UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS TELEMÁTICOS

Título: **"Development of a management system for virtual machines on private clouds"**

Autor: D. Mattia Peirano
Tutor: D. Juan Carlos Dueñas López
Departamento: Departamento de Ingeniería de Sistemas Telemáticos

Tribunal calificador:

- Presidente: D. Juan Carlos Dueñas López
- Vocal: D. David Fernández Cambronero
- Secretario: D. Gabriel Huecas Fernández-Toribio

Fecha de lectura: 15 de Marzo 2013

Calificación:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

Development of a management system for virtual machines on private clouds

Mattia Peirano
2013

To my sister

Acknowledgements

During these five years between Politecnico di Torino and Universidad Politécnica de Madrid, I have had the chance to meet many people along the way: some of them have come along with me, others joined just to share a stretch of road together and then follow on their own.

I want to begin by expressing my gratitude to those people who made this experience possible. First I want to thank my family that allowed me to embark on this adventure: they supported me during this year away with its ups and downs, and, even from afar, with their great love, they have been present in some way.

Thanks to Juan Carlos for allowing me to finish my studies, by welcoming me in his research group. He has been my guiding light since the very beginning of my adventure at the ETSIT, which began with his own lecture on Telematics Services Architecture. I have been very lucky in choosing that course: thanks for this opportunity you gave me.

Thanks to Rodrigo, workmate, fellow of lunches and breakfasts; he has been able to teach me a lot and to grant me a bit of his experience. His passion at work will allow him to be, besides a successful writer, a great professor, as those who fill the classroom with students sitting on the steps. Between chocolate cookies and our inevitable croissants with Eva, he introduced me to the true essence of Spanish customs and traditions.

Thanks to professor Guido Marchetto who allowed me to finish my degree with his supervision and support.

Thanks to Valentina who has been able to give me a bit of her strength, always encouraging me in all my important decisions. She was able to find hundreds of reasons not to give up; thanks for supporting and waiting for me for all this time.

Thanks to Julia, flatmate with whom I had the pleasure of sharing the best part of this journey. I hope our paths will cross again, despite the large distance. Thanks to Sara, the first who welcomed me in this city and in this school. Thanks to the many people I have had the pleasure to meet during this experience at both universities: somehow, everyone of you left me something, and I hope I've left something to you, too.

And a final thanks to my old friends, who, even if far, every time I came back, were able to make me feel I had never left.

Grazie a tutti

Madrid, February 20, 2013

Abstract

Cloud computing and, more particularly, private IaaS, is seen as a mature technology with a myriad solutions to choose from. However, this disparity of solutions and products has instilled in potential adopters the fear of vendor and data lock-in. Several competing and incompatible interfaces and management styles have increased even more these fears. On top of this, cloud users might want to work with several solutions at the same time, an integration that is difficult to achieve in practice. In this Master Thesis I propose a management architecture that tries to solve these problems; it provides a generalized control mechanism for several cloud infrastructures, and an interface that can meet the requirements of the users. This management architecture is designed in a modular way, and using a generic information model. I have validated the approach through the implementation of the components needed for this architecture to support a sample private IaaS solution: OpenStack.

Keywords: cloud computing; private IaaS; cloud management; management architecture; cloud interoperability; OpenStack.

Riassunto

Il cloud computing, e più in particolare i servizi IaaS privati, sono ormai visti come una tecnologia matura che presenta una miriade di soluzioni tra cui è possibile scegliere. Tuttavia, questa disparità di soluzioni e prodotti ha instillato nei potenziali utenti la paura di legarsi definitivamente ad un determinato fornitore o tecnologia. L'avvento di nuove interfacce di gestione, incompatibili tra loro, e di diversi stili di amministrazione ha dato ancor più voce a questi timori. In primo luogo, gli utenti del cloud possono voler lavorare contemporaneamente con diverse soluzioni tecnologiche, ma tale integrazione risulta difficile da realizzare nella pratica. In questa tesi magistrale propongo un modello di architettura di gestione che cerca di affrontare tali problemi offrendo un metodo comune di controllo per tecnologie cloud eterogenee e un'interfaccia che può essere adattata alle esigenze dell'utente. Questa architettura è stata progettata in modo tale da avere una struttura modulare e utilizzando un modello di informazione generico. Abbiamo convalidato il nostro approccio attraverso lo sviluppo e la implementazione dei componenti necessari, facendo riferimento a una specifica soluzione di IaaS privato: OpenStack.

Parole chiave: cloud computing; IaaS privati; gestione del cloud; architettura di gestione; interoperabilità tra cloud; OpenStack.

Resumen

La computación en la nube y, más concretamente, las IaaS privadas, están consideradas como una tecnología madura que presenta una multitud de soluciones entre las cuales elegir. Sin embargo, esta disparidad de soluciones y productos ha inculcado en los potenciales usuarios el miedo a ligarse perennemente a un proveedor o tecnología específicos. La aparición de varias interfaces de gestión, incompatibles entre ellas, y distintos estilos de administración han dado voz, aún más, a estos temores. Además, los usuarios de los servicios en la nube desean trabajar con varias soluciones al mismo tiempo, y ésta es una integración difícil de lograr en la práctica. En este Proyecto de Fin de Carrera propongo un modelo de arquitectura de gestión que no sólo trata de hacer frente a estos problemas, sino que ofrece una forma común de manejar varias soluciones de computación en la nube, y una interfaz que se puede adaptar a las necesidades del usuario. Esta arquitectura de gestión está diseñada de una manera modular, usando un modelo de información genérico. Hemos validado nuestro enfoque a través de una implementación de los componentes de tal arquitectura escogiendo como referencia una solución de IaaS privado específica: OpenStack.

Palabras clave: computación en la nube; IaaS privada; gestión de la nube; arquitectura de gestión; interoperabilidad entre nubes; OpenStack.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During recent years the term "Cloud Computing" has become very popular and many companies has changed their IT infrastructures and services, to fit in this new tendency. Cloud Computing has become a revolutionary technology that has changed the way services and resources are managed, enabling the access to computing power as an utility. For this reason, Cloud Computing could be considered the third evolution of the IT technology, after the mainframe and the client-server paradigms. This technology offers a great variety of services, at different levels of the service stack:

- IaaS - Infrastructure as a Service: Providing raw computing resources.
- PaaS - Platform as a Service: Providing elastic software platforms.
- SaaS - Software as a service: Providing software deployed onto the cloud.

In each case, the cloud approach offers to the user ease of access and management, freeing him from the troubles of the administration of physical machines. The user does not need to care what is happening behind his software/platform/infrastructure, nor he does need to know where his software is run or how many computers are involved in the computational work. These features change the traditional approach to IT industry.

Moreover, Cloud Computing can also save resources and optimize their management when used in a private infrastructure, reducing problems and management costs. However, to truly realize this, current private solutions should improve its optimization and automation capabilities.

1.1. Context

In mid-October 2001, IBM released a manifesto observing that the main obstacle to further progress in the IT industry was a looming software complexity crisis [1, 2]. The only sensible option was autonomic computing; computing systems that can manage themselves following high-level policies from administrators [2]. Public cloud computing solutions, such as Amazon EC2, provide customers with a great availability due to the enormous pool of resources they own. However, in private clouds the available resources are limited and a fair and sensible utilization is paramount. One does not need only to support applications actually running, but also to turn off unused machines to save on energy. Some authors claim that "immediate scalability and resources usage optimization are key elements for the Cloud. These are provided by increased monitoring, and automation of resources management in a dynamic environment" [3]. Therefore, this should be the main target of a complete cloud autonomic system: Markus Klems affirmed "With monitoring and increasing automation of resource provisioning we might one day wake up in a world where we don't have to care about scaling our web applications because they can do it alone" [4].

This *Master Thesis* is part of a bigger work that tries to follow this path, contributing to the creation of an autonomic management system for IaaS private clouds. Its central element will be a cloud manager, which will implement an autonomic loop (figure 1.1) that will manage both the physical and virtual infrastructure of the cloud. Starting from a data retrieval system [5] this will use the captured information to define a set of management actions that will, in turn, be performed over the managed elements of the cloud.

To realize this big picture, I propose for my *Master Thesis* a component that will be responsible for this last part: the "effector" of the loop, able to interface with the cloud infrastructure.

1.2. Technical Objectives

The main objective is to create a system able to manage virtual instances of cloud nodes in order to optimize, following some management policies, the use of physical resources. The system will be able to choose or change the location of virtual instances, trying to implement a resource distribution given by an external element (not developed in this *Master Thesis*): a reasoning engine. Therefore, this system will need to interact with a cloud controller. Working together they can achieve better performances and nearly optimal resource utilization. In order to achieve the proposed aim we will focus on the following sub-objectives:

1. Development of a virtual machine (VM) controller prototype, able to manage

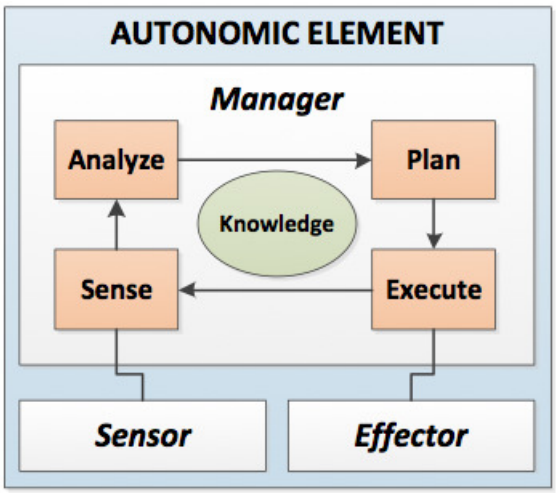

Figure 1.1: Autonomic element

all configuration and runtime parameters concerning the VM lifecycle and the initial configuration set. The controller should be able to run or stop virtual instances, assigning them resources according to an external plan.

2. Development of an interpreter that allows working with different private cloud technologies without changing the entire controller code and giving our system a greater level of interoperability. This way, just changing the interpreter implementation, the controller can be used with a different cloud technology.

Although the developed software will be able to work with the monitoring and reasoning components of the autonomic loop (figure 1.1), it will also be a standalone system. It will implement a set of control interfaces that can be easily accessed by third party components. In fact, these interfaces will be defined in this *Master Thesis*. Therefore, the proposed work will be capable of interacting with other external systems; it can work independently from the other elements of the autonomic system and could be also employed in other environments.

1.3. Didactic Objectives

This project also aims to complete the formation of the author as master degree student. This final work, being a software development project, allows to consolidate and apply in a practical way different concepts which had been studied along the degree. This type of work allows the candidate to improve his teamwork skills and management abilities in the context of software engineering projects.

From a more technical point of view, the work will force the candidate to learn and study new technologies and tools which could be the basis for possible future job opportunities.

1.4. Work Plan

The project followed a waterfall development cycle, with a duration of 6 months. The main part of the project was divided in 4 primary steps: analysis, design, implementation and test. The documentation task ran in parallel with the whole project leading to the final document writing. The following Gantt schema shows the proposed project lifetime.

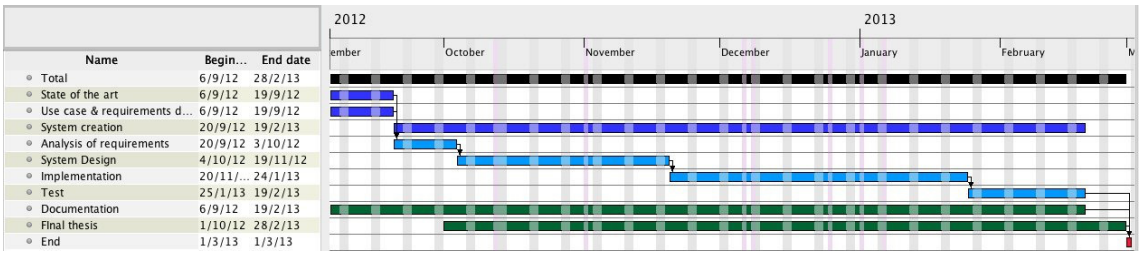

Figure 1.2: Project plan

As we can see in figure 1.2 the main steps were divided in smaller tasks, making the project better organized and manageable:

State of the art: analysis of the existing technologies and choice of which can be used in the project. Analysis of the cloud technologies, especially focusing on private IaaS solutions; review of the physical and logical structure of the cloud, analyzing the interconnection between the layers composing the system. This task also includes an analysis of the communication capabilities between the cloud technology and the external controllers that I developed, and the installation of the OpenStack cloud service on the machines of the laboratory.

Use case and requirements definition: identification of the system main requirements, starting from the use case analysis.

System creation: this task can be separated in a set of 4 subtasks:

- Requirements analysis
- System design

- Implementation
- Test on the infrastructure, compilation of results and an evaluation of the feasibility of a possible integration with an external reasoning system acting on the cloud.

Documentation and document redaction: report of the achievements and documentation of the performed work.

1.5. Document Structure

This *Master Thesis* consists of eight chapters. This division approximately corresponds to the natural cycles of the software development. In this first chapter I presented the context of the problem and set the main objectives to be met.

In the second chapter, I perform a detailed study of the state of the art in terms of cloud computing, monitoring, and other technologies that support the implementation of the system. In this section I explore the concept of cloud computing trying, to highlight the main characteristics of this technology.

The third chapter analyzes in detail all requirements the system must meet to resolve the planned issue. In this section I define so much the functional requirements of the software as the use cases in which it will be used. After it we arrive at a complete overview of the technical requirements of the system that I intend to design and develop.

The fourth chapter is the detailed design of the system structure, supported by UML diagrams that facilitate its comprehension. At this level, I introduce the features of the system that tries to satisfy the requirements defined in the previous chapter. This section also describes the data model that has been chosen to represent the cloud elements.

In the fifth chapter I describe the battery of tests performed in order to verify the correctness of the implemented system. I also included some brief metrics and statistics about the code, that could help in its evaluation.

The sixth chapter presents a brief overview over the main problems encountered during the deployment of the selected infrastructure.

The seventh chapter offers the idea for the development of a new service, which use as a basis the infrastructure I developed. This section describes the implemented service and lays the foundations for another, future, project.

The eighth chapter contains some final considerations and a several possible work lines for future projects related to this.

Finally, in the form of annexes, I provide samples of the configuration files used, and a brief user manual for the two developed services.

Chapter 2

State of the Art

In this chapter I provide a general overview of the Cloud Computing concept, his history and evolution. Subsequently I will describe the most interesting systems present in the actual state of the field and the technologies that will be adopted in the project. This part is the result of a process of documentation.

2.1. Cloud Computing Definition

In the literature many authors have tried to give a formal definition of the term "Cloud Computing", but usually only some of its aspects are treated, while other are not taken into account. According to the definition of the National Institute of Standard and Technology (NIST)[6] *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".*

Vaquero [3] instead defines Cloud computing as *"a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs".*

Both definitions describe the same "object", picturing it from different angles; it is necessary to admit that it is difficult to find a definition that collects synthetically every aspect of cloud computing. Because of this, in the next section I analyze the main characteristics of this technology.

2.2. Main Features

Some authors had tried to define the differences between cloud computing and grid computing technology. In this paragraph I will focus on the aspects that distinguish it from others computing systems [6].

- **On-demand self-service:** computing resources are delivered to the consumer without the human intervention. The system is able to scale via dynamic on-demand provisioning, according to the user needs.
- **Broad network access:** cloud services are available over the network, enabling users to access them regardless of their position or the platform they use (e.g. smart-phones, laptops, etc.).
- **Resource pooling:** cloud computing providers provide pooled computing capabilities that are used to serve consumers using a multitenancy model; this model reduces costs and utilization, since virtual and physical resources are assigned based on user demand. In this case virtualization plays an important role, providing maximum flexibility to configure various partitions of resources on the same physical machine.
- **Rapid elasticity:** resources provisioning can adapt very rapidly to the instant demand, optimizing resource utilization and increasing the perceived quality, giving to the consumer the impression of a single dedicated resource [3].
- **Measured service:** clouds allows for transparent tracing and report of user consumption. This characteristic enables the adoption of a pay-per-use model where users are billed based on their consumption.

2.3. Service Models

In this chapter I present, following a bottom-up approach, the different service models that had been defined in the cloud computing environment. I start presenting the lower layer, which is closer to the hardware infrastructure and directly interacts with it.

2.3.1. Infrastructure as a Service - IaaS

This is the most basic cloud service model that provides computing, storage and network resources. Usually the features are achieved through virtualization: service providers split, assign and dynamically resize virtual resources to build ad-hoc systems as demanded by customers [3]. According to this, it is necessary to

distinguish between two types of resources: physical resources set (PRS) and virtual resources set (VRS) services [7]. In an infrastructure of this type, users can access resources, manage operating systems, storage and deploy applications.

Usually, even storage and network services are virtualized in order to provide fully customized capabilities. VRS services are the most common and widely adopted (e.g. Amazon EC2 [8], OpenNebula [9], OpenStack [10] etc.).

2.3.2. Platform as a Service - PaaS

This is a model where providers deliver a platform (Operating System, programming language environment, libraries etc.). On top of it users can deploy their applications, without being troubled with the system configuration.

Providers offer to their customers a black box with auto-scale capabilities, able to change dynamically the allocation of resources, depending on the workload (application demand). Indeed, the underlying level is not accessible by users, which cannot manage the cloud infrastructure, and only use the overlying services (e.g. Heroku [11], Google AppEngine [12], Cloud Foundry [13], etc.).

2.3.3. Software as a Service - SaaS

This is the last service model, where the user can access running software programs, deployed over the cloud, through the use of cloud clients such as PCs, laptops, smart-phones... The consumer cannot access any layer of the structure; he can solely use the deployed application, accessing it through a browser and a network connection. Using the programs do not require any particular installation or system configuration. This is the case of online word processors like GoogleDocs [14] or online game platforms like Gaikai [15].

2.4. Deployment Models

The NIST defines four types of deployment models, according to who owns the infrastructure, or who make use of it.

2.4.1. Private Cloud

A private cloud is a cloud infrastructure used only by one organization; it could be on-premises or hosted in another infrastructure. This model raze some criticism, due to the fact that the owner still have to deploy and manage the system, contradicting one of the main features of cloud computing. On the other hand, the adoption of an on-premises Private Cloud guarantee a higher level of security: sensible data

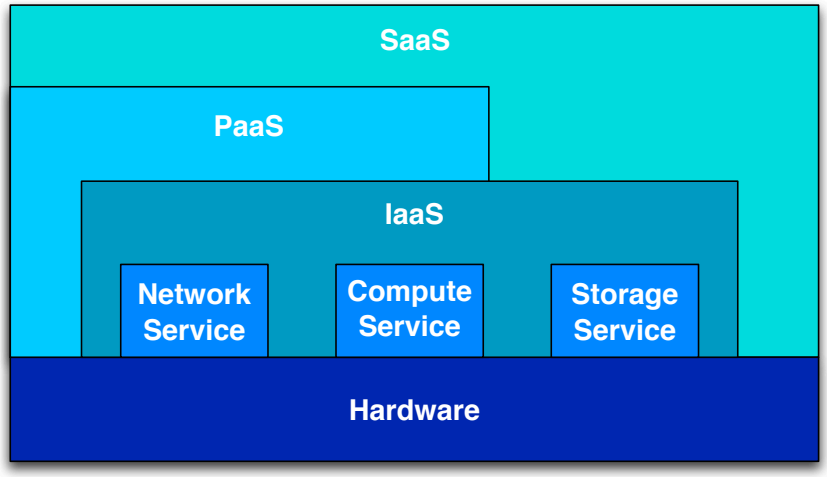

Figure 2.1: Service models

are stored in an internal infrastructure, instead of inside a third-party data center, where the control might be not so strict.

2.4.2. Community Cloud

This is a cloud deployment model where several companies share among them part of their resources, forming a kind of shared cloud. It can be managed by a third party, while resources can be hosted internally or externally.

2.4.3. Public Cloud

In this deployment model a company owns a cloud infrastructure and offers cloud computing services to third-party. Customers are usually billed according to a pay-per-use model, where only the effective utilization of resources is charged. This is the case of computing services providers like Amazon and Google.

2.4.4. Hybrid Cloud

This is a combination of two or more clouds (private and public) that continue to exist as independent systems, but are also bounded together. This bond is created by the sharing of resources.

2.4.5. Hosted Public Cloud vs. Hosted Private Cloud

Commercial companies (i.e. Citrix) , advertising about IaaS cloud products, talk about two types of private clouds: a "hosted private cloud" and a "hosted public cloud" service. Similarly, Microsoft proposes a similar offer but distinguishes between a private cloud on proprietary machines and a private cloud deployed on a hosted physical infrastructure. Sometimes it is not so clear where the border line between a Public Cloud service and Private Hosted Cloud is drawn.

Analyzing the matter we can assume that the difference consists in the resource allocation process: in a Private Hosted Cloud a user can purchase a specific amount of resources; while in a Public Cloud the resource allocation scales with the workload, without an established limit (on-demand); in this case the user will be charged only for the amount of resources that he uses.

Referring to the NIST definition, a private clouds infrastructure "is provisioned for exclusive use by a single organization comprising multiple consumers. It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises." [6] The author does not make distinction between the two implementations of private cloud, giving raise to some discussion on security issues.

