8 users logged in in our system. They each submit a lease, but all with the same time of start. This scenario is a common found in actual production environments.

The hardware platform used was composed from an AMD Phenom II X6, with 8GB RAM, RAID0 configured hard-disks running VirtualBox as hypervisor and an Intel DualCore, 4GB RAM as the scheduler. The network used is 10/100 MB.

In case of FIFO we obtained the following results, as seen in Table 7.1.

In the first column we see the order in which the leases were ran. This is 1, 2, 3, 4, 5, 6, 7 and 8. On the second column we can see the total lease time spent in creation of a lease.

To be more clear we present in Figure 7.2 a second chart in which we can see all the times above in a more friendly way. The actual timeline can be presented in Figure 7.3.

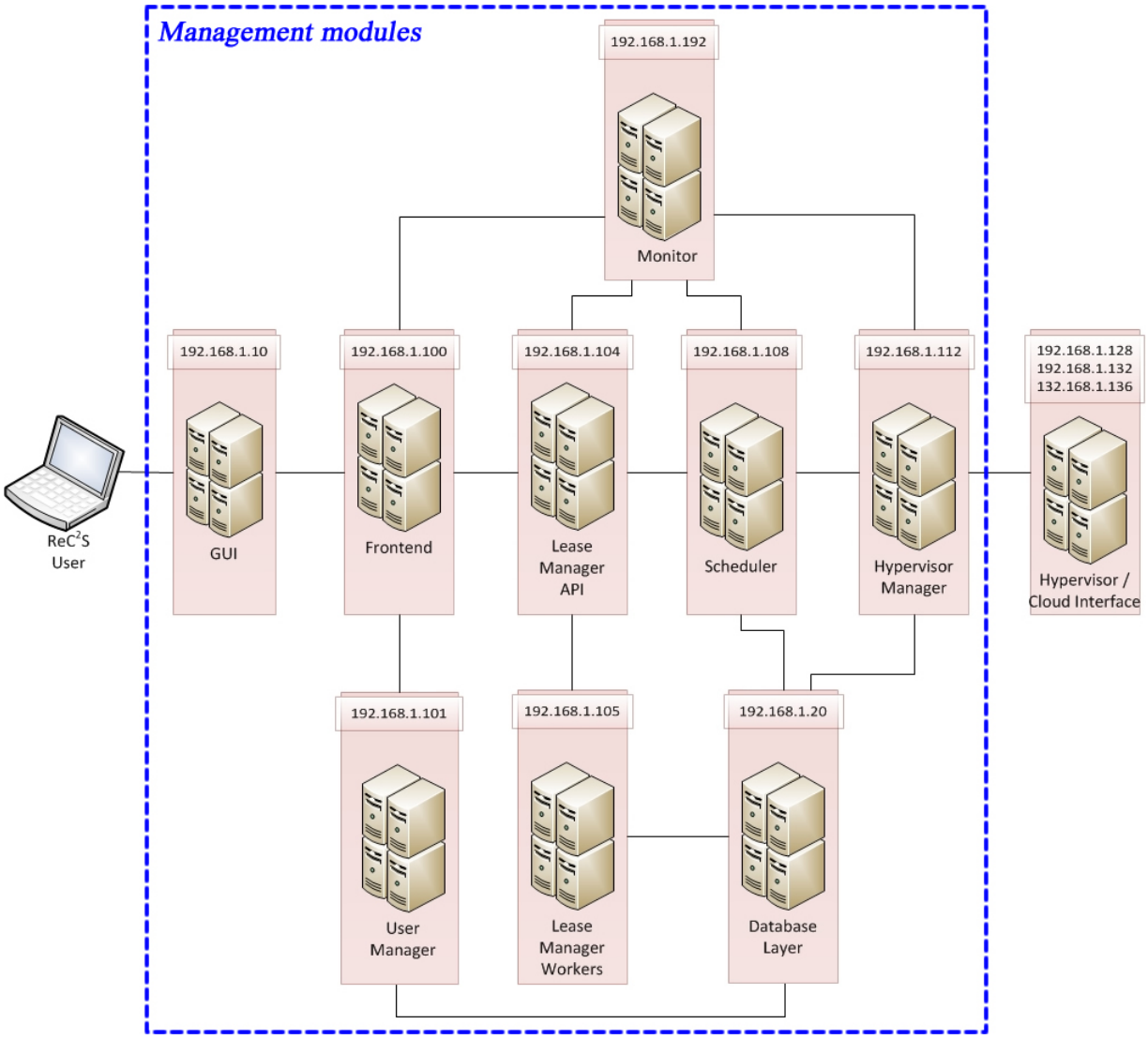

Figure 7.1: *Mapping modules to workstations.*

Table 7.1: *FIFO result table*