2.5. Computing as Utility

The Gartner's Hype-Curve (figure 2.2) shows that in 2012 the expectations for cloud computing specific technologies are high: Hybrid Cloud Computing and Private Cloud Computing reside at the peak of the curve. On the other hand, the cloud computing element is falling in the "disillusionment" zone, while the Cloud/Web Platforms technology, which was present in the 2011 Gartner's curve, is no longer mentioned. Gartner analysts define Cloud Computing as one of the fastest-moving emerging technologies and the offered portrait (Hype Curve) seems to suggest that the global scenario is moving towards a state in which analytic insight and computing power are nearly infinite and cost effectively scalable [16].

Some authors claim that cloud computing is being transformed into a model of services, delivered in a manner similar to classic utilities, like electricity, water, gas, etc. This type of services must be available at any time and consumers pay just what they consume (pay-per-use) [6].

Obviously, this kind of service delivery model is applicable to an environment where a Service Level Agreement is defined and the provisioning granted. This could be the new goal that Information Technology companies will try to reach for cloud computing systems.

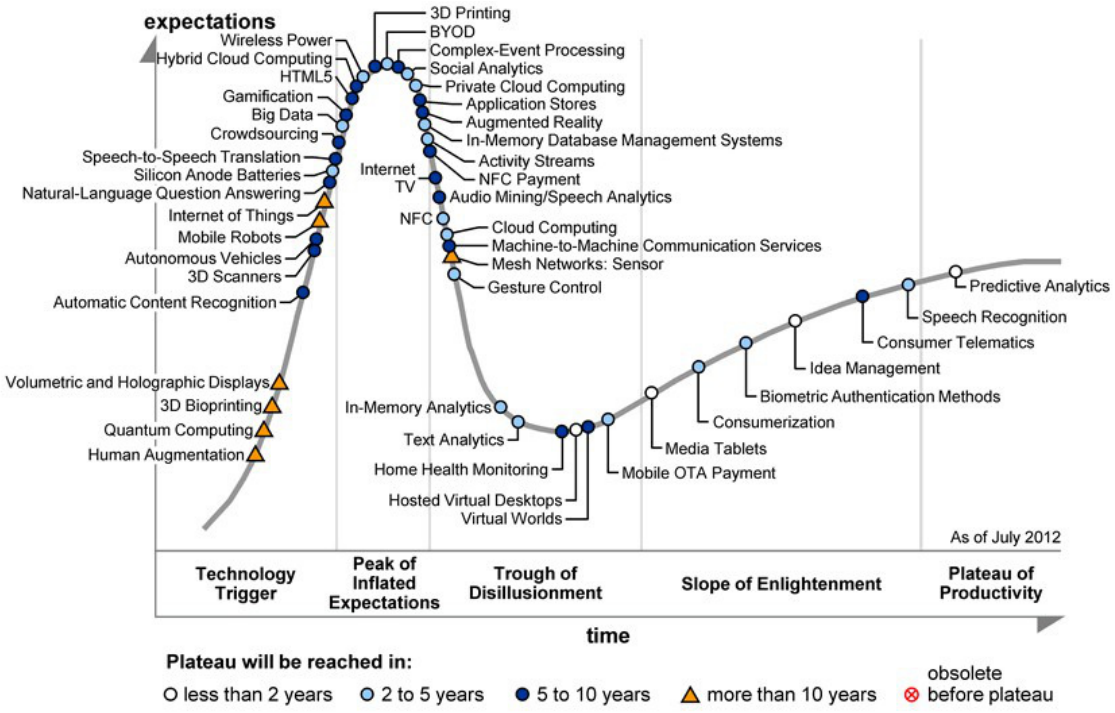

Figure 2.2: Gartner's Hype Curve for emerging technologies (July 2012)

2.6. Cloud Sustainability

During lasts years the cloud services demand has grown exponentially, and service providers started to keep into account energy issues, since the rising energy cost is a highly potential threat as it increases the Total Cost of Ownership (TCO) and reduces the Return on Investment (ROI) of Cloud infrastructures [17]. In order to lead the data center to a sustainable energy consumption, cloud infrastructures need an efficient management that not only meets the Quality of Service agreed with the customer, but cut on energy consumption.

Virtualization makes energy management a little easier: virtual instances are not tied to a specific physical machine and can be moved and grouped according to satisfy the demand. Naturally, it is necessary to found an algorithm that optimizes the energy/performance ratio, and then, implement a system that would be able to self configure according to this algorithm. This is because not all physical machines need to remain on.

It is important to understand which factors are dominant in the energy consumption of a data center that houses a cloud. Some authors [18], describing consumption models in cloud computing, argued that in some cases cloud computing savings are minimal, due to the considerable increase of network traffic. In other cases, where

network traffic is not so relevant (e.g. slow frame rate applications in public clouds or services in a private on-premise cloud) energy savings are substantial. As [18] claims, "The number of users per server is the most significant determinant of the energy efficiency of a cloud software service".

Looking at this assumption we infer that a resource manager can improve the current state of the technology.

2.7. Technologies

In this section I describe some existing cloud technologies, focusing on their main characteristics and architectures. Again, here I adopt a bottom-up approach, starting from the lower level of the service model stack (Infrastructure as a Service), and climbing up to reach to the highest one (Software as a Service).

2.7.1. Private IaaS

Eucalyptus

Eucalyptus (Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems) is an open-source cloud-computing framework [3] written in Java and C, which interface is based on the well known Amazon cloud services API. Eucalyptus has a modular and hierarchical design and supports most of currently available Linux distributions. As other IaaS technologies, Eucalyptus can interact with several hypervisor as XEN or KVM. The system takes advantage of modern infrastructure virtualization software to create elastic pools that can be dynamically scaled up or down depending on application workloads.

Architecture Eucalyptus consists of the following components:

- Cloud Controller: this component provides computational functionality, or rather it enables the deployment and management of virtual instances.
- Walrus: this component provides simple storage functionality. This module is equivalent to the Amazon Simple Storage Service (S3) which will be described in later paragraphs.
- Cluster Controller: this component provides management service for a cluster in your cloud.
- Storage Controller: this component provides Amazon Elastic Block Store functionality (section 2.7.2).

- Node Controller: this component controls virtual machine instances and their life-cycle. It runs on every node that is destined for hosting VM instances.

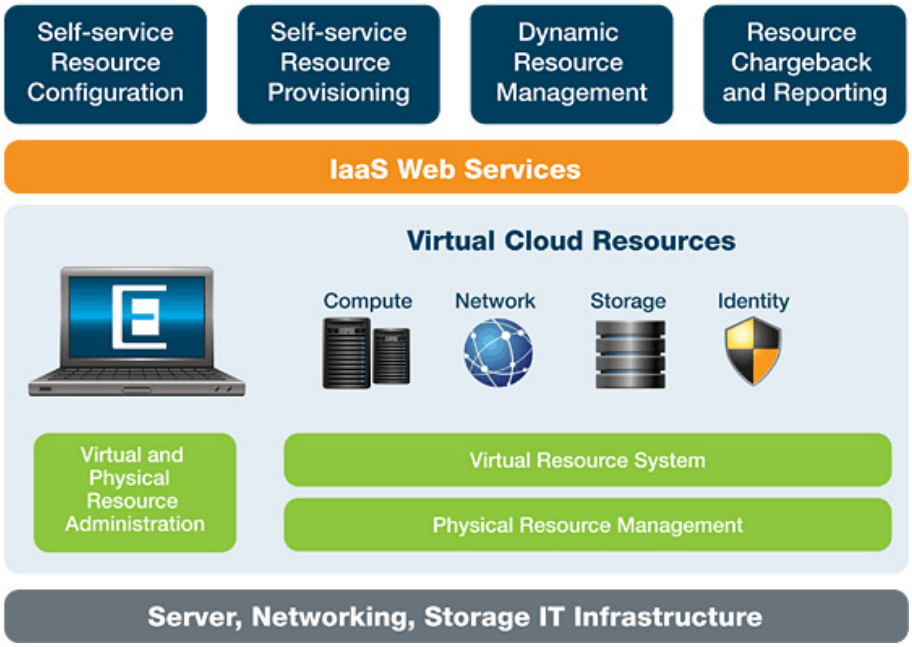

Figure 2.3: Eucalyptus structure (retrieved at [19])

Resource Administration Eucalyptus provides a public API, completely interoperable with Amazon's services: users can deploy instances in the proprietary on-premises infrastructure and then move it to the Amazon public cloud, using the same command set. This level of interoperability allows users to create Hybrid Clouds without management troubles.

The Eucalyptus Dashboard provides cloud administrators with a graphical console for performing several cloud management tasks including all virtual and physical resource management and virtual cloud resource configuration.

Nimbus

Nimbus is an open source EC2/S3-compatible Infrastructure-as-a-Service implementation specifically targeting features of interest to the scientific community such as support for proxy credentials, batch schedulers, best-effort allocations and others [20]. This is an open source tool set and also a cloud computing solution providing IaaS. Nimbus platform provides an integrated set of tools designed to overcome user needs: the *Phantom* service provides auto-scaling and high availability for collections

of resources. The *Context Broker* service is a service that allows clients to coordinate large virtual cluster. Finally the *cloudinit.d* toolkit is adopted for launching, controlling, and monitoring cloud applications. Nimbus supports XEN and KVM hypervisors. It also offers a special testing characteristic: the infrastructure could be run in "fake mode" where no virtual machines are ever started. This is very useful for testing the infrastructure topologies before the effective deployment.

OpenNebula

OpenNebula is a research project started as a management tool for the orchestration and configuration of virtual machines in data centers. The main aim of the project was to address challenges from business use cases in real-world conditions [9].

OpenNebula orchestrates the existing storage, networking, virtualization, monitoring, and security platforms. It provides a great compatibility that allows companies to continue to use their own infrastructure, updating offered services to the latest technology models.

OpenNebula can manage different user groups, separating projects that share the same infrastructure. The system provides a full control of the infrastructure: the user can deploy VM according to predefined VM templates, create and manage virtualized networks with traffic isolation.

OpenNebula supports XEN, KVM and VMWare hypervisors and supports different access interfaces including REST-based interfaces, OGF OCCI service interfaces, and the emerging cloud API standard, as well as the de-facto AWS EC2 API standard [21].

The system provides a Marketplace where users can choose virtual appliances and deploy their systems with just a few steps. The list of available instances offers CentOS, Debian, openSUSE and Ubuntu releases.

The most interesting characteristic of OpenNebula remains the interoperability with existing IT infrastructures, given by the modular and extensible architecture. The software is delivered under Apache license.

OpenStack

OpenStack is a private IaaS cloud computing system composed by a set of smaller projects, which are community maintained [10]. Each project concerns a different service, necessary to deploy the entire infrastructure.

Architecture Several parts make the Openstack system: each of them specialized in a specific role. The interaction between them could be appreciate in figure 2.4. The architecture is composed of six core blocks, each one presenting a code-name:

- Identity - Keystone

- Compute - Nova
- Image management - Glance
- Dashboard - Horizon
- Object Storage - Swift (Like Amazon S3)
- Volumes - Cinder (Like Amazon EBS)
- Networking - Quantum

Each service can interact with the others, and they are accessible trough a public API. Another important aspect is that Openstack's APIs are compatible with Amazon EC2 and Amazon S3 services, so clients written for the Openstack infrastructure can be easily used with the Amazon cloud [22].

Identity - Keystone is an OpenStack project that provides *Identity, Token, Catalog* and *Policy* services. It is implemented using a RESTfull interface and each connection to the API server should be created using secure HTTP (HTTPS). The authentication token validity lasts twenty-four hours and it is sent in all request messages against the API.

Compute - Nova service provides and manages large networks of virtual machines. It provides on-demand computing resources and it is able to work with different types of hypervisor technologies. its main features are:

- Management of virtualized commodity server resources (CPU, Memory, Network Interfaces)
- Management of Local Area Networks (If the network module - *Quantum* - is not present)
- A distributed and asynchronous architecture
- Virtual Machine management
- Volume management (If the volume module - *Cinder* - is not present)

Image Management - Glance provides discovery, registration, and delivery services for virtual disk images. It also provides an API with a REST interface, with which it is possible to retrieve information about the existing disk images, create new ones etc. Glance is compatible with different types of images, offering a good interaction with others systems (Raw, VHD, VDI, VMDK, OVF, qcow2 etc.).

Dashboard - Horizon provides a web user interface for managing Openstack services. With the web interface the user can easily access resources and manage them by a visual interface. All operations can also be performed through the Command Line Interface (CLI)

Object Store - Swift provides a fully distributed storage platform that can be used from applications. OpenStack provides redundant and scalable object storage using clusters of standardized servers, capable of storing petabytes of data. Swift is a distributed storage system for static data (as virtual machine images) that guarantees reliability and integrity by replication: data are stored in different disks all over the data center.

Network - Quantum is a project created in order to provide high level management over the network of the cloud. It enables the definition of the network connectivity and the addressing of virtual machines, allowing multi-tenant connections. Quantum can configure many network topologies, managing L2 and L3 configurations (L2 and L3 stands for Layer 2 and Layer 3 of the OSI model). Each tenant can have multiple networks with a private addressing that does not interfere with other tenant's addressing plans. Quantum service is used by other projects as Nova or Swift.

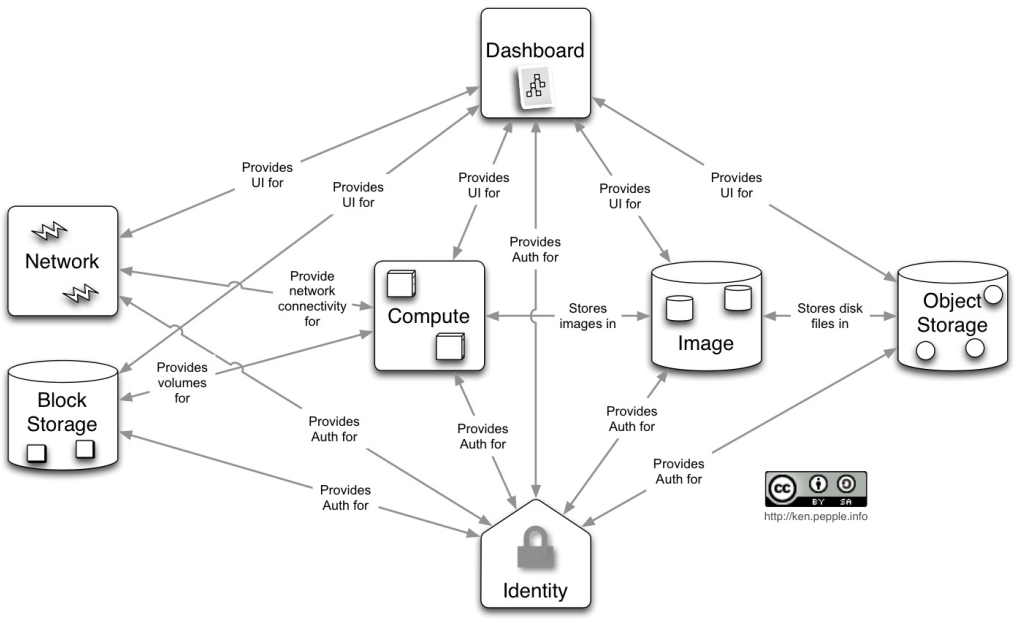

Figure 2.4: OpenStack conceptual architecture[1]

CloudStack

Apache CloudStack system does not stray too far from the other platforms of this type. It is an open source software designed to deploy and manage large networks of virtual machines as a highly scalable Infrastructure as a Service. Unlike OpenStack, CloudStack has a monolithic architecture where all the services are offered by a single cloud manager that run on a virtual machine itself. It controls the allocation of virtual machines to hosts and assigns storage and IP addresses to the virtual machine instances. The Management Server runs in a Tomcat container and requires a MySQL database for persistence. CloudStack supports XEN and KVM hypervisors and provides Network-as-a-Service, user and account management and a full and open native API compatible with AWS EC2 and S3. This last characteristic is paramount for the deployment of a Hybrid cloud infrastructure. In April 2012 CloudStack has been donated to the Apache Software Foundation, where it was accepted into the Apache Incubator.

Comparison of Open-source IaaS Cloud Frameworks

As we can see, the number of IaaS solutions is considerably wide. With all these different open-source cloud technologies, the decision to choose the most suitable one that meets our needs becomes a difficult task, because every platform has specific characteristics. Therefore I decided to base the decision taking into account some comparisons already presents in the literature [21, 23, 24] that point out the characteristics of these products, indicating which are the pros and cons of each one of them.

It is important to realize that some of these comparisons have been made a couple of years ago, and since them some of these technologies have experienced a substantial improvement; the offered services have increased, and many differences between these solutions are no longer valid.

On this basis, I compared the infrastructure services that provide a high level of customization, since we need the possibility of interacting with VMs at a very low level, controlling them with many degrees of freedom. For this reason I have reduced the array of private cloud solutions to the following three:

- OpenNebula
- OpenStack
- Eucalyptus

I have used to decide the following criteria:

[1]Retrieved on Jan-10-2013 at: “http://docs.openstack.org/folsom”

Software deployment An important feature is the ease of software deployments: the easiest to deploy is OpenNebula because we only have to install a single service in the main controller, and no OpenNebula software is installed in the compute nodes. On the other hand, the deployment of Eucalyptus and OpenStack is more difficult due to the number of different components to configure and the different configuration possibilities that they provide [21].

Security All of the IaaS frameworks support X.509 credentials as authentication method for users. OpenStack and OpenNebula also support authentication via LDAP, although it is quite basic [21]. OpenStack has developed a standalone project (*Keystone*) in charge with all security issues over the infrastructure, offering a centralized module that provides unified authentication across all projects.

Interfaces All solutions offer the same set of interfaces: a Open Cloud Computing Interface (OCCI) and the EC2 and S3 interfaces, all based on REST.

Networking The network is managed differently for each IaaS framework, providing various options in each of them. But analyzing the possibilities that each infrastructure provides [21], only OpenStack offers a standalone service that provides the possibility of creating several virtual network topologies. None of the others gives the user such level of control over the networking environment.

Virtualization All of the IaaS that listed before support KVM and XEN, which are the most popular open source hypervisors. OpenNebula and Openstack also support VMWare, while Eucalyptus only supports VMWare in its commercial version. Additionally, OpenStack also supports LXC and Microsoft's HyperV that provides integration and support of Windows Servers.

Scale and activity level of the community The last parameter I consider is the scale and the activity level of the community that lies behind each IaaS software. To evaluate this feature, I made use of a set of charts that provides a comparison between the aforementioned solutions, based on the total number of monthly new threads in the community forum and the monthly number of active participants contributing to the project [25].

In figures 2.5 and 2.6 we see that the OpenStack community is much more active and wide than the others. This feature implies a greater development and maintenance process that, in this open source scenario, is of paramount importance.

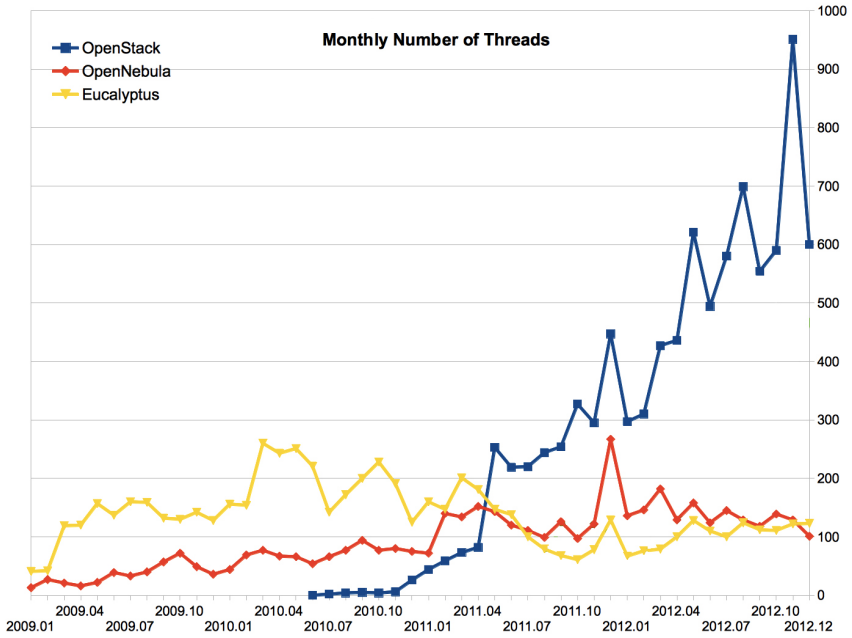

Figure 2.5: Monthly number of threads

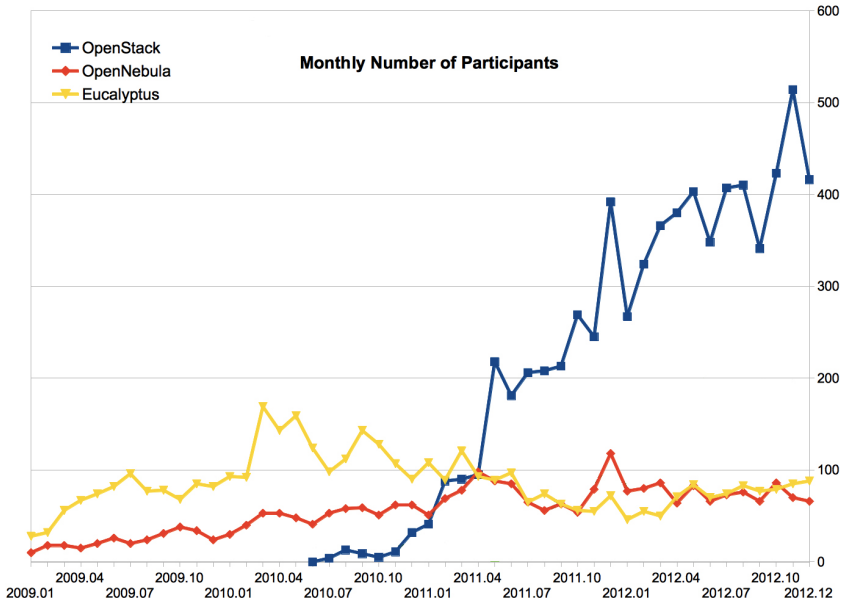

Figure 2.6: Monthly number of participants

Conclusion In light of this analysis, even if the aforementioned technologies are in many cases equivalents I chose to use the OpenStack IaaS. Although its deployment may be more difficult than OpenNebula, it offers a set of capabilities superior to the other solutions, the more important of all the advanced virtual network service. Moreover, using OpenStack I'm able to work with the support of a community that is in constant ferment, where bugs are resolved quickly and software versions are updated constantly.

2.7.2. Public IaaS

Amazon Web Services - AWS

Launched in 2002 is now one of the most important cloud systems in the market. It offers a complete set of infrastructure and application services that enable the customer to run virtually server instances in the cloud. The most well known Amazon's services are Amazon Elastic Computing Cloud (EC2) and Amazon Simple Storage Service (S3).

EC2 allows users to rent virtual computers on which they can run their own applications; in addition, users obtain the complete control of his computing resources with the possibility of managing them as they prefer. Amazon EC2 allows the customer to start instances from a preconfigured VM image or to create it with personal libraries, applications and data. The user chooses the instance type, selects a Virtual Appliance (configuration set regarding memory, CPU, instance storage, etc.) and obtains the root access to each VM.

The *Auto Scaling* service provides the automatic adaptation to the workload, avoiding a decrease of performances during bursts of demand and minimizing costs during periods of low usage. This service fits with applications that have very different utilization rates throughout the day. Amazon offers various payment types, from the on-demand model to the "Reserved Instances" model, specific for applications that require reserved capacity; the SLA provides 99.95% of availability over each Amazon EC2 zone.

Amazon Elastic Block Store offers persistent storage on the Amazon's infrastructure. Blocks are mounted through the network and are totally independent from the life-cycle of an instance; durability is provided by replication. Amazon EC2 works in conjunction with Amazon Simple Storage Service (Amazon S3), Amazon Relational Database Service (Amazon RDS), Amazon SimpleDB and Amazon Simple Queue Service (Amazon SQS) and offers interfaces accessible through HTTP, using REST and SOAP protocols or the CloudFront service.

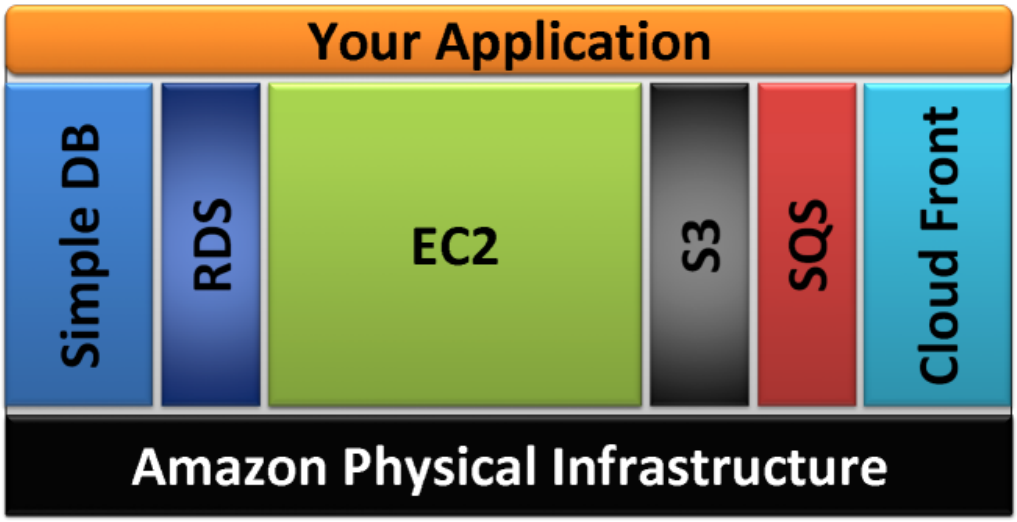

Figure 2.7: Amazon web services

Azure

Microsoft's Windows Azure was born as a PaaS product but in June 2012, Microsoft announced in preview at the Meet Windows Azure Event its new IaaS service. Azure supports different types of operating systems that can be retrieved from a default list provided by the system, or personally uploaded. The Microsoft's IaaS service offers built in monitoring for basic CPU, memory, network, and disk metrics, and a continuous geographic data replication across data centers [26]. Windows Azure also competes in the PaaS market.

Google Compute Engine - GCE

Google Compute Engine, announced at Google IO (June 29, 2012), is a *Infrastructure as a Service* product that permits to deploy virtual servers on Google's datacenters, the same infrastructure used by the company for its web-services like Gmail, Maps and Search. GCE has a number of advanced cloud computing features: user-level access to individual servers, persistent and attachable storage deployed in a specified Google data center/zone. First opinions state that GCE could start to weaken the Amazon's hegemony in the IaaS cloud services field. Being a new product GCE is in limited preview.

Rackspace Public Cloud

Rackspace Public Cloud is a cloud computing solution offered by Rackspace. This was one of the first commercial proposals among cloud computing services.

Rackspace Cloud Server service enables the deployment of hundreds of VMs, and supports several Linux distributions. It works with the XEN hypervisor and in 2010 has contributed to the source code of the OpenStack project, delivered under the Apache License. Recently (April 2012) Rackspace announced that it is going to include Openstack Compute service in its infrastructure.

Rackspace Public Cloud provides automatic scaling, on-demand load balancers, virtualized networks, periodic incremental backups, data encryption and high performance databases. It also offers a cloud monitoring service that sends alerts to the user (via mobile phone, laptop, etc.) when something is going wrong with servers, applications or configurations.

Other

Other solutions are present, like IBM Small Cloud Enterprise or GreenQloud, a company offering green cloud computing services powered by emission-free energy sources.

2.7.3. PaaS Services

Google AppEngine

Google AppEngine is a platform provided by Google that allows the user to run web applications on Google's infrastructure (Google Cloud), easing the management of the entire physical equipment, its scalability and the data storage. With Google App Engine users can develop an application and make it visible to the world, without problems of physical resources management. The scalability of the service is directly managed by the App Engine, which automatically distributes the load of incoming requests over the available resources.

Google App Engine supports several programming languages, like Java, Python, JavaScript, Ruby and Go. From an economical point of view, Google App Engine initially provides a totally free of charge access, naturally with limited storage capacity (1 GB) and limited number of accesses. If more computational power or resources are needed, the user can enable the paying service where only usage exceeding the free limit will be charged.

Google App Engine provides a dynamic web server system with persistent storage that presents different data warehousing strategies and access typologies:

- App Engine Datastore: a NoSQL distributed storage that works with a no-relational query and transactions engine.
- Google Cloud SQL: a relational storage offering SQL access, based on a distributed Relational Database Management System, which provide high data availability.

- Google Cloud Storage: a Large Objects Storage Service that combines the performance and scalability of Google's cloud (distributed replication, access control, etc.)

In addition to this, Google App Engine provides a cache memory service (Memcache) accessible through a Java interface, which increases the performance and reduces the resource usage of web applications.

Heroku

Heroku is one of the first cloud PaaS to appear. Launched in 2007, it supports several programming languages like Ruby on Rails, Java, Phyton, Clojure etc. The adoption of Git [27], a free revision control software makes the deployment of new applications incredibly easy: with very few commands the application can be deployed on Heroku's infrastructure. The platform provides interoperability with third party services, like databases, email, management, search etc.

The application management is performed through a command-line interface, a web console or a RESTfull API. Heroku also provides a release management service that enables the user to safely make changes to the developed code and, if needed, to roll back to a previous version.

The application status is continuously logged, so the customer can follow the behavior of his application during its lifecycle: all operations performed in each part of the platform are recorded giving the user full visibility of the system.

The platform provides an auto scale capability that increases automatically the amount of resources (Dynos) assigned to the application, using a dynamic routing service balances automatically the web traffic.

2.7.4. SaaS Services

Storage

There are a lot of companies that focus their business on a platform that stores files in the cloud; there are many examples like the famous Dropbox [28], which are able to work either with mobile devices or traditional computers. A relatively new entry in this field is Google Drive, which tries to compete by integrating it with its other cloud services. SkyDrive is the Microsoft's answer, offering storage service integrated with all the products of Windows Live. For all solutions, data is available trough web navigation with a web browser, but in some cases the company also produces PC or Smartphone software providing a full synchronization with the cloud.

Office

The online text editor service is another type of cloud service that has seen a rapid growth in recent years. This service frees the user from the issues of portability and access, very common in classic editing programs. Google and Microsoft has begun to offer this kind of services with their products Google Docs and Office Live Workspace, enjoying great success. In this category we can also found the Alfresco cloud enterprise content management system.

Pictures elaboration

Also, in the image editing scenario a lot of cloud services are rising up: an example is Photoshop Express, the web cloud version of the most famous image editing software. This version saves the occasional customers from the huge cost and big size of the full release.

Security

In the security field there are some innovative proposals of cloud anti-virus. One example is the Panda Cloud Antivirus [29] where the file analysis is run on the cloud instead of the local PC like all classic anti-virus software products.

Games

Cloud computing has started to catch on the world of videogames; a typical example is the Gaikai [15] platform, where the game is run in the cloud enabling low capacity computers to run new generation games that usually need an high level of resources. Gaikai has been acquired by Sony and the future PlayStation 4 will feature cloud gaming.

2.8. Management of cloud services

With the expansion of the sector, automated cloud controllers have become of paramount importance, mostly when the private cloud infrastructure experiences a rapid growth. In contrast to the elasticity and flexibility of cloud services the traditional management methods and tools seem inappropriate because they usually require local software installation with continuous updates and patches [30].

All private IaaS solutions offer their own management interfaces, tailored to its specific needs and features, and are rarely able to interact with other solutions, or even other cloud deployments of the same solution. The only exceptions to this fact are not by design: it is just that some private IaaS solutions try to replicate the same

capabilities and abstractions offered by more popular public offerings, like Amazon AWS. And, in doing that, they develop very similar or even identical interfaces.

The existing management tools for cloud infrastructure and storage services can be classified as follows [31]:

- Command-line tools.
- Locally installed management applications with a graphical user interface.
- Firefox browser extensions.
- Online tools.

These management systems are usually proprietary solutions that work only with a specific cloud service, and are not compatible with other service providers [32]. In the next paragraphs I will focus on some open source solutions that can be found in the cloud computing scenario.

2.8.1. KOALA

KOALA (Karlsruhe Open Application (for) cLoud Administration) is a web based application able to manage and control AWS compatible cloud services [30]. It allows the user to work with a large variety of services of various public and private cloud providers in a seamless and transparent way [31]. KOALA is an open source project that presents an innovative characteristic: in contrast to the majority of management systems, it does not require a local installation since itself could be deployed in the cloud, on a scalable platform service such as Google App Engine. The user interface allows customers to start, stop and monitor their instances and volumes in various cloud infrastructure regions, and to have access to the console output of virtual machines. KOALA supports S3, Google Storage and Walrus storage services.

2.8.2. Scalr

Scalr is a proprietary, cross platform cloud management software that provides auto scaling, disaster recovery and server management. It is open source, available at Google Code[2] but a hosted version is available as paid service. The manager is able to scale the virtual infrastructure according to the load. Scaling strategies could be based on CPU, RAM, disk, network or date. This last can be useful in the case an increase of traffic is expected as during scheduled public events. The code is distributed under Apache 2 license.

[2]http://code.google.com/p/scalr/

2.8.3. Puppet

Puppet is an IT automation software that helps system administrators manage infrastructure throughout its lifecycle, easing the automation of the repetitive tasks [33]. This configuration management tool is written in Ruby and provides some specific modules for cloud management. The software is distributed for free with some utilization restrictions. The paid version offers a solution without limits.

2.9. Virtualization

2.9.1. Hypervisors

A hypervisor, also called Virtual Machine Manager, is a software that allows to run and orchestrate many operating systems (OS), that usually are denoted "guest", on the same hardware, usually denoted as "host". It is responsible for the management of virtual resources, which are distributed among the running guests. There are two main types of hypervisors (figure 2.8):

- Type 1 (or native): this kind of hypervisors runs directly on the hardware, without the mediation of a host operating system.
- Type 2 (or hosted): this kind of hypervisors runs over a conventional operating system.

The two mostly widely used hypervisors are XEN and KVM which is included in Linux distributions since the 2.6.20 version.

XEN

Xen is a open source software developed by Cambridge University and licensed under the GNU General Public License (GPLv2). It belongs to the first category of hypervisors and is responsible of managing CPU, memory and interrupts [34]. It supports two virtualization modes: paravirtualization or full virtualization (also called Hardware-assisted virtualization). The management service is provided trough a special Virtual Machine that has privileges in accessing the hardware configuration: it offers an interface to the exterior, with which the user can manage the life cycle of virtual instances.

KVM - Kernel-based Virtual Machines

KVM is a native hypervisor that provides virtualization solutions; it is a open source software distributed as a module included in Linux mainline. In contrast

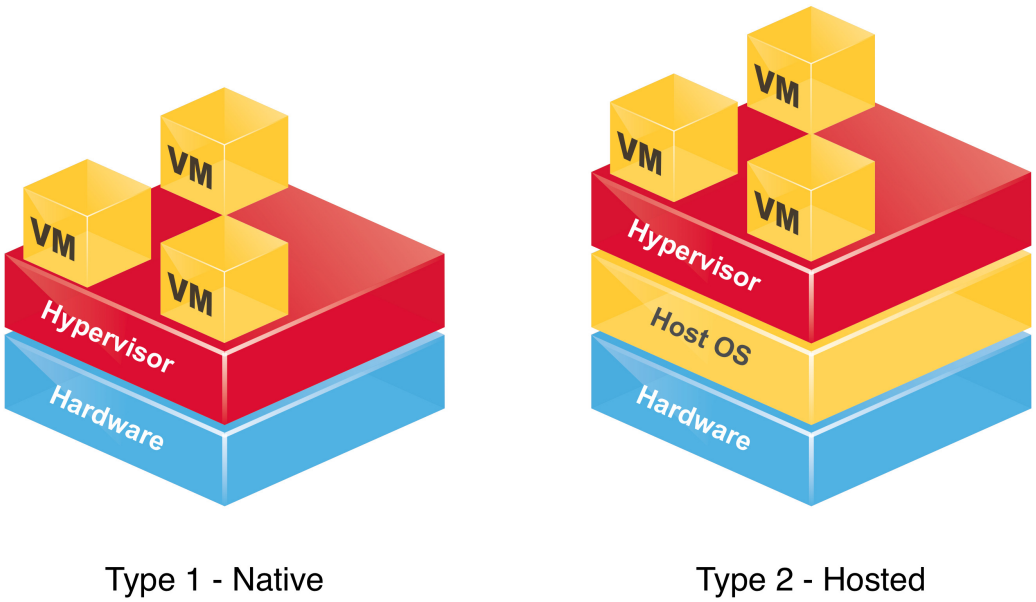

Figure 2.8: Types of Hypervisors

with XEN, KVM it is not an external software, but a component of Linux that uses the regular Linux scheduler. Like XEN it supports several guest operating system including Ubuntu releases and Windows.

2.9.2. LibVirt

LibVirt (VIRTualization LIBrary) is a toolkit used to interact with the virtualization capabilities of recent versions of Linux [35]. It is a free software, providing a rich API that allows the user to manage and perform actions over hypervisors. It allows to manage remote hypervisors using a special daemon called libvirtd that runs on remote nodes. This daemon is started automatically when libvirt is installed on a new node and can automatically determine the local virtual machine manager and set up drivers for it [36]. This tool can interact with a wide variety of hypervisors as Linux KVM, Xen or VMware.

2.9.3. Other solutions

In the virtualization environment other solutions are present: the most known are VMWare an VirtualBox. The former is developed by a company that provides software solutions for cloud and virtualization services. Some versions of the software are distributed free of charge (VMWare Player), while the enterprise product involve

a fee; it is compatible with most of classic operating systems (Linux, Ma OS X, Windows).

VirtualBox (formally Oracle VM VirtualBox) is a virtualization software created by the German company *Innotek GmbH* and then purchased by *Sun Microsystems*. The core package of the software is distributed under the GPLv2[3] license, while the *VirtualBox Oracle VM VirtualBox extension pack* is a proprietary solution, distributed for free under the PUEL License[4]. Even in this case, this technology is able to run several operating systems over a single host machine.

2.10. Supporting technologies

2.10.1. Eclipse Modeling Framework

The EMF project is a modeling framework and code generation facility, for building tools and other applications, based on a structured data model [37]. EMF allows the automatic creation of a set of Java classes starting from a XML specification that defines such model. XML is not the only supported language: EMF can read, analyze and convert diagrams expressed in UML or other languages specific for model design.

2.10.2. GitHub

GitHub is an web based hosting service for the management of software projects throughout the *Git* control version protocol. It enables the user to store his code, which remains publicly available through the internet. The web page offers social networking functionality and includes two types of accounts: the former is a totally free solution that allows users to store code only publicly. The latter is kind of premium account which provides the ability to save the code privately (not accessible publicly). This feature is only available after paying a subscription fee.

2.10.3. Launchpad

Launchpad is a web application maintained by Canonical Ltd. that supports software developing. In July 2009 the source was released under GNU Affero General Public License[5]. The web page is divided into several sections: some parts are

[3]Available at "http://www.gnu.org/licenses/gpl-2.0.html"

[4]PUEL - Personal Use and Evaluation License available at "https://www.virtualbox.org/wiki/VirtualBox_PUEL"

[5]Avalable at http://www.gnu.org/licenses/agpl-3.0.html

specifically dedicated to developers' questions, while others are related to the bug notification process. Developers or users can join the community to share their experience, problems and solutions. A specific section is dedicated to bug tracking, where users can contribute to resolve software's matters using the code hosting service, managed through the GNU Bazaar control version system.

2.10.4. REST

The *REpresentational State Transfer* is a type of software architecture introduced in 2000 by Roy Fielding in his Ph.D thesis [38]. REST refers to a set of assumption that had been fundamentals in the evolution of the Internet. This architectural style consists of clients and servers that interchange messages describing addressable resources and its state. REST ease the interaction between services decreasing the coupling between them.

Chapter 3

Requirements Analysis

In this chapter I analyze in detail all aspects of the software that I am going to develop. I will review all functions that the *Infrastructure Manager* (hereafter IM) have to comply to to create a comprehensive management module. After having introduced the main topic and the working scenario I will describe in specific terms the problem and the main objectives of the project.

3.1. Problem Definition

As described in the state of the art chapter, the cloud computing scenario is populated by several private IaaS solutions. However, this wide offer of private IaaS cloud technologies also involves an important drawback: each one is managed using different abstractions (sometimes for the same concepts) and through different management interfaces.

This fact presents problems for a more widespread adoption of private IaaS cloud computing, since potential users fear of being locked-in with a particular solution that falls behind the others in terms of features or support. One should be able to change its previously chosen technology for private IaaS without having to modify the management interfaces, a fact that sometimes incurs expensive retraining and even more expensive errors during production deployments.

Moreover, an enterprising private IaaS user could have the desire of deploying two or more different cloud offerings, leveraging the strong features of each for a solution better tailored to his specific needs. In this situation the user would benefit greatly from an integrated management interface that could wrap this mixture of products into a uniform whole.

With this problem in mind, I proceed to design a management system for a private IaaS cloud, OpenStack, providing the possibility of being expanded in order to create a general management interface. In particular I implement the system for

a specific cloud solution, leaving the integration with the others as a future line of work.

This management system should provide the possibility of offering its interface through different implementations (like a REST web service, a command line, or a web page), to better suit the user's needs. Despite this, the greatest effort of the work lies in the definition of the general architecture and the development of the main structure, the heart of the system.

So, with this premise, I can state that the main aim of the *Infrastructure Manager* is the deployment and management of virtual machines over a private IaaS solution, regardless of the adopted technology. More precisely, the main target is to create a system that provides an extensible operational interface, which allows the user to interact with the cloud infrastructure throughout a set of primitive functions. It should emulate what actually is done by a human administrator, managing manually all occurrences over different infrastructures.