#	Lease creation time (s)	Resource lookup time (s)	Lease lookup time (s)	Virtual machine clone time (s)	Create time (s)
1	9.939	0.100	0.560	9.029	0.250
2	9.940	0.098	0.560	9.029	0.253
3	9.948	0.098	0.560	9.029	0.261
4	9.927	0.097	0.560	9.029	0.241
5	9.926	0.101	0.560	9.029	0.236
6	9.939	0.095	0.560	9.029	0.255
7	9.931	0.098	0.560	9.029	0.244
8	9.951	0.098	0.560	9.029	0.264

Thanks to our included Virtual Machine preloader, before the actual lease are created, the virtual machine template (800MB) is transferred to the destination. This took about 82 seconds in our test environment. The

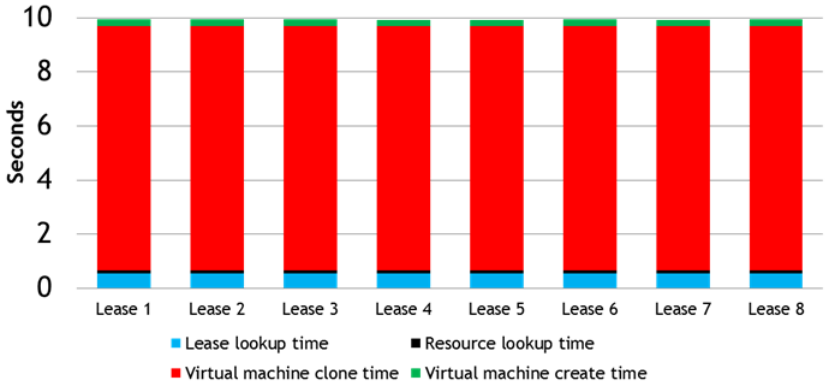

Figure 7.2: *FIFO result table.*

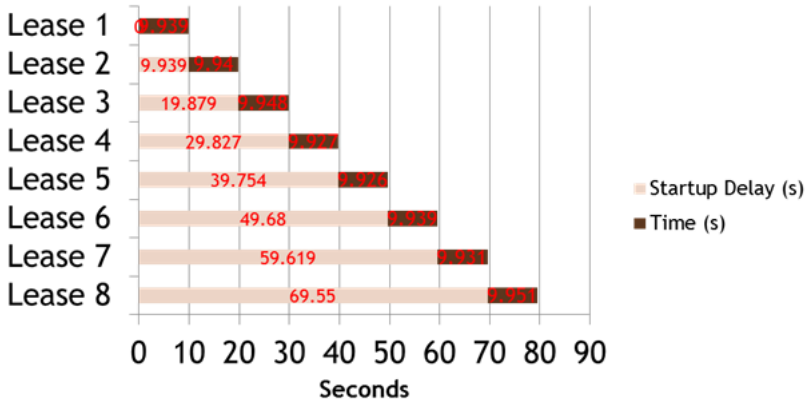

Figure 7.3: *FIFO result chart.*

total time spent was $(\sum Leasecreationtime) + 82 = 156.735seconds$

In case of SJF we obtained the following results, as seen in Table 7.2 and Figure 7.4, Figure 7.5.

Table 7.2: *SJF result table*

#	Lease creation time (s)	Resource lookup time (s)	Lease lookup time (s)	Virtual machine clone time (s)	Create time (s)
2	9.940	0.098	0.560	9.029	0.253
4	9.927	0.097	0.560	9.029	0.241
7	9.931	0.098	0.560	9.029	0.244
1	9.939	0.100	0.560	9.029	0.250
3	9.948	0.098	0.560	9.029	0.261
8	9.951	0.098	0.560	9.029	0.264
5	9.926	0.101	0.560	9.029	0.236
6	9.939	0.095	0.560	9.029	0.255

We can see from the previous table that the lease are picked to be run in a different order, 2, 4, 7, 1, 3, 8, 5 and 6. The total time spent in this case was $(\sum Leasecreationtime) + 82 = 155.947seconds$

In case of ReC2Sched we have obtained the following results, as seen in Table 7.3 and Figure 7.6, Figure 7.7.

We can easily conclude that the times are lower than the previous two engines. Also, because the lease

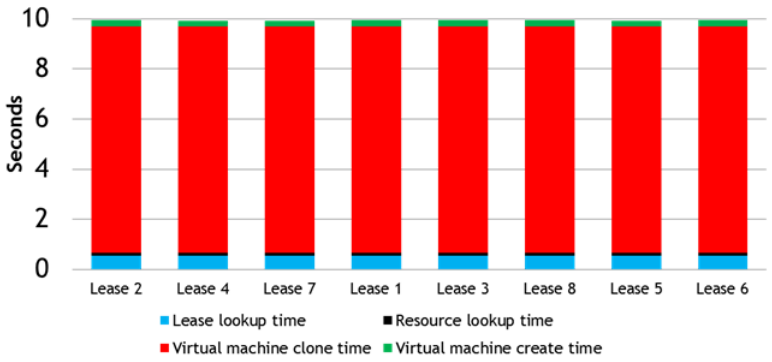

Figure 7.4: *SJF result table.*

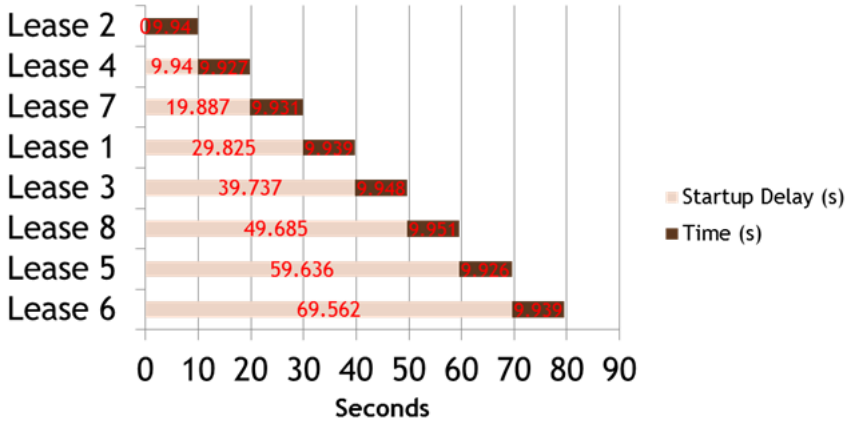

Figure 7.5: *SJF result chart.*

Table 7.3: *ReC2Sched result table*

#	Lease creation time (s)	Resource lookup time (s)	Lease lookup time (s)	Virtual machine clone time (s)	Create time (s)
1	9.768	0.055	0.560	9.029	0.124
2	9.767	0.056	0.560	9.029	0.122
3	9.767	0.056	0.560	9.029	0.122
4	9.772	0.056	0.560	9.029	0.127
5	9.764	0.056	0.560	9.029	0.119
6	9.766	0.055	0.560	9.029	0.122
7	9.771	0.057	0.560	9.029	0.125
8	9.766	0.055	0.560	9.029	0.122

are running in parallel, the total time was $Max(startleasetime + delay) + 82 = 92.264seconds$ which is the lowest obtained in our test.