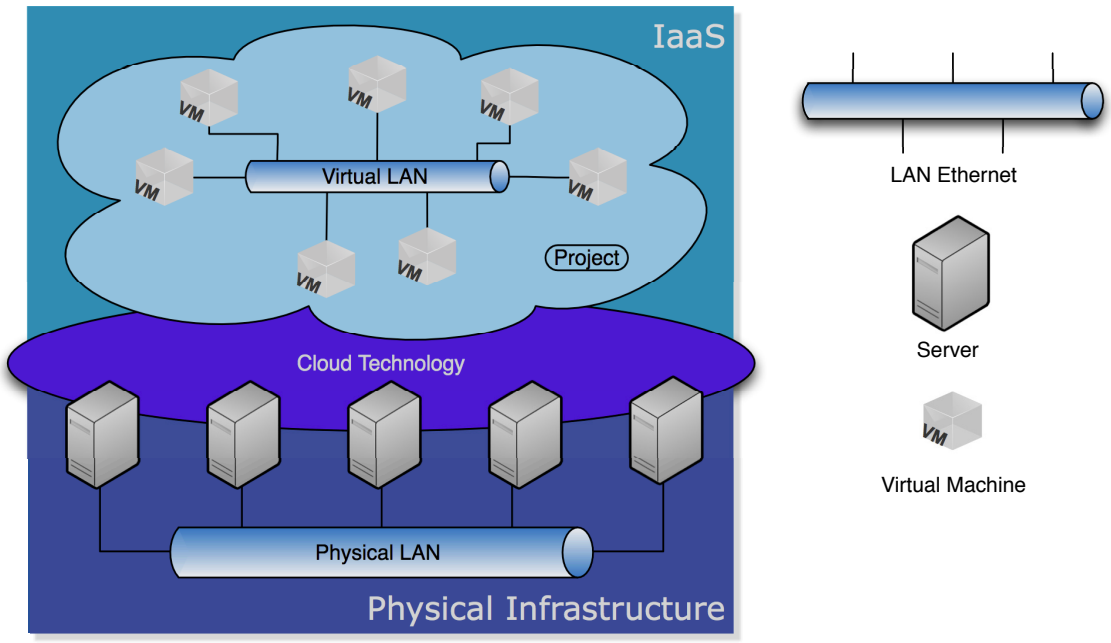

Figure 3.1: Cloud Computing scenario

3.2. Domain model

A good way to describe an environment is to start analyzing each element from which it is composed. This section tries to define a common vocabulary of terms in order to avoid misunderstandings between similar concepts used in the scenario. In figure 3.2 we can see the graphical representation of all elements listed hereafter.

Host refers to a physical machine, running an operating system, which holds a Hypervisor. It can host one or more *Virtual Machines*.

Virtual Machine hereafter VM, is a running virtual computer. It is defined by a set of virtual resources and could be seen as a virtualized hardware where its specifications are defined as a set of virtual resources. Its execution is managed by the hypervisor.

Virtual Instance refers to a software program running on a VM. It is deployed over a VM. It could also be called *Server*.

Virtual Appliance hereafter VA, is a minimum configuration set that defines a list of virtual resources (as RAM, Hard Disk capacity, CPU...) needed to the VI

Hypervisor also called Virtual Machine Manager is a software, installed on a *Host*, that presents to the Virtual Machines a virtual operating platform and manages their execution.

Virtual CPU virtual resource, or rather a virtualized Central Processing Unit assigned to a Virtual Machine. More than one Virtual CPU can be assigned to each VM, according to its definition.

Virtual Memory virtual resource, or rather a virtualized random-access memory (RAM) assigned to a VM.

Virtual Storage Unit virtual resource, or rather a virtualized volume of persistent memory, which can be attached to a VM.

Virtual Network Interface virtualized resource that simulates the connection to a Local Area Network.

Image is a single file containing a deployable operating system. The format can vary depending on the virtualization technology that is adopted.

Cloud Controller host running the IaaS Service, which manages the main cloud platform.

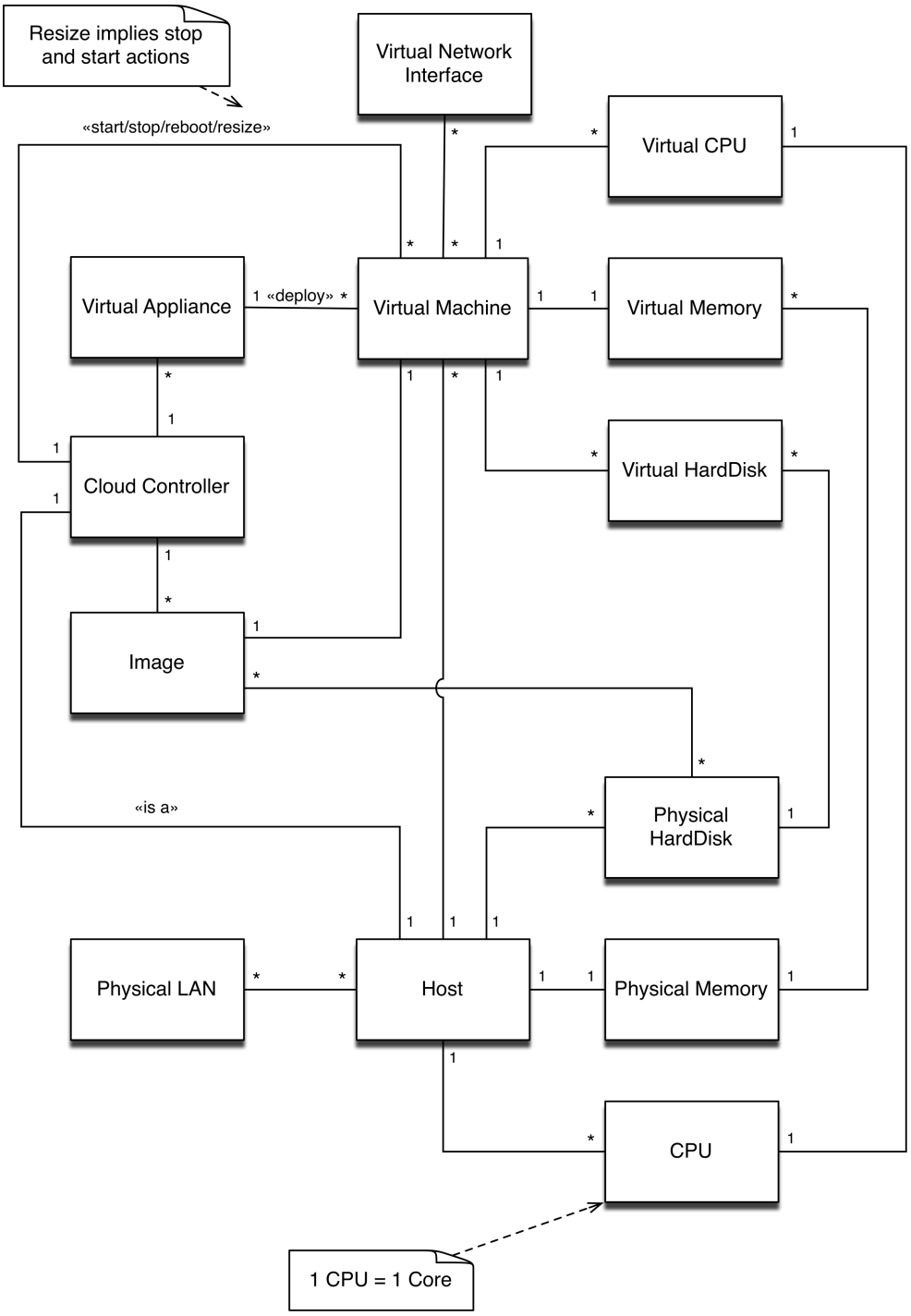

Figure 3.2: Schematic representation of the domain model

Infrastructure Manager is the system that interacts with the IaaS platform in order to manage the infrastructure of virtual machines. The creation of this module corresponds to the aim of this project

Migration displacement of a *Virtual Machine* from a host to another, turning off the instance.

Resize deployment of a *Virtual Machine* with another set of virtual appliance.

Physical Storage Unit secondary storage of a host, usually expressed in gigabytes.

Physical Memory main memory of a host (RAM) usually expressed in megabytes.

CPU the term defines the elemental physical processing unit; more precisely I intend a single core of a multicore processor.

User the person or system that interact with the Infrastructure Manager.

3.3. Requirements

The analysis phase includes the study and the definition of the system requirements. In this section I expose which are the sub-objectives of the IM and I itemize the set of functions our software will provide. Requirements define in detail all services the system should offer, identifying its restrictions and characteristics. I will distinguish between two types of requirements:

- Functional requirements: they define all functions that the system has to provide to the user, specifying which are inputs and outputs of each task.
- Non-Functional requirements: all limitations that affect the system behavior. In this part I will include all development restrictions and the use of specific standards.

Each requirement can be flagged with two different priorities:

- Mandatory: essential characteristics without which the system could not be defined complete.
- Optional: noncompulsory characteristic that enriches and adds value to the software.

In the next section I will list all requirements following a standard tabular format; each requirement is identified by an unique ID - FRXX where XX stands for the number of the list.

3.3.1. Functional Requirements

FR01	Start/Stop a Virtual Machine
Description	
The system must be able to start and stop *Virtual Machines* according to a *Virtual Appliance* definition. In the creation phase, it should retrieve the information about the available virtual resources set; if any suitable is present, it should create a feasible one that accomplishes user needs. Moreover, the system should control the presence of available images and respond to the user with the proper notification. On stopping phase, the system must inform the user with the final status, or any problem regarding the requested action. The user should be able to provide as optional value the *Host* name over which the VM will be instantiated.	
Priority	Mandatory

Table 3.1: Functional requirement 01

FR02	Retrieve the Virtual Machine list
Description	
The system must provide the list of all running *Virtual Machines*, specifying the detailed information as identifier and status (running, stopped, booting etc.). In this way the system is aware of the actual situation.	
Priority	Mandatory

Table 3.2: Functional requirement 02

FR03	Resize a Virtual Machine
Description	
The system must be capable to resize a running VM, or rather to change the resource set associated to a running VM. The user has to enter the new parameters and the *Infrastructure Manager* must control if the running VM fits the targeted resources set and, in the affirmative case, proceed with the operation. In the event of problems, the system must inform the user that the operation cannot be performed.	
Priority	Mandatory

Table 3.3: Functional requirement 03

FR04	Create/Delete a Virtual Appliance
Description	
The system must provide the possibility of create a new *Virtual Appliance* definition. It have to check the validity of the inserted parameters (resources set, Image, etc.) and store it in a reliable and consistent mode. On the other hand, also the possibility of deleting an existing VA must be provided.	
Priority	Mandatory

Table 3.4: Functional requirement 04

FR05	Retrieve the Virtual Appliance list
Description	
In order to provide consistent information, the system must return the list of all available VAs. With this, the user can obtain a actualized view of the environment and could use the retrieved information with other functionality as the FR01	
Priority	Mandatory

Table 3.5: Functional requirement 05

FR06	List all/active hosts
Description	
The system must provide detailed information about how many and which are the active and inactive hosts underlying the cloud infrastructure.	
Priority	Optional

Table 3.6: Functional requirement 06

FR07	Suspend/Restart a Virtual Machine
Description	
The system must provide the capability of suspending and restarting Virtual Machines.	
Priority	Mandatory

Table 3.7: Functional requirement 07

FR08	Create/Delete a Virtual Network
Description	
The Infrastructure Manager must be able to create a new Virtual Network. The user will provide the network address and the network mask, associated to this instance. The system must control the availability of addresses and return the id of the created element.	
Priority	Mandatory

Table 3.8: Functional requirement 08

FR09	Retrieve Virtual Network information
Description	
The information about the available Virtual Networks must be retrievable. For each virtual network definition, the IM should show net address and mask.	
Priority	Mandatory

Table 3.9: Functional requirement 09

FR10	Create/Delete SSH key pairs
Description	
The system must be able to create and store SSH key pairs in order to permit a secure access to virtual instances.	
Priority	Mandatory

Table 3.10: Functional requirement 10

FR11	Automated boot of instances
Description	
The system must be able to start a cloud elements starting from a ad hoc configuration file. This must be compatible with other technologies of this type as the *Virtual Networks over linuX* technology[39].	
Priority	Optional

Table 3.11: Functional requirement 11

FR12	Authentication
Description	
The systems must authenticate against the cloud technology, in order to perform operations on the infrastructure. The authentication information should be stored as a session, avoiding continuous authentication requests.	
Priority	Mandatory

Table 3.12: Functional requirement 12

3.3.2. Non-functional Requirements

NFR01	Support to different private cloud infrastructures
Description	
The IM should be implemented in order to permit an easy adaptation with most of cloud infrastructures.	
Priority	Mandatory

Table 3.13: Non-functional requirement 01

NFR02	Extensible structure
Description	
The software mus follow a modular architecture in order to permit an easy extension of the code.	
Priority	Optional

Table 3.14: Non-functional requirement 02

NFR03	Integration in the main project
Description	
The project must follow all conventions established by the research group in order to permit the integration in the main project.	
Priority	Mandatory

Table 3.15: Non-functional requirement 03

NFR04	Java implementation
Description	
The software must be implemented in the Java language, using the Java Developers Kit 1.6 or higher.	
Priority	Mandatory

Table 3.16: Non-functional requirement 04

NFR05	Easiness of use
Description	
The system must be easy to use, so that simplifies the ordinary sequence of operations that should be performed manually.	
Priority	Mandatory

Table 3.17: Non-functional requirement 05

3.4. Use cases

With the aim to describe in the best way all characteristics of our system, I will discuss some situations in which the *Infrastructure Manager* could be encountered. Before detailing each use case, it is important to define the *Use Case* concept and which are the connections between it and the user.

A use case is a list of steps, typically defining interactions between a role (known in UML [40] as actor) and a system, to achieve a specific goal. This is a technique usually used in software engineering processes that permits to retrieve the system requirements in a exhaustive way in order to obtain a quality developed software.

The use case section identifies and describes the basic scenarios of use of the system and the actors that interface with it. A use case must be elemental, or better non decomposable into simpler use cases that have still complete sense for the involved actors. Use cases will be represented with UML as shown in the next figure.

Figure 3.3: Use Case

Some use cases could be linked between them in different ways. In a use case diagram there are two ways of defining these relationships. The former one is the *inclusion*: this relation is specified when a particular use case is carried out employing another one. The latter one is the *extension*, when a use case adds a a further functionality to another one.

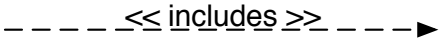

Figure 3.4: Inclusion

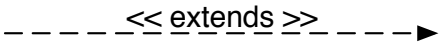

Figure 3.5: Extension

The last main element of a use case UML diagram is the Actor. An Actor is an external agent that starts a use case; it could be a person (usually with a specific role) or a automatic module (computer, intelligent agent, etc.).

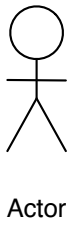

Figure 3.6: Actor

In our specific system, the actor is a simple user that could be represented by a person, as a system administrator, or an automatic element, as a reasoning module or an intelligent controller. The user can interact with the *Infrastructure Manager* and start the defined use cases.

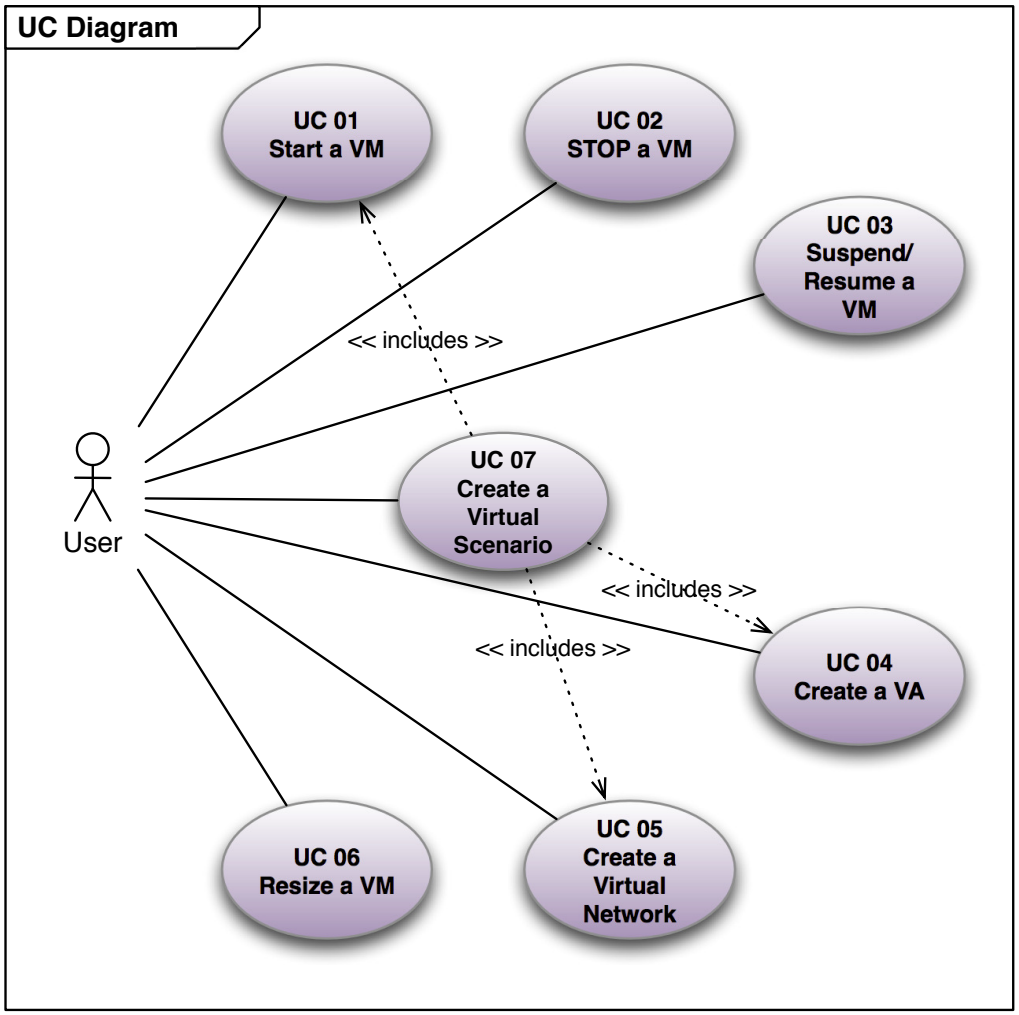

Figure 3.7: Use case diagram

In the next part I analyze in a detailed way the use cases defined in the UML schema represented in figure 3.7. Each use case description follows a standard layout that specifies, preconditions, actions that the system must perform, post-conditions and the exception list, or rather the index of possible impediments that prevent the success of the transaction. Each Use Case will be referred with a standard notation: UCXX where XX stands for the use case number.

UC01	Start a Virtual Machine from a Virtual Appliance definition	
Description	The system starts a new *Virtual Machine* according to the user specifications (*Virtual Appliance, Network, Host,* etc.) and returns an information set that allows the user to reach the new running instance.	
Preconditions	The system must be authenticated with the *Cloud Controller* in order to be able to perform all requested operations. Furthermore, in the system must be defined at least an available Virtual Appliance, where its parameters are coherent with the cloud technology (see UC04).	
Sequence	Step	Action
	1	The system checks the availability of the selected Virtual Appliance.
	2	If specified, it checks the availability of the selected Host.
	3	It checks the availability of the selected Network.
	4	The system boot the *Virtual Machine* and check the running state.
	5	The user receives the credentials and the IP address of the *Virtual Machine*.
Post-conditions	The new VM is running and the user earns the credential that permits to access the virtual instance.	
Exceptions	Case	Condition and action
	1	If the VA is not present, the system returns an error message.
	2	If the Network is not present, the system returns an error message.
	3	If the *Host* is not present, the system returns an error message.
Comments	If no *Host* is specified, the physical position of the new *Virtual Machine* will be chosen by the *Cloud Controller*	

Table 3.18: Use case 01

UC02	Stop a Virtual Machine
Description	The system stops the *Virtual Machine* selected by the user
Preconditions	The system must be authenticated with the *Cloud Controller* in order to be able to perform all requested operations. The selected *Virtual Machine* must exist.

Sequence	Step	Action
	1	The system checks if the selected VM is running.
	2	It shutdowns the VM.
Post-conditions	The selected VM is off.	
Exceptions	Case	Condition and action
	1	If the VM is not present, the system returns an error message.
	2	If the VM can not be switched off, the system returns an error message.
Comments	None	

Table 3.19: Use case 02

UC03	Suspend/Resume a Virtual Machine	
Description	The system suspends/resumes a Virtual Machine selected by the user	
Preconditions	The system must be authenticated with the *Cloud Controller* in order to be able to perform all requested operations. The selected *Virtual Machine* must be running.	
Sequence	Step	Action
	1	The system checks if the selected VM is running/suspended.
	2	It suspends/resumes the VM.
	3	The user receives the final status of the VM.
Post-conditions	The selected VM is in the suspended/active state.	
Exceptions	Case	Condition and action
	1	If the VM is not present, the system returns an error message.
	2	If the VM can not be suspended/resumed, the system returns an error message.
Comments	Physical resources attached to a suspended VM must be freed.	