7.2.2 ReC^{2}S experimental results

In order to test our implementation, besides the scheduler part which is thoroughly tested in our previous work published in [1], we used the Node.JS module called "node-inspector" which allowed us to get all parameters from the V8 virtual machine. A sample of output can be seen in Figure 7.8 We ran an analysis both on the

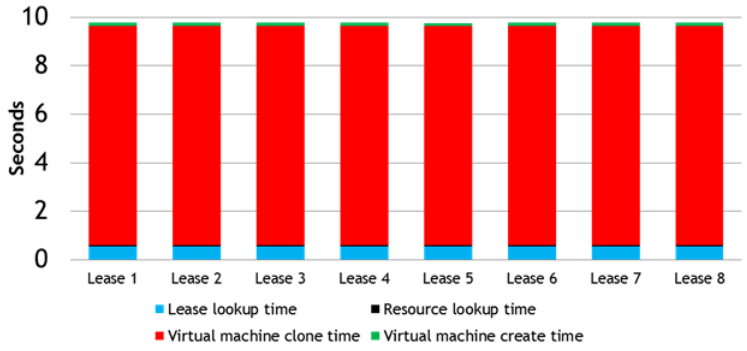

Figure 7.6: *ReC2Sched result table.*

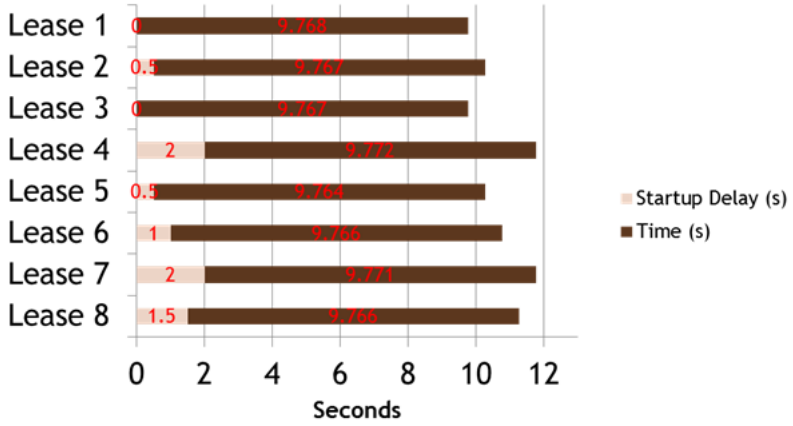

Figure 7.7: *ReC2Sched result chart.*

"Lease Manager" and "Hypervisor Manager" modules with 5 different leases, all having "Advanced Reservation" type. The results are presented in the following tables.

In Table 7.4 we can see on the second column the time needed for a lease to be created on the system. On the third and fourth column we can see the amount of time needed to check the lease for consistency and storage. In Table 7.5 we can see on the second column the time needed for the retrieval of an lease from the storage and on the last column we see the amount of time needed by the Hypervisor Manager to check a lease resource availability on the system.

Table 7.4: *Lease Manager result table*

#	Lease creation time (ms)	Lease check-up (ms)	Lease store (ms)
1	204	10	1
2	205	11	1
3	289	11	1
4	208	10	1
5	262	10	1

In Figure 7.9 can be seen an screenshot from the Firebug addon, with data from the "GUI" module. Firebug is a web development tool that facilitates the debugging, editing, and monitoring of any website's CSS, HTML, DOM, XHR, and JavaScript. One of the most usefull tab is its net panel, that can monitor URLs that the browser requests. This panel can also estimate the time each asset took to load. We can see clearly the time needed for a lease to be persisted, at the end of each row from the Firebug log.

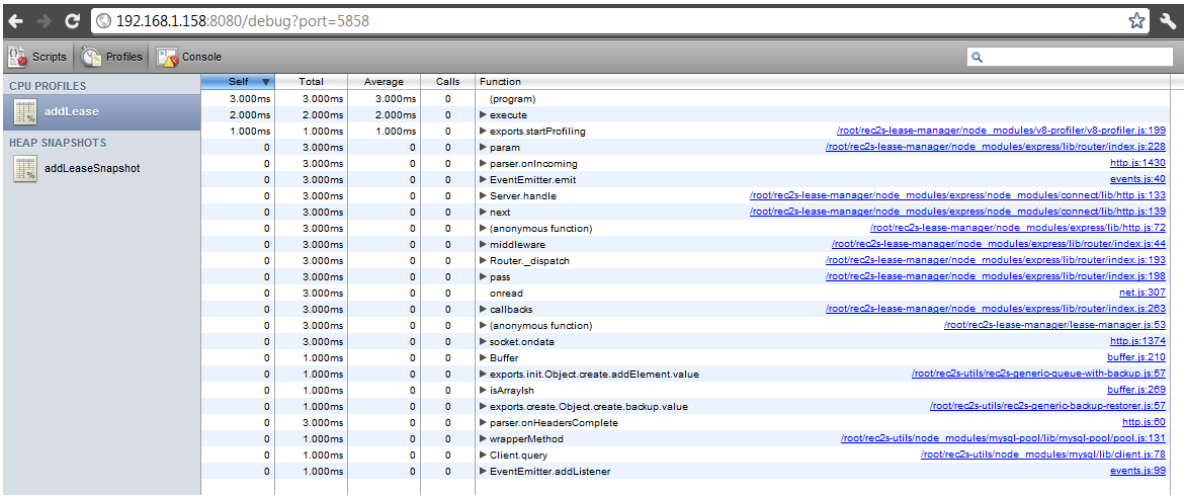

Figure 7.8: *"node-inspector" browser window.*

Table 7.5: *Hypervisor Manager result table*

#	Lease retrieve (ms)	Lease resource check (ms)
1	3	25
2	3	28
3	3	51
4	3	26
5	3	39

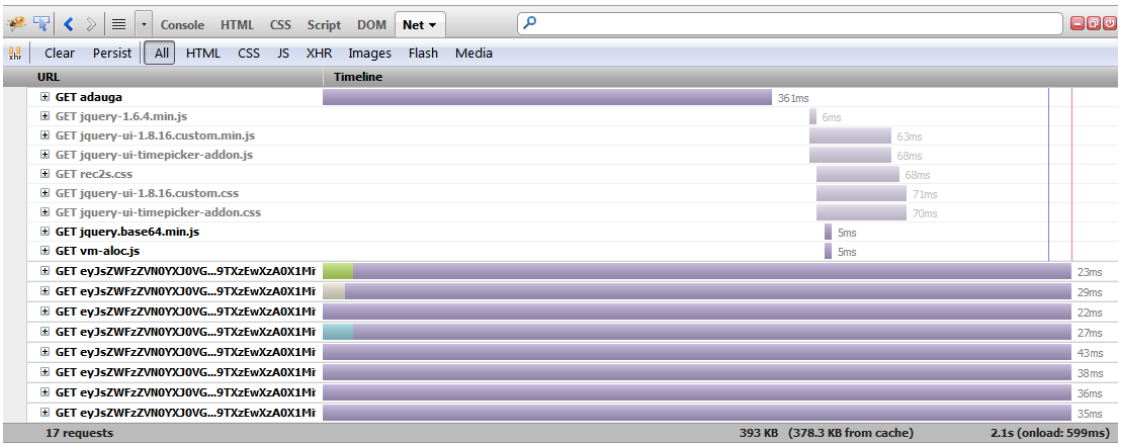

Figure 7.9: *"GUI" add lease from Firebug.*

Chapter 8

Conclusions and Future Work

In this thesis we presented a novel solution to provide reliability and security for Cloud users. Our approach takes the form of a complete framework on top of an existing Cloud infrastructure and we have described each of its layers and characteristics. Furthermore, the experimental results prove its efficiency and performance.