Table 3.20: Use case 03

UC04	Create a Virtual Appliance
Description	The system creates a new *Virtual Appliance* and store it in a reliable database. The whole stored information must be coherent with the system.

Preconditions	The system must be authenticated with the *Cloud Controller* in order to be able to perform all requested operations.	
Sequence	Step	Action
	1	The system checks if an equivalent VA is already present.
	2	It checks the validity of parameters passed by the user (*virtual resources*, *Image*, etc.).
	3	The user receives the ID of the new VA.
Post-conditions	The selected VA is created.	
Exceptions	Case	Condition and action
	1	If the VA is already present, the system returns its ID.
	2	If parameters are not valid, the system rise an exception.
Comments	None	

Table 3.21: Use case 04

UC05	Create a Virtual Network	
Description	The system creates a new *Virtual Network* on the cloud infrastructure.	
Preconditions	The system must be authenticated with the *Cloud Controller* in order to be able to perform all requested operations.	
Sequence	Step	Action
	1	The system checks if an equivalent Network is already present.
	2	It checks the validity of parameters passed by the user.
	3	The user receives the ID of the new network.
Post-conditions	The selected network is created.	
Exceptions	Case	Condition and action
	1	If the network is already present, the system returns its ID.
	2	If parameters are not valid, the system rise an exception.
Comments	None	

Table 3.22: Use case 05

UC06	Change virtual resources to a specific VM	
Description	The system manages the set of virtual resources that had been attached to the *Virtual Machine*. According to the user needs it changes this set with a new one without altering the parameters that do not refers to this aspect.	
Preconditions	The system must be authenticated with the *Cloud Controller* in order to be able to perform all requested operations. The selected *Virtual Machine* must be running.	
Sequence	Step	Action
	1	The system checks if the selected VM is running.
	2	It checks if the virtual resources set exists.
	3	If it does not exists, the system create a new one
	4	The VM is shut down and the resources set changed
	5	The VM is reactivated with the new configuration.
Post-conditions	The selected VM is running with the new set of virtual resources.	
Exceptions	Case	Condition and action
	1	If the VM is not present, the system returns an error message.
	2	If the VM can not be resized, the system returns an error message.
Comments	Credentials used by the user to accede the VM must not change.	

Table 3.23: Use case 06

UC07	Create a complete virtual scenario	
Description	The system creates a compete scenario with *Virtual Machines* and *networks* defined by the user by a configuration file.	
Preconditions	The system must be authenticated with the *Cloud Controller* in order to be able to perform all requested operations.	
Sequence	Step	Action
	1	The manager parse the configuration file.
	2	It checks the validity of introduced values and creates the VA definitions
	3	The IM create the networks according to addresses and net-mask defined into the configuration file.

	4	A The list of active servers and networks is returned.
Post-conditions	The cloud elements defined in the configuration file are active	
Exceptions	Case	Condition and action
	1	If parameters are not valid, the system rise an exception.
Comments	Optionally this function could be exported as a REST service.	

Table 3.24: Use case 07

3.4.1. Traceability

The relationships between use cases and requirements can be maintained through a tracking matrix, which shows the mapping between use cases and functional requirements. In general, each functional requirement should be reflected in at least one use case, and no one would involve functional requirements not present in the requirements section or contradict them.

In the tracking matrix we can see how many functional requirements are involved in each use case. The last FR is involved in all UCs since it refers to the authentication phase, an essential step that is mandatory in order to perform any kind of operation over the cloud.

The last use case also involves a lot of functional requirements, since the creation of a complete virtual scenario is a process that needs a wide set of capabilities.

	UC01	UC02	UC03	UC04	UC05	UC06	UC07
FR01	X	X				X	X
FR02		X	X			X	
FR03						X	
FR04				X			X
FR05	X			X			X
FR06	X		X				
FR07			X				
FR08					X		X
FR09	X				X		X
FR10	X						X
FR11							X
FR12	X	X	X	X	X	X	X

Table 3.25: Traceability matrix mapping Functional Requirements to Use Cases

Chapter 4

Design

After defining the domain model and the functional aspects that the infrastructure manager should provide, I design a software architecture that accomplishes with these preliminary conditions. In order to ease the comprehension I will accompany the description with graphical representations, following the UML standards.

In this chapter I will describe the adopted solution, moving from the software architecture to the description of each functional element.

4.1. General architecture

I will start to analyze the architecture from an high level description and drop down up to a detailed overview of each specific component. In figure 4.1 we can see how the system can be divided in three main layers:

- A top layer denominated *Control Layer* that represents the main controller module. This component make use of services provided by underlying levels; on the other hand it also provides services to the outer world.
- A medium layer, called *Manager Layer* that represents the heart of the whole system. This part is formed by a set of modules, each of which is charged to manage a specific functional area.
- A lower layer, the *Client Layer* the plug of the software to the cloud technology, which behaves as server.

The architecture of the system will follow the Roman concept of *"Divide et impera"* (divide and rule). The aim of this technique is to split the main problem into smaller ones, whose solution results more feasible. In this optic, I have designed some components (managers), which are charged for specific functional areas. In other words, I have organized the software architecture assigning to each manager

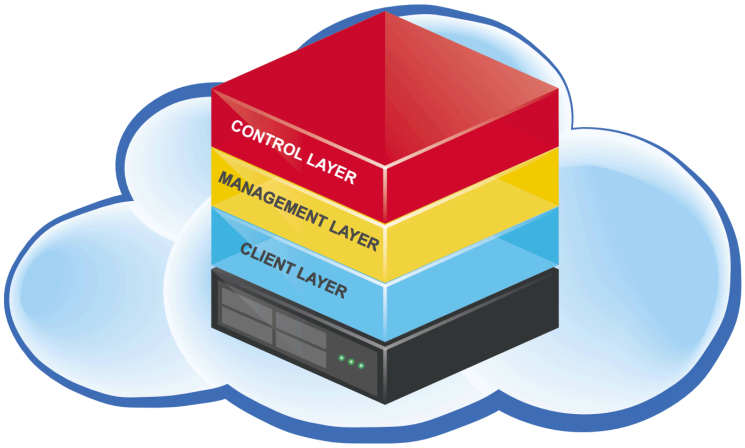

Figure 4.1: High level system diagram

tasks related to a restricted area of the cloud technology. The *Network Manager* is an example: it is responsible of all what concerns the network definition, creation, deletion, etc. but it does not matters what is going on with virtual machines.

This type of architecture gives to the code a modular structure that facilitates reuse and maintenance, granting low coupling between areas. Afterwards I have defined a "vertical" interaction between managers: two elements of different functional areas will not be able to interact directly. The interaction, where needed, will be provided in a hierarchical way by the root of the architectural tree. The modularity allows a posterior integration with additional blocks, related to new functional areas.

The lower part of the system will be the module that cooperates with the cloud technology. It could be defined as a Cloud Client, since it is a Java implementation of commands and operations offered by the API of the selected cloud technology. I chose to make it as an independent block in order to make the developed system independent from the chosen communication technology. Openstack offers a rich REST interface that allows to perform various operations over the infrastructure. I will not dwell on all these features, but instead focus on those that are necessary for the *Infrastructure Manager*.

From the user point of view, the developed system will provide a public interface. The implementation of this interface exploits manager services trying to achieve its goal. This main module determines the higher level of integration of the software; it orchestrates managers, obtaining data that allow it to manage the cloud.

I will start analyzing how the information flows throughout the system up to reach the user. In order to make effective the communication between modules, it is necessary to define a common vocabulary, or rather an information model. In next

section I focus on the information base used by modules to communicate with the exterior and between them, approaching the idea of defining a sort of ontology for these scenarios [41].

4.2. Assumptions

Before defining the information model I will state some assumption, or rather some simplifications that reduce considerably the complexity of the problem while not changing the essence of the work.

A01	Single cloud administrator
Description	First of all we assume that only Infrastructure Manager acts on the cloud infrastructure: in this way we avoid inconsistency problems that could be caused by a parallel management of several administrators.

Table 4.1: Assumption 01

A02	No over-provisioning
Description	Subsequently, we do not allow the split of a physical CPU into more virtual cores than physical ones it has. In other words we are preventing the over provisioning of computational resources.

Table 4.2: Assumption 02

A03	Already charged cloud elements
Description	We assume that some cloud elements such as disk *Images* had been already uploaded to the Cloud Controller through the specific service.

Table 4.3: Assumption 03

A04	Hosts with similar hardware configuration
Description	We assume that all Hosts, or rather all physical machines mount a CPU with the same architecture.

Table 4.4: Assumption 04

4.3. The Cloud Computing Information model

4.3.1. EMF Model

An important non functional requirement of the project is the utilization of an information model that had been defined by the research group and assumed as a standard. The definition of the model does not make part of the performed work, but it has a crucial importance in the cooperation between the IM and other elements developed by the members of the group; for this reason, I reserve some paragraphs to describe it.

The model has been defined using the ECore architecture, defined in the *Eclipse Modeling Framework*. This is a variant of the classic Essential Meta Object-Facility (EMOF) a closed meta-modeling architecture defined by the Object Management Group (OMG)[42].

Virtual environment

The virtual environment is defined around the concept of virtual machine, fulcrum of the system. A *Virtual Machine* is a virtualized running hardware: it is composed by a set of virtual *Resources* as *Virtual Memory, Virtual Storage Unit* etc. This element is strictly linked with other two modules: the *Virtual Appliance* and the *Virtual Instance.* The former is a definition of minimum resources that a VM needs to be deployed, while the latter represents the software running on it. The relationship between elements could be seen in the figure 4.2. All virtual elements are mapped on a physical one: physical resources are split and redistributed among VMs, but, as mentioned in the assumption section (4.2) CPU over-provisioning is not supported.

Physical environment

In the figure 4.3 we can see the integrating part of the previous EMF model; in this case the representation refers to the physical infrastructure of the cloud (the computational nodes), or rather, the infrastructure where *Virtual Machines* will run. A *Physical Machine* is composed by several resources, as *Processing Unit, Memory, Storage Unit* and *Network Interfaces.* These elements will be associated to the virtual resources, therefore to Virtual Machines. Virtualization techniques allow the physical component to host different virtual elements (e.g. a physical memory can host two virtual RAM, but each of them will be mapped to a dedicated area). All the virtual elements can not prescind from the physical ones, so each virtual element must have a total matching with the underlying resources. With this model we offer a static view of the entire infrastructure, but we not consider the dynamic

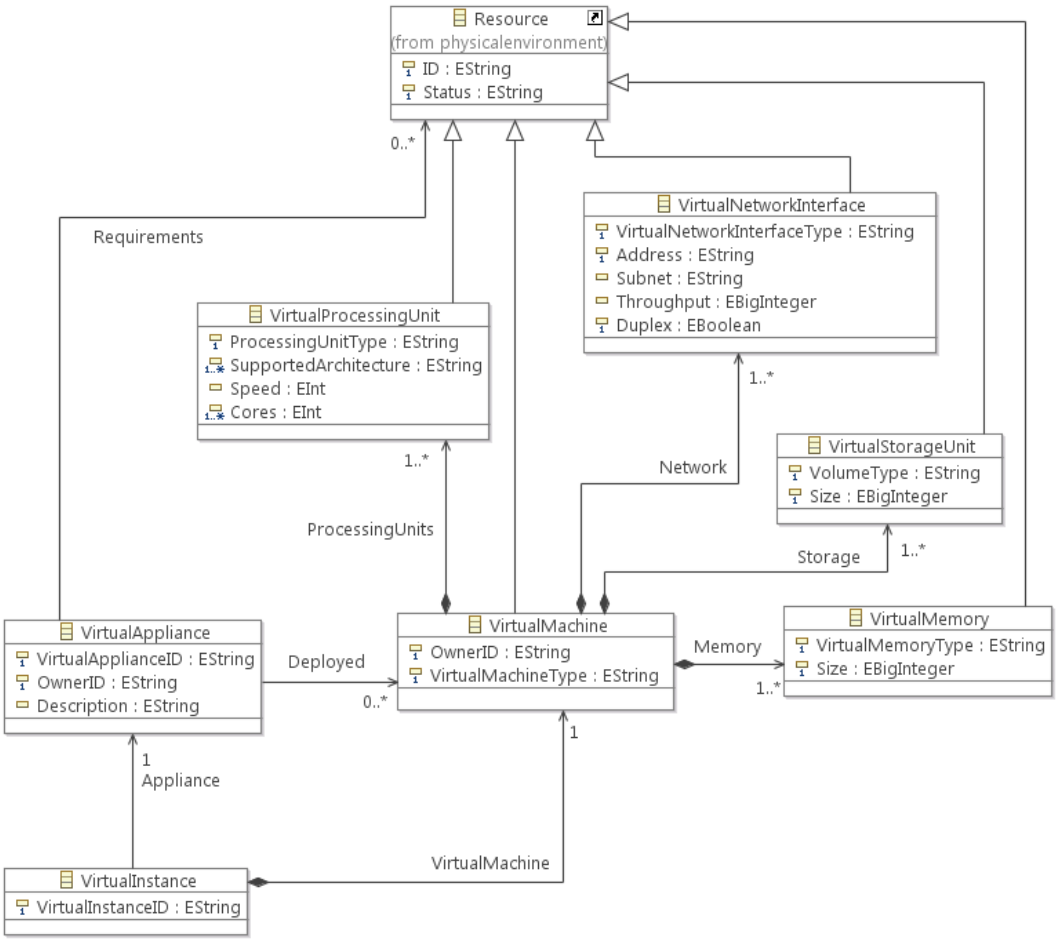

Figure 4.2: EMF Model of the Virtual Environment

information such as load or utilization etc. This information model is so general that permits to describe each kind of infrastructure.

4.3.2. OpenStack Information Model

In the OpenStack environment the conceptualization of the infrastructure had been though in a different way; the information model is more limited and pragmatic. It is mainly focused on the virtual aspect, while the physical part is left a bit in the background. In the Openstack world many elements I have previously explained are defined with other names, so it is necessary to describe the Openstack information model and make a mapping to ours. The central element of the OpenStack infrastructure is the Server Instance; this is a deployment of a OS disk *Image*

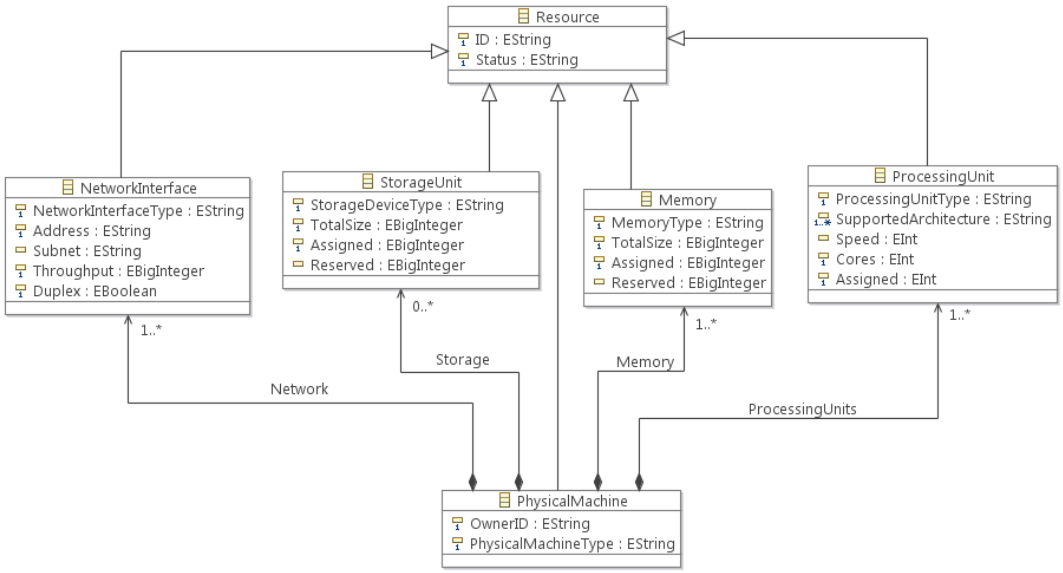

Figure 4.3: EMF Model of the Physical Environment

in a *Virtual Machine*. In OpenStack we can not separate the concept of Virtual Machine and Virtual Instance: the two elements are merged together in a single element: the Server. It is necessary to describe which are the bearing blocks of the system. Figure 4.4 puts in evidence the main elements of the OpenStack data model, grouping them according to the service they are managed from.

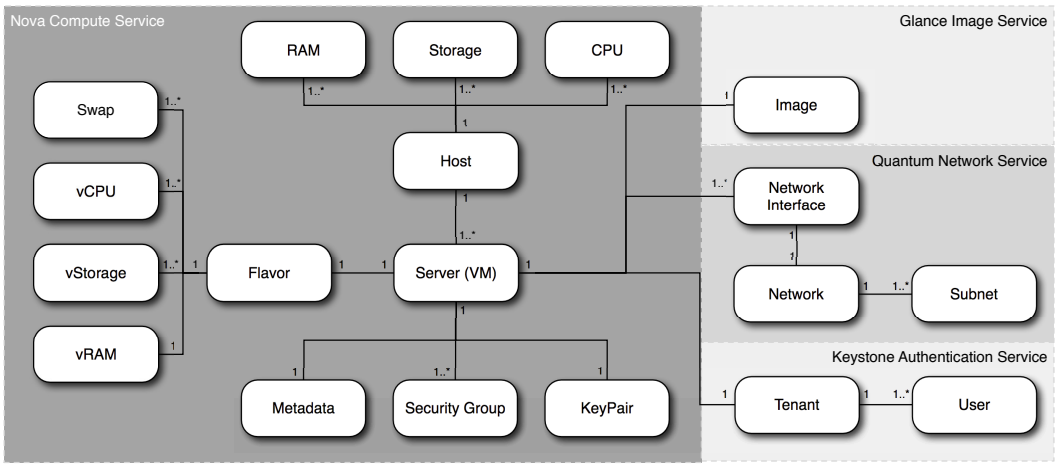

Figure 4.4: OpenStack Data Model

Tenant The OpenStack infrastructure could be organized in several projects, or better, following the producer definition, *Tenants*. A tenant could be defined as a restricted area of the infrastructure where only authorized *users* can access. This is used to isolate access to computational resources, and could be simply considered as an alternative term for a project. Each project has an unique identifier and a boolean value indicating its state. Tenants and users are managed by the *Keystone* authentication service, the security center of the infrastructure.

Listing 4.1: Example of Tenant specification

```
+----------------------------------+----------+---------+
|                id                |   name   | enabled |
+----------------------------------+----------+---------+
| bcb940507f644777849d1b3c47c6d6c0 |  gojira  |   True  |
| ca242d34fc7543bdbb964d7f97d80001 | mazinger |   True  |
| e706a03561e54b1f952330f7f632ff96 | service  |   True  |
+----------------------------------+----------+---------+
```

Network Interface The *Network Interface* is an attachment to a defined virtual network. Networks can be previously created by using the *Quantum* service, a module that provides the possibility of customize network topologies. As the previous elements each network owns an identifier associated to a network name and to some subnets; these are referenced by its identifier. All what concerns the network administration is borne by the specialized service (section 2.7.1) that offers a still incomplete API.

Listing 4.2: Example of Network specification

```
+----------------------+---------+--------------------------+
|  id                  |   name  |   subnets                |
+----------------------+---------+--------------------------+
|   910a4463-7b5f-4a90 | private | 1578ce33-26a7-4d97       |
+----------------------+---------+--------------------------+
```

Flavor The *Flavor* could be defined as virtual hardware template in which the system administrator defines sets of virtual resources. Each *Flavor* is identified by a unique numerical code and specifies a flavor name, the quantity in MB of Virtual RAM, the quantity expressed in GB of Virtual Storage Unit, an additional (optional) ephemeral memory expressed in GB, an optional swap memory, the number of Virtual CPUs, the ratio between reception and transmission speeds that could be used to arrange the network bandwidth (RX/TX), a boolean value that indicates if the flavor is shared among all tenants or not and finally an additional field used to specify optional restrictions as on which compute nodes the flavor can run on.

Listing 4.3: Example of Flavor specification

```
+---+-------------+-----+----+----+----+----+------+------+-----+
|ID |     Name    | RAM |Disk|Eph |Swap|Vcpu| rxtx |Public|extra|
+---+-------------+-----+----+----+----+----+------+------+-----+
| 1 |  m1.tiny    |512  | 0  | 0  |    |  1 | 1.0  | True | {}  |
| 2 |  m1.small   |2048 | 10 | 20 |    |  1 | 1.0  | True | {}  |
| 3 |  m1.medium  |4096 | 10 | 40 |    |  2 | 1.0  | True | {}  |
| 4 |  m1.large   |8192 | 10 | 80 |    |  4 | 1.0  | True | {}  |
+---+-------------+-----+----+----+----+----+------+------+-----+
```

Image The *Image* is a disk image containing an executable operating system. It must be loaded in the system through the specific service *Glance* (section 2.7.1). The image format must follow the rules defined by the service. The image is identified by a unique alphanumerical code which is associated to the image name, the status and the server name (only if the image had been created as a copy of a running server).