Our work is focused on increasing reliability, safety, security and availability of Distributed Systems. The characteristics of such systems present problems when tackling with secure resource management due to its heterogeneity and geographical distribution. We presented the design of a hierarchical architectural model that allows users to seamlessly scale workloads both vertically and horizontally while preserving scalability of large scale distributed systems.

As future work we intend to continue in this research direction in order to further optimize the scheduling algorithms, as well as the modules of the framework. We also intend to offer more complex security solutions which would greatly improve the usability of the system. A partial homomorphic encryption (PHE) scheme is going to be implemented.

Of course, further testing using more complex scenarios and a thin integration with other existing Cloud infrastructures would also help us to further improve our solution.

Bibliography

[1] A. Pătraşcu, C. Leordeanu, C. Dobre and V. Cristea, "ReC^{2}S: Reliable Cloud Computing System", European Concurrent Engineering Conference, Bucharest, 2012.

[2] A. Pătraşcu, C. Leordeanu, V. Cristea, "Scalable Service based Antispam Filters", Proceedings of First International Workshop on the Service for Large Scale Distributed Systems(Sedis 2011) in conjunction with the EIDWT 2011 conference, Tirana, 2011, ISBN 978-0-7695-4456-4

[3] D. Maimuţ, A. Pătraşcu and E. Simion, "Homomorphic encryption schemes", 5th International Conference on Security for Information Technology and Communications (SECITC), Bucharest, 2012.

[4] A. Pătraşcu, D. Maimuţ, and E. Simion, "New Directions in Cloud Computing. A Security Perspective.", COMM International Conference, Bucharest, 2012.

[5] M. A. Rappa, "The utility business model and the future of computing systems", IBM Systems Journal, 43(1):32-42, 2004.

[6] C. S. Yeo et al., "Utility computing on global grids", Chapter 143, Hossein Bidgoli(ed.), The Handbook of Computer Networks, ISBN: 978-0-471-78461-6, John Wiley and Sons, New York, USA, 2007.

[7] I. Foster and S. Tuecke, "Describing the elephant: The different faces of IT as service", ACM Queue, 3(6):26-29, 2005.

[8] M. P. Papazoglou and W.-J. van den Heuvel, "Service oriented architectures: Approaches, technologies and research issues", TheVLDBJournal, 16:389-415, 2007.

[9] H. Kreger, "Fulfilling the Web services promise", Communications of the ACM, 46(6):29, 2003.

[10] B. Blau, D. Neumann, C. Weinhardt, and S. Lamparter, "Planning and pricing of service mashups", in Proceedings of the 2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, Crystal City, Washington, DC, 2008, pp.19-26.

[11] C. Gentry, "Fully homomorphic encryption", 2008.

[12] S. Tayal, "Tasks Scheduling optimization for the Cloud Computing System", (IJAEST) INTERNATIONAL JOURNAL OF ADVANCED ENGINEERING SCIENCES AND TECHNOLOGIES, Vol5, 2011

[13] S. K. Garg, C. S. Yeo, A. Anandasivam and R. Buyya, "Energy efficient Scheduling of HPC application in Cloud Computing environments", 2009

[14] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility", 2008

[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph and R. Katz, "Above the clouds: A Berkeley view of cloud computing", UC Berkeley Reliable Adaptive Distributed Systems Laboratory White Paper, 2009.

[16] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval", J. ACM, 45, 6 (1998), pp. 965-981.

[17] N. Carr, "The Big Switch: Rewiring the World, from Edison to Google.W. W. Norton and Co.", New York, 2008.

[18] C. Catlett, "The philosophy of TeraGrid: Building an open, extensible, distributed TeraScale facility", in Proceedings of 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, Berlin, Germany, 2002, p. 8.

[19] F. Gagliardi, B. Jones, F. Grey, M. E. Begin, and M. Heikkurinen, "Building an infrastructure for scientific grid computing: Status and goals of the EGEE project", Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 363(1833):1729, 2005.

[20] I. Foster, "Globus toolkit version 4: Software for service-oriented systems", Journal of Computer Science and Technology, 21(513-520), 2006.

[21] R. Buyya and S. Venugopal, "Market oriented computing and global Grids: An introduction, in Market Oriented Grid and Utility Computing", R. Buyya and K. Bubendorfer (eds.), John Wiley and Sons, Hoboken, NJ, 2009, pp. 24-44.

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

[23] J. Broberg, S. Venugopal, and R. Buyya, "Market-oriented Grid and utility computing: The state-of-the-art and future directions", Journal of Grid Computing, 6:255-276, 2008.

[24] R. P. Goldberg, "Survey of virtual machine research", IEEE Computer, 7(6):34-45,1974.

[25] R. Uhlig et al., "Intel virtualization technology", IEEE Computer, 38(5):48-56, 2005.

[26] P. Barham et al., "Xen and the art of virtualization", in Proceedings of 19th ACM Symposium on Operation Systems Principles, New York, 2003, pp. 164-177.

[27] Xen.org Community, http://www.xen.org, 22/4/2010.

[28] Citrix Systems Inc., XenServer, http://www.citrix.com/XenServer, 22/4/2010.

[29] Oracle Corp., Oracle VM, http://www.oracle.com/technology/products/vm, 24/4/2010.

[30] KVM Project, Kernel based virtual machine, http://www.linux-kvm.org, 22/4/2010.

[31] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, KVM: The Linux virtual machine monitor, in Proceedings of the Linux Symposium, Ottawa, Canada, 2007, p. 225.

[32] VMWare Inc., VMWare Virtual Appliance Marketplace, http://www.vmware.com/ appliances, 22/4/2010.

[33] Amazon Web Services Developer Community, Amazon Machine Images, http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID5171, 22/4/2010.

[34] Distributed Management Task Force Inc, Open Virtualization Format, Specification DSP0243 Version 1.0.0, 2009.

[35] J. Matthews, T. Garfinkel, C. Hoff, and J. Wheeler, Virtual machine contracts for datacenter and cloud computing environments, in Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds, 2009, pp. 25-30.

[36] International Business Machines Corp., An architectural blueprint for autonomic computing, White Paper Fourth Edition, 2006.

[37] M. C. Huebscher and J. A. McCann, A survey of autonomic computing degrees, models, and applications, ACM Computing Surveys, 40:1-28, 2008.

[38] N. Fallenbeck, H.J. Pinch, M. Smith and B. Freisleben, "Xen and the art of cluster scheduling", Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing, IEEE, 2006.

[39] P. Ruth, P. McGachey and D. Xu, "VioCluster: Virtualization for dynamic computational domains", Proceedings of the IEEE International Conference on Cluster Computing, IEEE, 2005.

[40] N. Kiyanclar, G.A. Koenig, and W. Yurcik, "Maestro-VC: A paravirtualized execution environment for secure on-demand cluster computing", Proceedings of the 6th IEEE International Symposium on Cluster Computing and Grid, IEEE, 2006.

[41] D. Irwin, J. Chase, L. Grit, A.Yumerefendi, D. Becker and K.G. Yocum, "Sharing networked resources with brokered leases", USENIX Technical Conference, 2006.

[42] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu and X. Zhu, "From virtualized resources to virtual computing grids: the In-VIGO system", Future Generation Computer Systems, 2005

[43] http://www.nist.gov/itl/cloud/index.cfm

[44] http://www.vmware.com/

[45] VMWare Inc., VMware vSphere, http://www.vmware.com/products/vsphere/, 22/4/2010.

[46] http://aws.amazon.com/ec2

[47] http://en.wikipedia.org/wiki/Scheduling_%28computing%29