Listing 4.4: Example of Image specification

```
+----------------------+--------------+--------+------+
| ID                   | Name         | Status |Server|
+----------------------+--------------+--------+------+
|aee1d242-730f-431f-88c1| Ubuntu 12.04 | ACTIVE |      |
|0b27baa1-0ca6-49a7-b3f4| Ubuntu 12.10 | ACTIVE |      |
|df8d56fc-9cea-4dfd-a8d3| CirrOS       | ACTIVE |      |
+----------------------+--------------+--------+------+
```

The Virtual Machine - Server The Server is the running *Virtual Machine*, or rather the deployment of an *Image* with a *Flavor* over a *Host*. In the creation phase, the user can specify some additional attributes in order to customize the new instance. Servers represent the core of the cloud infrastructure and could be managed according to user or application needs. As we can see in the listing 4.5 even in this case each instance own a unique ID. To each identifier is associated also a name, which not necessary must be univocal, a status and a network address according to the specified network during the creation phase.

Listing 4.5: Example of Server specification

```
+--------------------------+---------+--------+------------------+
| ID                       | Name    | Status | Networks         |
+--------------------------+---------+--------+------------------+
| 86273742-e4fe-4d02-82f3 | fenix01 | ACTIVE | private=10.2.2.6 |
+--------------------------+---------+--------+------------------+
```

Additional elements On booting phase *Virtual Machines* are configurable with additional attributes. The first element we analyze is the*Security Group* attribute. The security group is a definition of network rules, that controls the traffic transmission over the virtual network. With the security group, the user can chose which protocols and which addresses are allowed to cross the virtual network.

The second important attribute is the *Key Pair*. In the OpenStack environment the controller can inject an SSH public key into an account on the instance, assuming the virtual machine image being used supports this. This aspect proves to be very important at the time of having to automate the secure connection between two machines.

Lasts two versions of OpenStack offer the possibility of choosing the location of new VMs. In other words, on booting the user can select which physical machine will host his instance. This feature is provided by the *Availability Zone* parameter, or rather an Amazon EC2 concept of an isolated area that is used for fault tolerance. The user can specify the name of the *Host* on which the server will be run.

Naturally, each machine will tend to have a distinct nature. For this reason, the user can customize the VM by sending an initial configuration file that will be run during the booting phase. Not necessarily this file must be a script, but it can also be a list of configuration parameters that follows a specific format.

With these elements we are able to to run a server instance accessible through a SSH connection. According to the idea of modularity we follow, in the design of the Infrastructure manager we charge each one of these elements to a specific manager that controls all its aspects.

The OpenStack environment includes many other aspects and possibilities to personalize the infrastructure; in our case a restricted set of functionality is used. Finally we notice how the same scenario could be formalized in different ways.

4.4. Component Diagram

In this section I analyze in detail each component already mentioned in the previous general overview. The pseudo-UML component diagram in figure 4.5 shows the system structure with an higher level of detail. We start from the lower part of the system, namely which is responsible for communicating directly with the cloud technology.

4.4.1. Client Layer

The client layer is the lower part of our system. It is a module that provides connectivity with the cloud through a REST interface that provides scalability, generality of interfaces and independent deployment of components. This level of

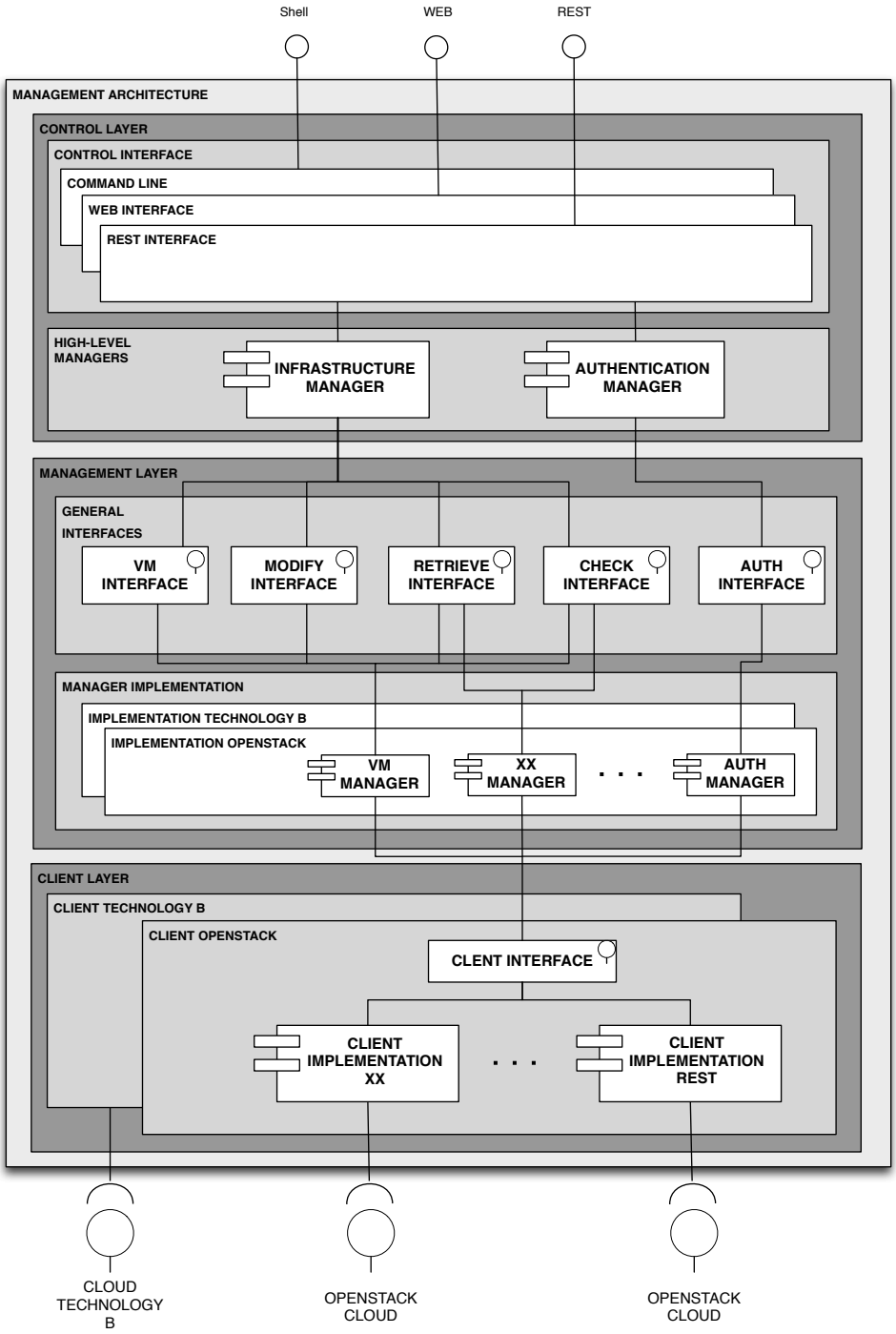

Figure 4.5: High level component diagram

the system does not make entirely part of the performed work, since I used a partially implemented Java API. I have designed and programmed an additional module in order to communicate with the OpenStack service *Quantum* (section 2.7.1).

The client uses a technology that permits to create and manage easily the JSON [43] messages as communication medium between the two operators. This part implements the most elemental calls to the cloud service, and represents the founding base of the entire Infrastructure Manager. It could be considered as a translation of the specific cloud API to JAVA language. This layer is strictly linked to the chosen technology that the user is running, and so it should represent a standalone block.

This is done through one or more modules, each designed to work with a particular access technology and cloud solution. The implementation of such interface may change according to the selected private to the adopted communication protocol (e.g. SSH, REST).

4.4.2. Manager Layer

The medium layer is the hearth of the system. In this part all functional areas are under the specific control of a specialized manager that is able to perform operations on the cloud. The sectorization and modularity allow adding new managers, where needed, without affecting the functioning of the rest of elements.

This layer is divided into two others: *General Manager Interfaces* and *Manager Implementations*. The *General Interfaces* sublayer offers three interfaces which provide management primitives that can be applied to every *Resource* as defined in our information model. In addition it defines other two interfaces, the former specific for the administration of virtual machines, the latter designed to accomplish all authentication tasks.

These interfaces are in turn implemented in the remaining sublayer, the *Manager Implementation* that includes the following managers:

- Authentication Manager - The *Authentication manager* is in charge of the first step of the communication between the IM and the cloud (more precisely the cloud controller 5.2.3). Before any interaction, the system must be sure of being already authorized to access the service.

- Virtual Machine Manager - The *Virtual machine Manager* controls and interacts with the compute service, or rather it manages VMs (Servers) in all their aspects.

- Flavor Manager - The *Flavor Manager* overlooks the correct definition of virtual resources configurations.

- Image Manager - The Image Manager interacts with the image storing service and manages the retrieving of image's information.

- Host Manager - The Host Manager has the task of obtain reliable information about physical machines making part of the infrastructure. In this way users could know where they can run their VMs.

- Network Manager - The Network Manager is the module that defines, deletes and modifies virtual networks that will be employed into the connection between VMs.

- Subnet Manager - The Subnet Manager is a adjunct of the previous manager. Its task is to administrate IP ranges in network definition.

- Key Pair Manager - The Key Pair Manager operates on the generation and control of ssh key pairs, that will be used accessing the instances.

- Security Group Manager - The Security Group manager is in charge of the retrieval of network rules that have to be applied in virtual networks.

- Virtual Appliance Manager - The Virtual Appliance Manager creates and destroy all kind of virtual machine templates, storing VA definitions in a local database.

To adapt a cloud solution to our proposal we had to complete three tasks: first specify a translation between our generic information model and the solution's own, second develop a corresponding set of Manager Implementations, and third create at least one interface for the Client Layer.

As mentioned, since the OpenStack infrastructure works with its own information model, our software should act as an interpreter between the two views of the same world. During the description of the developed software we will go on using as reference the former model but we will evidence the adaptation process that we had chosen.

Table 4.5 focus on the mapping between the two different information models (General Information Model 4.3 and the Openstack information model 4.3.2). Each one includes some elements that are not present in the other. For example the Virtual Appliance element is not defined in the Openstack environment. When we need to boot a server instance we need to pass all parameters *such as Image, Flavor, Security Group, Metadata, Network* etc. The introduction of the VA eases this task.

The VA could be seen as a Virtual Machine template that once defined can be reused: when the user need a specific instance, for example a web application server or a database server, he can retrieve and deploy an already defined VA where an image that overcomes that service is defined. In the template I define only fixed elements as image, flavor etc.

The definition of elements such Virtual Network would be inappropriate since the network, which VM is connected, depends on user needs and can not be defined

in a template; the same principle applies to Key Pairs. Since OpenStack does not provides this kind of functionality, the Infrastructure Manager add an interesting element in the deployment of cloud infrastructures.

Mapping between the General Information model and the Openstack model	
GIM[1]	Openstack
Virtual Instance	Virtual Machine (Server)
Virtual Appliance	Image
	Flavor
	Name
	Security Group
	Metadata
Virtual Memory	Flavor (RAM)
Virtual Storage	Flavor (Disk)
	Volume
Virtual CPU	Flavor (CPU)
Virtual NIC	Virtual Network
Owner	Tenant
Physical Machine	Host
-	User
-	Key Pair
-	Floating IPs

Table 4.5: Mapping between information models

4.4.3. Controller Layer

The top level can be considered as the orchestra conductor that coordinates all the musicians (managers) making a great "symphony". The Control Interface is the outward interface that connects the system to the manager. This layer is designed to be interchangeable and support several interface implementations at the same time. The motivation for this schema is that different users could prefer different management interfaces. Moreover, the user does not need to be a human operator at all, it can be other system, and therefore there is a need for interfaces more suited to this task. Samples of these interfaces could be command-line tools or an administration web page for a human operator, and a web services interface for an autonomic system that keeps care of the private cloud infrastructure.

[1]General Information Model

The High Level Managers sublayer provides to the Control Interface two components: an Infrastructure Manager for controlling the cloud through a set of management actions defined in our information model (and therefore technology-agnostic), and an Authentication Manager in charge of monitoring and enforcing the security model for the private cloud.

4.5. Man.O.S. detailed Design

The Man.O.S. acronym stands for Managing OpenStack, but refers also to the Spanish word *"hands"*, as the system can be seen as the executive part of a bigger project, formed by a monitoring module (*O.J.O.S.* - *"eyes"*) and a reasoning block (*Ce.Re.Pro* - similar to cerebro *"brain"*).

In this section I analyze the implementation of each component composing the bone structure of the IM.

4.5.1. The Client

As mentioned before, the client level provides communication with the cloud infrastructure and does not make part entirely of the performed work. In this section I will describe the changes and the improvement made to the module created by the Openstack Java SDK community[2].

Due to the continuous upgrading of the cloud technology, it has been necessary to integrate the client code with some modules. The introduction of the virtual network manager *Quantum* has forced us to design a specific client, that takes profit from all new features of the new network controller. The improvement of the source code has followed the architectural style of the already developed software in order to ease the collaboration in this community project.

For the collaboration with the Openstack Java SDK community I took advantage of the gitHub platform's facilities. The Openstack Java SDK is distributed under the Apache 2 license[3], so as open source software. I will maintain this characteristic trying to bring profit to the community[4].

On 21st December 2012 the code has been integrated in the main branch of the *Openstack Java SDK* project.

[2]For the source code, thanks to Luís Gervaso see: "https://github.com/woorea/openstack-java-sdk.git"

[3]For details over the Apache license see http://www.apache.org/licenses/LICENSE-2.0.html

[4]For the implemented and distributed source code "https://github.com/mattiapei/openstack-java-sdk.git"

The Quantum Model

In order to allow the communication between the client and the Quantum controller I have defined a set of basic classes (POJO - Plain Old Java Object) that will be serialized to a JSON message and then sent to the controller.

Each message that is sent to the main controller must follow a specified schema. Starting from the message definition[5] I created Java classes representing such messages, which properties will represent the variable's content. The documentation has been a bit misleading since the real content of the messages was partially different from those described in the guide. Thanks to the debug option of the *Quantum Phyton Client* (a developers release, since the official one does not exists yet) it has been possible to study the correct set of messages used during the communication.

The serialization and deserialization is made automatically throughout the adoption of the Jackson library [44]. Trying to explain the behavior I propose an explicative example of how this module works. In the following listings we can see a simple POJO representing a user (listing 4.6) and the relative JSON message (listing 4.7). For each attribute the serializer create a key-value entry in the JSON packet and vice versa, while for nested classes it creates a new JSON object[6]. Java lists are serialized in JSON arrays.

Listing 4.6: Simple POJO Example

```
public class User {

  public static class Name {
    private String _first, _last;

    public String getFirst() { return _first; }
    public String getLast() { return _last; }
    public void setFirst(String s) { _first = s; }
    public void setLast(String s) { _last = s; }
  }

  private Name _name;

  public Name getName() { return _name; }
  public void setName(Name n) { _name = n; }
}
```

[5]Available at: "http://docs.openstack.org/api/openstack-network/2.0/content/"

[6]For JSON element definition refer to: "http://www.json.org"

Listing 4.7: JSON message of the serialized POJO

```
{
  "name" : { "first" : "Joe", "last" : "Sixpack" }
}
```

Subsequently I created a specific class for each element of the Opensack network model. Each class simply reflects an API standard message.

- Network
- NetworkForCreate
- Networks
- Pool
- Port
- PortForCreate
- Ports
- Subnet
- SubnetForCreate
- Subnets

We notice that I designed a specific class for the creation phase, since the message content differs from the case in which an information request is performed. In other words, the message used to create a network differs from a message sent by the controller to describe an existing one.

The Quantum Client

After defining the messages that will be interchanged, we focus on the client, or rather the module in charge of sending and manage these messages. In this module we had defined all operations provided by the Quantum API as Java methods; each methods is associated to a specific URL, which, following the Openstack documentation, corresponds to a specific action.

Each method requires a parameter, precisely an object of the Quantum Model that will be serialized and sent. So, for example, if I need to create a new network in the cloud infrastructure, it will be necessary to define a *NetworkForCreate* object specifying all characteristic we need and pass it to the *createNetwork* method that will send it to the correct address.

In figure 4.6 we can see the client architecture: for each network element I created a set of specific classes that defines all operation that could be performed over it. The *Core* classes define static methods that recall the specific functions.

The Quantum Client object receive on creation 2 parameters with which it can invoke the *Quantum Command* interface:

1. The endpoint URL of the network service that usually is obtained by the Keystone service.

2. The Authentication Token, that allows the client to perform operations over the infrastructure, previously retrieved in the authentication phase.

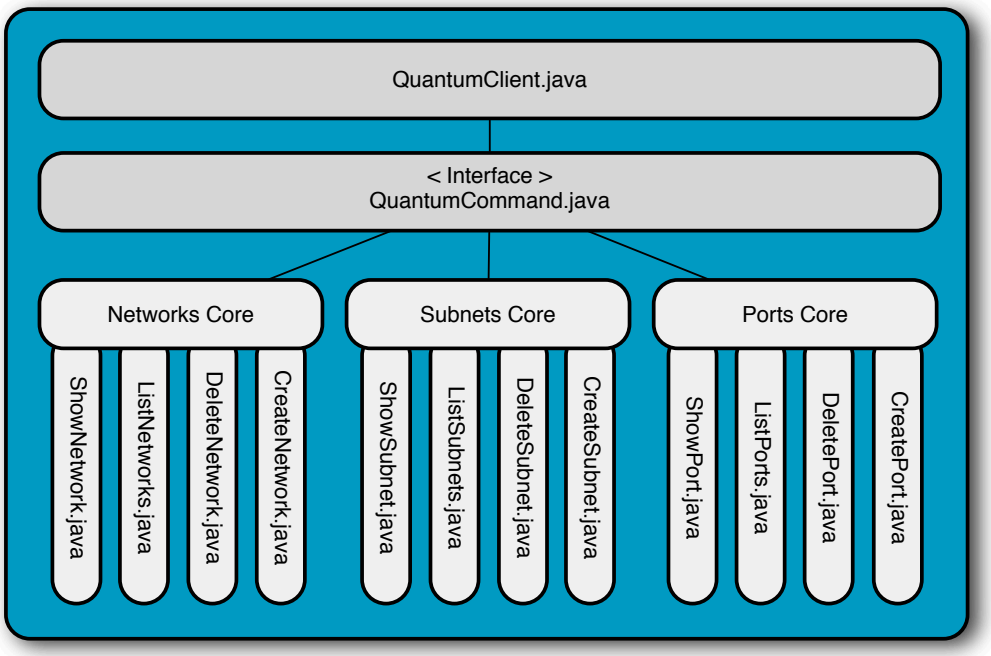

Figure 4.6: Quantum Client Structure

After describing the lowest level of our architecture we turn to define the beating heart of the Infrastructure Manager.

4.5.2. System interfaces

With the aim of designing a modular and scalable system, I designed a set of standard interfaces that define the connection point between the main levels described in the previous section. Interfaces are defined between Manager layer and Controller Layer and finally between Controller Layer and Outer World. Figure 4.7 represents the system stack and shows where interfaces are collocated.

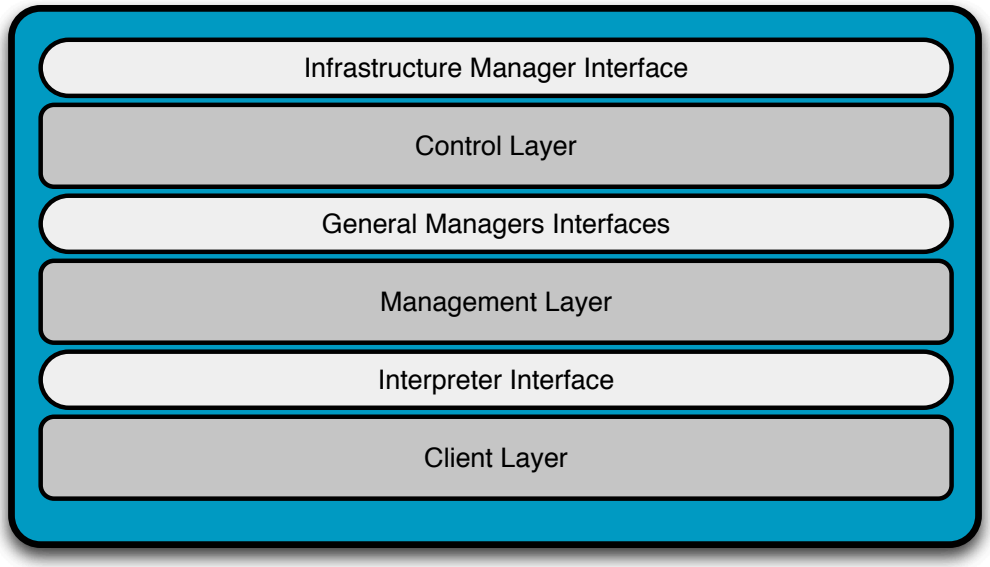

Figure 4.7: System Interfaces

The Client Interface - Interpreter

The client interface is the module that allows decoupling the Infrastructure Manager to the client technology. It defines all basic operation to perform against the cloud which will be implemented according to the elected protocol. With this solution, the adoption of a different client technology would imply only the rewriting of the client implementation, avoiding structural changes in the rest of the system.

Management Layer interfaces

Above the *Manager Layer* component I defined a set of interfaces that summarize the main operations that modules provides over the elements of a cloud:

- *Retrieve:* encloses functions to obtain data from the cloud.
- *Modify:* refers to all methods that modify the state of the cloud.
- *Check:* contains methods that allow to check the existence of a specific element, identified by its unique ID value.
- *VMinterface:* encompasses specific actions that only can be performed over Virtual Machines.
- *Authentication:* This interfaces is implemented only by modules that provide authentication functions.

All these interfaces define functions that handle JAVA Exceptions: this characteristic will be treated in section 4.5.5. Naturally, depending on the functional area, each manager handles specific objects.

These interfaces are implemented in the underlying layer. Each manager is not forced to implement all interfaces since the behavior of some methods is specific only for certain managers and it could not be generalized to all modules. For example the Virtual Appliance manager does not implements the VMInterface since it does not make sense to implement a function *move* that permits to change the position of the VA since this characteristic is merely own by a *Virtual Machine*.

Interface Auth The authentication interface just defines two simple methods that are used to authenticate the user and to check the validity of the token that is used to perform requests. The *getAccess* function receives as parameters the user credentials and the authentication endpoint URL that changes for each infrastructure; it returns an *Access* object that contains the token that will be used in all future requests. This element has a limited validity, so I created a function that checks if the tokens has already expired.

Listing 4.8: Interface Auth.java

```
public interface Auth {
  public Access getAccess(String tenant, String user, String psw,
       String authURL) throws ExceptionAuthentication;
  public boolean verifyAccessexpiration(Access access);
}
```

Interface Retrieve This interface defines two methods that could be used to retrieve information about a specific element of the cloud or to know which are the cloud elements already defined and active in the infrastructure. It consists of two main methods:

- *getList:* is used to retrieve a list of specific elements present in the cloud. For example it may return the list of the Virtual Machines running on the infrastructure, if the proper implementation is used.

- *getSpecific:* returns a specific element identified by a unique number.

If an error on retrieving occurs, a specific exception (*ExceptionOnRetrieve*) is thrown.

Listing 4.9: Interface Retrieve.java

```
public interface Retrieve<T> {
   public List<T> getList(Access access) throws Exception;
```

```
    public T getSpecific(String id, Access access) throws
        Exception;
}
```

Interface Modify In this case the interface defines methods used to change the state of the cloud. The retrieve methods do not change the number of the running instances or its configuration while this one does. Due to its nature, the implementation of this interface requires more precautions, since it modifies cloud instances.

- *create:* it generates a new element of the system, ensuring the accuracy of the entered parameters. The main argument passed to the function refers to the object that wants to be created. If we want to create a new *Virtual Machine* the passed object will be a VM object.

- *delete:* conversely is used to delete a cloud element, retrieved by its unique identifier.

Even in this case, if errors occur during the execution of the methods, specific exception will be thrown (*ExceptionOnCreate or ExceptionOnDelete*).

Listing 4.10: Interface Modify.java

```
public interface Modify<T> {
  public String create(T object, Access access) throws Exception;
  public void delete(String id , Access access) throws Exception;
}
```

Interface Check The last common interface is the control interface; although it defines just two method, it is very useful as utility. The defined function checks if a element, identified by its ID value, already exists in the system (*checkId* function). If it does not, a false value is returned. The second method verifies the presence of an equivalent element (this feature is not implemented in all managers since in some of them this function would not have a significant response). This interface has a paramount aim: avoid the creation of duplicated elements.

Listing 4.11: Interface Check.java

```
public interface Check<T> {
  boolean checkID(String id, Access access) throws Exception;
  T check (T object, Access access) throws Exception;
}
```

Interface VMInterface Now I pass to define an interface that envelops functions specific of virtual machines. These methods act over VMs changing their state or position. As we can see in listing 4.12 the first two functions could be used to suspend and resume an active running virtual machine, while the last two determine a change of the resource configuration: in one case the change of the physical host, in the second the change of allocated virtual resources. In this case, all functions handle a specific exception type, derived by the VMExeption packet.

Listing 4.12: Interface VMInterfe.java

```
public interface VMinterface {
  public void suspend(String vmID, Access access) throws
     VMExceptionOnCheck, VMExceptionOnRetrieve;
  public void resume(String vmID, Access access) throws
     VMExceptionOnCheck, VMExceptionOnRetrieve;
  public void migrate(String serverId, String hostid, Access
     access) throws VMExceptionOnRetrieve, VMExceptionOnCheck;
  public void resize(String serverId, Access access, String
     flavorid) throws VMExceptionOnRetrieve, VMExceptionOnCheck;
}
```

The Infrastructure Manager Interface

The *control layer* is designed to be interchangeable and support several interface implementations at the same time. The motivation for this schema is that different users could prefer different management interfaces. Moreover, the user does not need to be a human operator at all, it can be other system, and therefore there is a need for interfaces more suited to this task. Samples of these interfaces could be command-line tools or an administration web page for a human operator, and a web services interface for an autonomic system that keeps care of the private cloud infrastructure. The Infrastructure Manager interface defines all operations offered to the external user. As we can see from the figure 4.8 methods allow to manage virtual Machines and Virtual Appliance Configurations accessing all previously described functional areas. The starting element undoubtedly is the authentication function:

- *authenticate:* it permits the user to authenticate and obtain a token that will be used in all requests against the cloud infrastructure. The returned key is associated to token and must be used as reference for future requests.

- *checkAccessExpiration:* this method controls if the specified access token is still valid returning a boolean value.

- *startVM:* the statVM method permits to deploy a Virtual Machine from a defined Virtual Appliance identified by an “id” parameter. As all the following

methods, the invocation of this function requires the previous authentication. If an *Host* identifier is passed as parameter, the controller forces where the VM will be deployed.

- *stopVM:* the user can adopt this function to stop (shut down) a running Virtual Machine identified by a specific identifier.

- *suspendVM:* a running Virtual Machine can temporally be suspended or rather paused according to the user needs.

- *resumeVM:* the resume function must be adopted to return the suspended VM to the active state.

- *createVA:* this method enable to create new definitions of Virtual Appliance, storing it in a reliable way.

- *deleteVA:* as the name suggests, this method permits to delete a definition of a Virtual Appliance.

- *createNetwork:* the creation of a virtual network could be considered essential, since this is the one of the most important elements of the virtual infrastructure.

- *deleteNetwork:* with this function the user acquires complete control over the element Network.

- *changeVirtualResources:* sometimes the variation of workload may imply the necessity of improving the virtual resources attached to a VM; this method performs this function allowing the user to change the set of virtual resources dedicated to the overloaded server.

- *moveTo:* the increase of servers workload affects also the physical machine over which the VM is running. This method allows the user to move a VM to another physical host, redistributing the load of the infrastructure.

- *retrieveNetworks:* this function provides to the user the list of all defined virtual networks.

- *retrieveVMlist:* this is the last method and it gives to the external user the list of all running virtual machines.

The InfrastructureManager implementation can be considered the boss of the infrastructure and it is responsible for the execution of the above described functions. If any error occurs during the call to an underlying service (managers) it issues an exception that specifies the occurred problem.

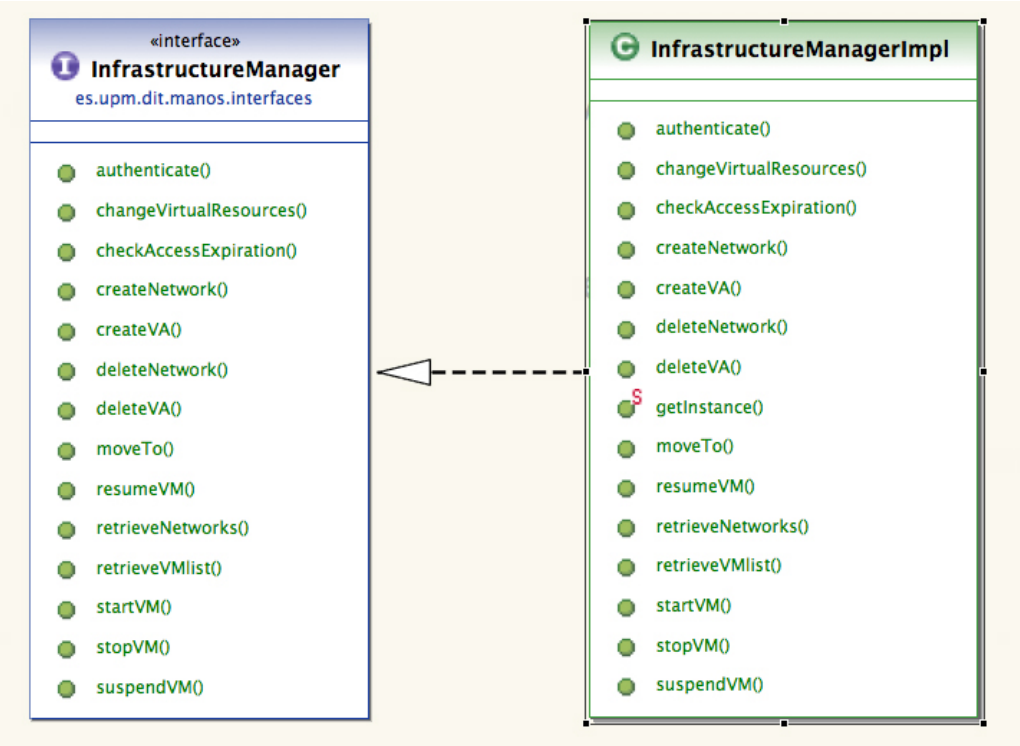

Figure 4.8: Infrastructure Manager interface

4.5.3. Manager Implementation

The Manager Level Implementation

In the previous chapters I talked about the specialized managers or workers that compose the core of the IM. As I defined before each worker is in charged of a functional area that may be mapped to the element list defined in the OpenStack information model (section 4.3.2). More precisely, whatever refers to *Virtual Machines* will be charged to a specific *Virtual Machine Manager*; the same will occur for *Virtual Appliances, Virtual Instances* etc.

In each module errors are managed following a standard process: each block handles specific exception, defined expressly for each functional area, easing the error detection. This is the layer where the mapping between the Openstack information model and the general cloud information model (4.3) is performed.

Another central point of the architecture is the amount of information that is going to be stored in the system; we know that the cloud technology has its own information base that should not be replicated in order to avoid inconsistency problems. With this assumption I created a system where data are not replicated and the majority of information elements are polled from the cloud and transformed in

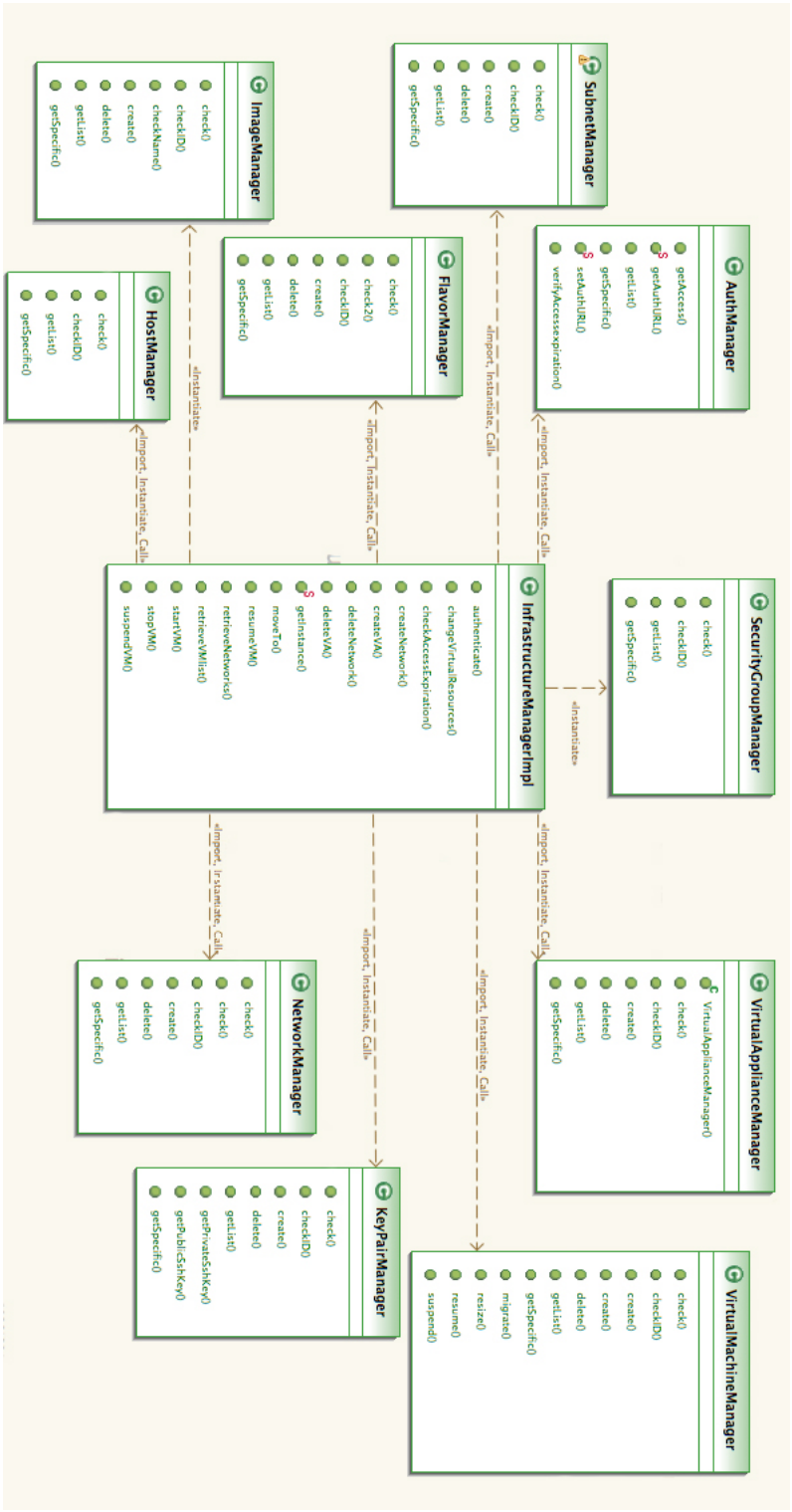

Figure 4.9: UML class diagram of the Infrastructure Manager implementation.

volatile objects that are used to carry the information through the different levels of the architecture.

In the management layer I also designed some additional classes used as utility. These classes offer additional services that workers will exploit. Figure 4.9 shows an overview of the management level detailed structure. From the UML diagram we can see how the Infrastructure Manager implementation uses the services provided by the *Managers*. If we think of a future integration with a distinct type of service, which may be the management service of storage volumes that will be introduced in the next official release of OpenStack, we can see that it would be easy to add a new module without disrupting the structure of the system, since the structure has a hierarchical and modular architecture.

Authentication Manager

As usual, in order to access a specific service users need to perform a previous authentication to verify its identity. A specific manager controls this aspect: the *Authentication Manager* is responsible of retrieving an authentication token, or rather a code that confirms the successful authentication. Moreover, the manager permits to retrieve the list of active tenants previously defined in the cloud or describe a specific one. This class represents the starting point for every workflow.

The *Authentication Manager* implements two interfaces as represented in figure 4.10.

The authentication function receives as parameters four elements: *user, password, tenant and authentication URL* which should be previously introduced in the cloud database by the system administrator.

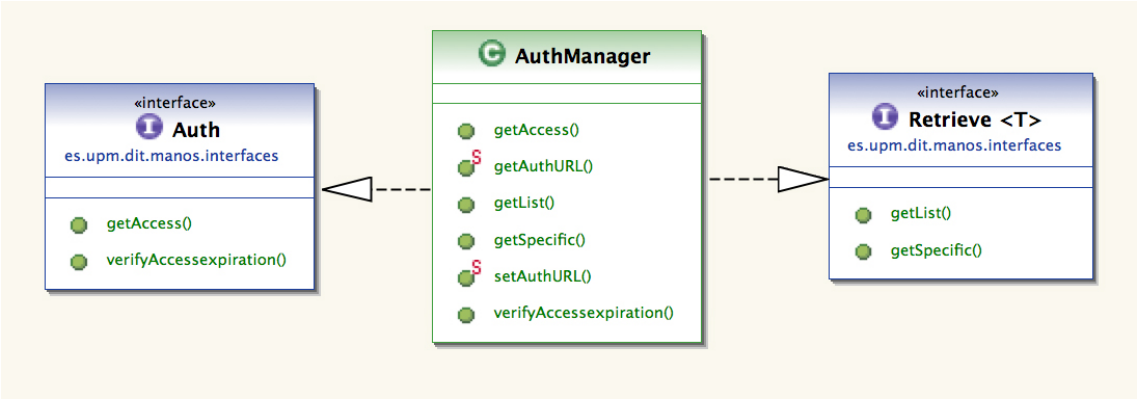

Figure 4.10: Authentication Manager

The *authentication URL* indicates the access point to the authentication API and varies depending on where the service is implemented. Retrieving functions can be performed only after the authentication procedure. Since the token has an expiration date defining the validity period, the *Authentication Manager* provides a function that checks if the token is still valid.

Image Manager

The image manager implements the interface *retrieve* and *check*. This implies that this module is only able to retrieve information about the existing elements and it is not authorized to create new OS image definitions. As we assumed, the IM works with OS images previously uploaded. Nevertheless the *Image Manager* can retrieve detailed information about existing images or check the validity of a specific one.

Flavor Manager

This module manages all what concerns virtual resources. The manager controls all aspects related to *Flavor* management, creation, deletion or modification, and it assures the process ends successfully. In addition, the manager permits to check the existence of equivalent *flavors*. In other words, if the user requests the creation of a new element, it is important to verify if, between the already created flavors, another one that has the same characteristics exists; in an affirmative case, it is not necessary to duplicate an element, so the system must return the item already defined. In this way, no duplicated element are present, maintaining the system clean. *Flavor IDs* are defined by an algorithm that grants their uniqueness.

Key Pairs Manager

The Key Pairs Manager controls all what refers to the creation of SSH key pairs (public and private keys). In order to connect to a VM is necessary that the cloud manager inject a ssh key during the VM's booting process; the key pair is created and saved through this controller. All keys, public and private are stored and managed locally, and a copy is loaded in the cloud system. When a key is deleted, the manager grants that the information remains consistent. Storing the key is necessary in order to provide an access method to the external user.

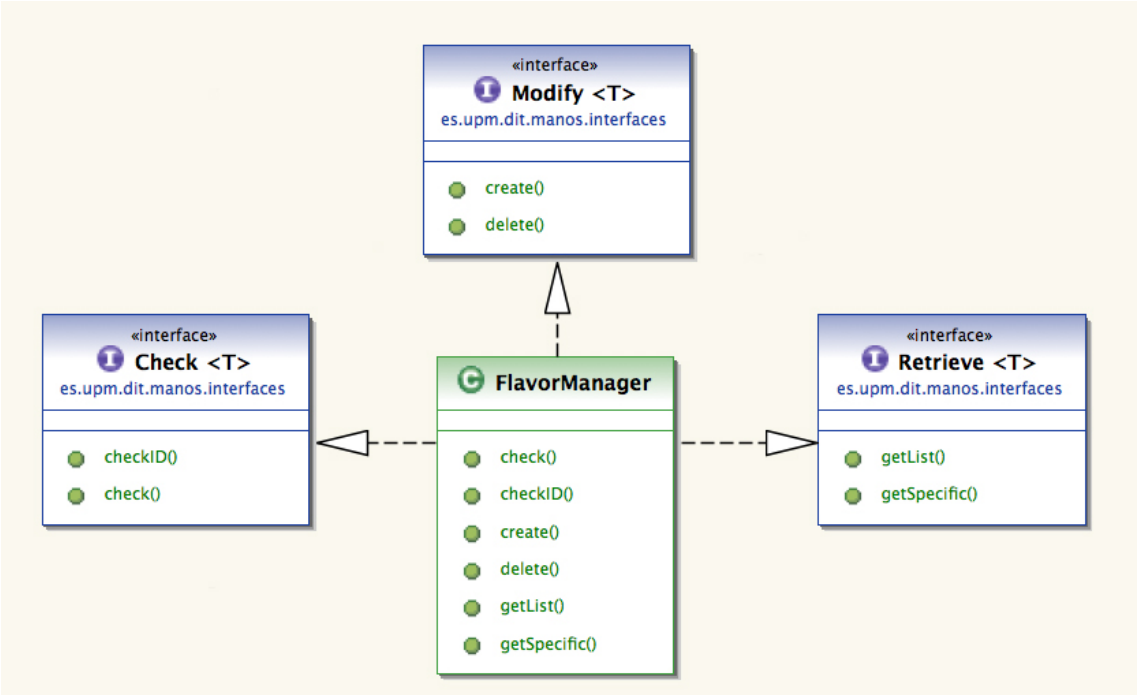

Figure 4.11: Flavor Manager

Security Group Manager

The Security Group Manager controls the set of rules governing the IP traffic in the network. In each group the user can specify which are the admitted protocols, the active ports or the allowed range of IP address from which a request can be made. So it limits the protocols and the traffic allowed by the instance. As the *Image Manager* this module permits the user to retrieve only pre-uploaded configuration sets. This feature must be active in the cloud infrastructure configuration.

Virtual Appliance Manager

The Virtual Appliance Manager maps the concept of Virtual Appliances to the Openstack's information model. I defined a Virtual Appliance Model class that refers to the *General Model*. As we can see in figure 4.12 each VA defines a list of minimal requirements associated to a specific image and is composed by four elements: an *identifier*, the *name*, the *image identifier*, the *flavor identifier*, and the list of *security groups*.

In this way we map the flow of information between the two environments. The manager, which implements the main aforementioned interfaces, controls the creation, deletion and modification of all *Virtual Appliances*. It stores and deletes each

VA configuration aided by an utility class that I will describe in the section 4.5.4.

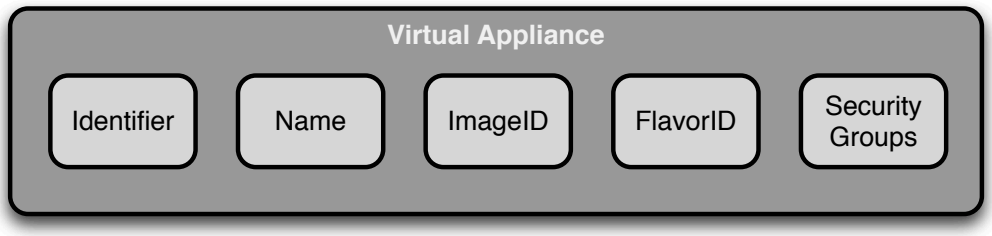

Figure 4.12: Virtual Appliance

Network Manager and Subnet Manager

With the integration of the quantum module in the OpenStack project, the customization of the virtual network infrastructure has increased considerably and new network topologies can be designed. In our system, the module in charge with these aspects is the *Network Manager*, that in collaboration with the *Subnet Manager*, responsible for managing the organization of IP addresses, can create and manage various environments of virtual networks. The user can create a new instance of virtual network passing to the manager the name, the network address and the network mask of the new LAN. The manager returns the identifiers of the created element.

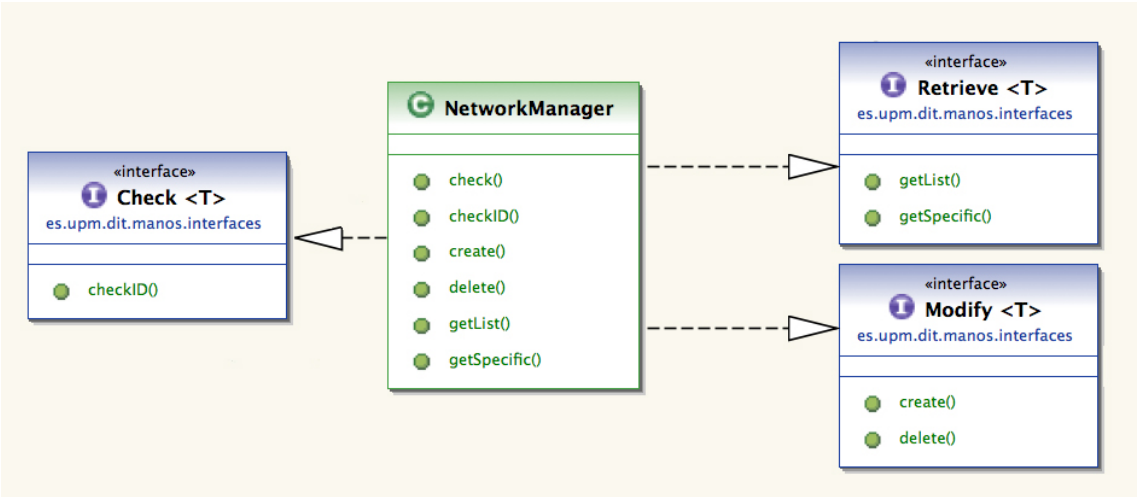

Figure 4.13: Network Manager

Virtual Machine Manager

The *Virtual Instance Manager* is committed to manage the life-cycle of *Virtual Machines*. Like all others managers it implements the three interfaces *Check, Modify* and *Retrieve* but in addition implements the specific interface VMinterface. The module has the complete control over VMs, and it could create, delete suspend or resume them. The manager is able to retrieve the list of running servers or a detailed description of a specific one. VMs can be deployed starting from a VA from which, if not specified, it takes the name. But the two most relevant functions refers to the modification of the VM status. As defined in the VMInterface this manager is able to change the set of virtual resources attached to the VM. This function receives as parameter only the VM identifier and the identifier of the new *Flavor*. VMs can also be moved from an host to another, in order to permit a redistribution of virtual instances over the physical infrastructure. In order to ease this feature the manager provides the possibility of indicate on which *Host* the *Server* must be run, avoiding in case of critical resources, the necessity of an immediate displacement.

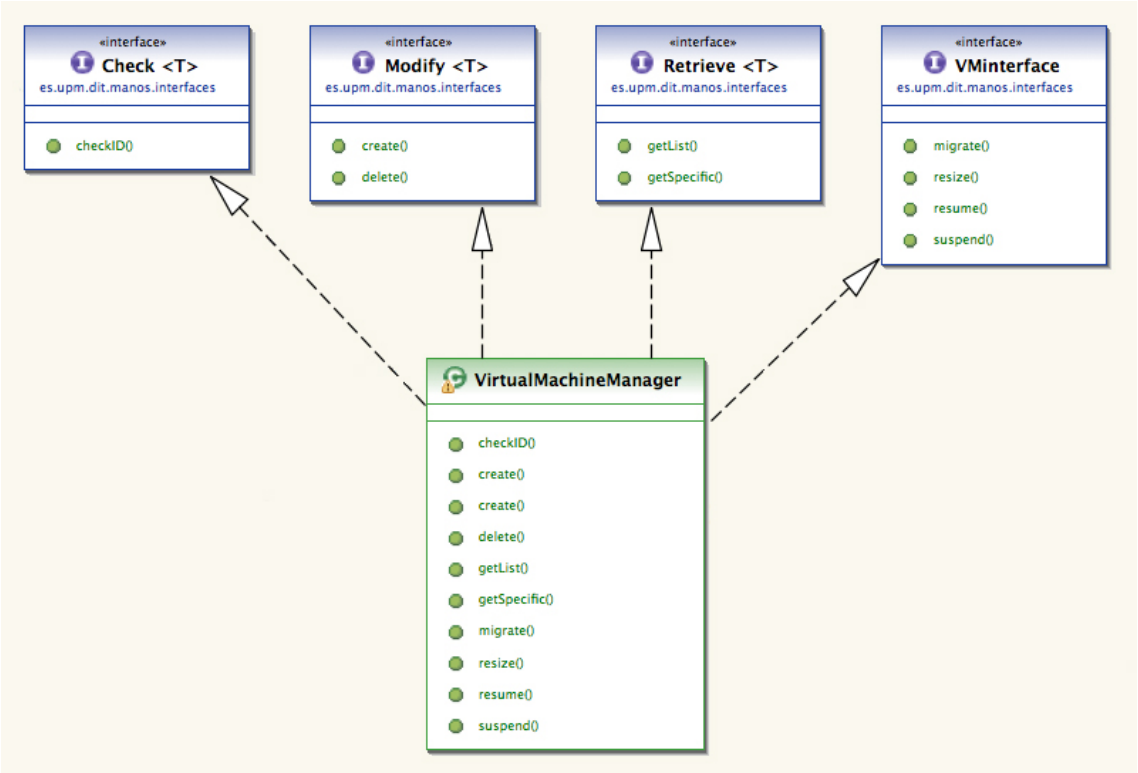

Figure 4.14: Virtual Machine Manager

Host Manager

Last but not least the *Host Manager*, the manager charged of retrieve reliable information about physical machines composing the underlying infrastructure. As others managers it implements just the interfaces needed to obtain information about cloud nodes since it would not be possible create or delete new *hosts*. This manager is of crucial importance when booting a new VM the *host* parameter is not null, or when a server needs to be displaced on another host machine.

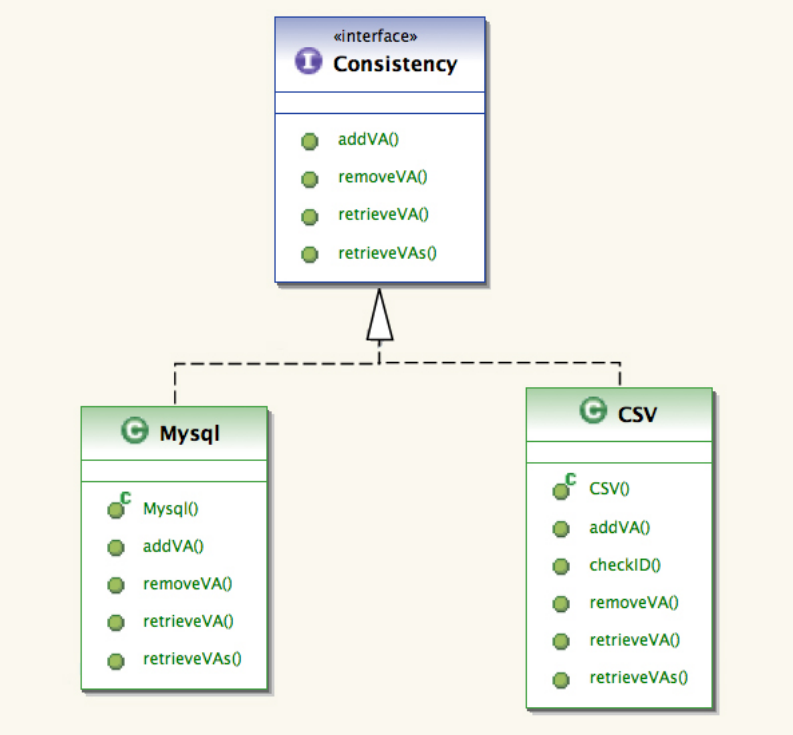

Figure 4.15: Consistency interface implemented using CSV and mysql.

4.5.4. The Utility Package

The utility package is a tool set, used to perform operations that not concern directly with the cloud system. I defined a Java interface (figure 4.15) that defines a set of functions that could be used to perform operations referring the storing of *Virtual Appliances* and *Key Pairs*. This interface was initially implemented by a class that saves data into a mysql database.

Given the low amount of stored information, the use of a database of this type was completely inappropriate and I opted for an implementation that would require a much lower computational cost. As a consequence I defined a second implementation

that stores data in a CSV file[7]. This option is found to be better in terms of time and occupied space. In the case the amount of data stored increases, if it were considered necessary, it would be easy to change the implementation used to the former one.

4.5.5. Exceptions

Handling errors is an central part of software development and can be managed in different ways. In our *Infrastructure Manager* errors are managed through the use of Java Exceptions. In order to ease the recognition of the trouble I created different classes of exceptions (extending the java class *Exception.java*), which follow the division in functional areas. Thus, each manager triggers its own set of exceptions, making the error management easier.

As I mentioned before, exceptions refers to the *phase* in which the error arose. I defined three types of exception classes: *OnCreate* exception triggered during the creation task, *OnRetrieve* exception thrown when errors on retrieving functions occur and finally *OnCheck* exceptions.

In principle, the software throws an exception when a user request can not be performed or a valid response can not be provided. Exception packages follow a modular scheme: depending on which part of the process the problem has appeared, a different exception is thrown.

4.5.6. The Infrastructure Manager Implementation

The controller level is substantially formed by the infrastructure Manager implementation. As I said before it is the conductor of the system that uses the facilities provided by the medium level and offers a service to the outer world. It implements the *InfrastrucutureManager.java* interface and it is responsible to collect all required parameters that permit a correct execution of the plans.

An important aspect of this module is its implementation nature: it is a singleton element. This means that a single instance of this class can run at a time. When the manager is invoked, if another instance is present, the reference to the already created one is returned, otherwise a new one is instantiated. With this construct, we ensure that only one manager is managing the cloud and we don't have inconsistency problems.

Every function described by the *Infrastructure Manager Interface* accepts only parameters that make part of the *General Information Model - GIM*, abstracting the user from the underlying technology.

[7]Comma-Separated Values

The *Control Layer* is composed by two main logical elements: the *Infrastructure Manager* and the *Authentication Manager*. The former manages the interaction with the outer world, while the latter manages the security aspect of the infrastructure.

In next paragraphs I analyze the implementation of the primitives offered by the Infrastructure Manager.

Authentication Functions

In order to avoid continuous and repeated authentication processes, when an authentication is performed, the *Authentication Manager* retrieves an access token and verify its validity; afterwards it maps the obtained value in a *table* where the *access* object is associated to a key value that will be returned to the user.

In this way, each time the user invokes a requests, he simply introduces the *key* value as parameter without caring about the access object. Each time a request is performed, the manager controls the validity of the access element; if it has already expired, an *Authentication Exception*, specifying the cause of the failure, is triggered and a new authentication procedure is requested .

Create and Delete a Virtual Appliance

The creation of the *Virtual Appliance* involves several managers, since a VA is composed by different cloud elements. The user passes as a parameter all configuration options that he needs for its new VA: the name, the image identifier, the quantity of vRAM and disk needed by the future VM and the security group rules. All these elements are enveloped in the structured element *VirtualAppliance*, which makes part of the GIM. Naturally the previous authentication is a mandatory condition. After checking the validity of the access token, the manager controls that another equivalent *Resource Set* is present. If so, it retrieves the resource identifier, otherwise it creates a new one with the specified values.

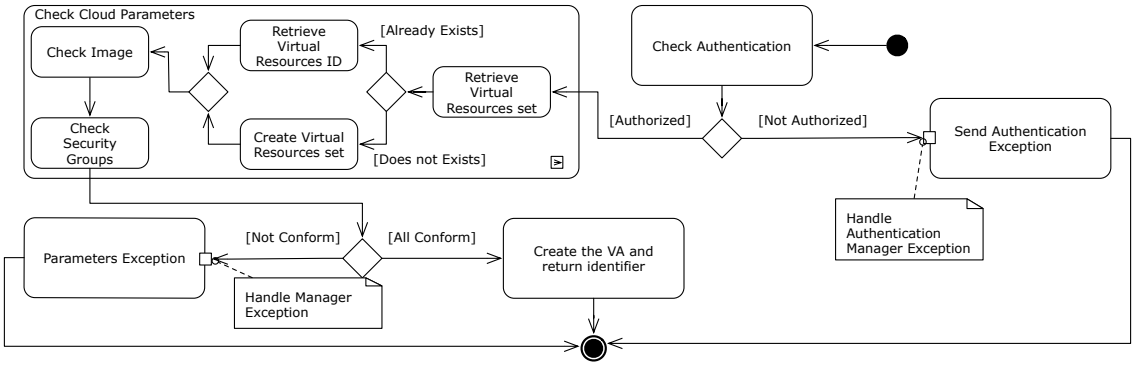

Figure 4.16: Activity diagram of the Virtual Appliance creating process.

Subsequently it controls the existence of the specified *Image* and *Security Groups*. Last, if the resource set had been created from scratch it creates the new VA and returns the identifier, else it checks if another equivalent VA has already been specified. If so, the crating process aborts and the identifier of the existent VA is returned, otherwise the new *Virtual Appliance* is created.

On the delete phase the manager simply controls if the element effectively exists. In that case it proceeds with the cancellation of the VA from the database.

Create and Delete Networks

The second step of the path towards the creation of a VM consists in the creation of a new *Virtual Network*. This primitive function reflects the characteristics of the underlying *network manager*, which allows you to create a new virtual network by specifying the name, the network address and network mask. Even in this case, parameters are passed through an object defined in the GIM (section 4.3). The identifier of the resource is returned to the user and it will be employed in the creation of new VMs or in the deletion phase.

Start and Stop a Virtual Machine

The two most important primitives functions refer to the creation and deletion of *Virtual Machines*. The boot of a VM requires a previous authentication and definition of at least one VA and one Virtual Network. A *VirtualAppliance* object is passed as parameter as the *VirtualNetworkInterface* and the key pair name. The function checks the validity of the introduced values employing the underlying managers. When the key pair does not exists, the IM takes care of creating one. After that the validity of the Network and VA identifiers is checked. If a *PhysicalMachine* object is introduced, the IM verifies the active state of the host and forces the creation of the VM over such machine. If all tests are passed, it creates the new Virtual Machine according to the specified parameters and returns its identifier.

The *PhysicalMachine* parameter is an optional element. In the case where the parameter were not specified, the choice of the most appropriate *Host* will fall on the cloud controller. If an error occurs, a *InfrastructureManager* exception is raised.

Advanced Operations over Virtual Machines

The last set of primitives is used to complete the set of functional requirements that I defined at the beginning of the project. More precisely the IM exports four primitives that allow the user to change the set of resources attached to a VM checking the validity and correctness of all parameters. These functions as the *moveTo* method are of paramount importance when the external user is an automated agent that tries to balance the work load of the infrastructure optimizing the number

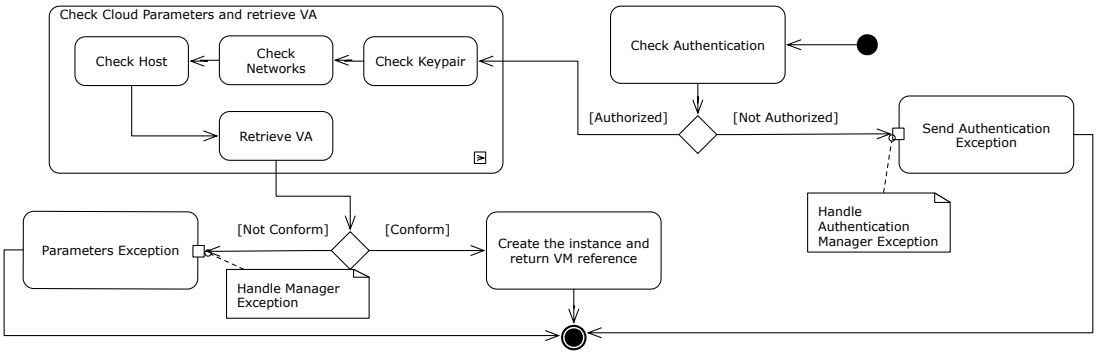

Figure 4.17: Activity diagram of the Virtual Machine starting process.

of used physical resources, since load balancing with infinite resources would be a trivial problem.

Later, it can be interesting for an administrator to suspend instances, if a maintenance is planned, or if the instance is not frequently used. Suspending an instance frees up memory and vCPUS (suspension could be compared to an hibernation mode of common desktop OS).

Chapter 5

Test

In this chapter I will focus on the main steps of the test process of our system. I will start with a small overview of the testing world and then I will detail the characteristic of our test design. In the first part I will describe the design of the tests, and the main tools that has been used in the deployment of this part.

5.1. Introduction to Tests

The test plan is used to prove the system is functioning as designed and the developed software responds to the user requests. The main aim is to grant the quality of the product since the complexity of these software systems, in companion with the human fallibility may nick the integrity of the system. Tests perform verification and validation of the system: validation is the proof that the system is providing all functionality required by the user; verification is the ascertainment that the system has been correctly developed. Although the utilization of batteries of tests may be considered a paradoxical behavior, we can not assure the absence of errors, since the test itself may contain them.

Usually, tests use to be divided into three categories that are linked to the main phases of the development: unit tests, integration tests and system tests. In addiction, it is necessary to make regression tests in order to verify that modifications and changes to the code do not introduce errors in the system and to detect errors in previous tests.

- **Unit test:** Unit tests are used to verify the functionality of a module or component. In general, they are also known as behavior tests or structure tests, respectively. They could be designed as Black Box Tests or White Box Test: the former ones take into account the code structure, whereas the latter ones just focus on the correctness of the interface or module definition.

- **Integration test:** Integration tests allow to verify if all modules properly interact in order to meet the system requirements.
- **System test:** The aforementioned tests are verification tests. System test are validation tests which try to verify if the developed software satisfy user requirements.

In addiction to these, we can consider performance tests, whose aim is to verify that capacity requirements, established for the system, has been fulfilled.

5.1.1. Tests in Java Environment

In order to test our code we can choose among various alternatives:

- **Print debugging** is the act of watching trace statements, or print statements, that indicate the flow of execution of a process. This is sometimes called printf debugging, due to the use of the printf statement in C.
- **Log debugging,** that can be provided by many available tools as Log4J [45] or the package *java.util.logging* embedded in the java framework.
- **Debugging tools,** that usually are integrated in common IDEs (*Integrated Development Environments*).

In my case I will use Eclipse IDE [46], which provides an embedded debugger and permits to trace the state of our software during its life-cycle. As all debugging tools it permits to stop the execution, to see the execution stacks of each thread, the variables etc.

During the installation of the infrastructure I made massive use of Openstack system logs, where the system write all performed operations; in this way I could control the set of modules loaded at the start of the process and in case of error, which was its source. On the other hand, each test was accompanied by a process of monitoring via terminal which verified the correct performance of the work. The correct deployment of cloud elements was constantly monitored through simple *shell* commands. Execution logs were also a big aid, since they show the communication (exchanged messages) between IM and Cloud Controller.

5.2. Test Architecture

In this section I describe the set of implemented tests that lead to the validation of the implemented software, certifying the completion of the proposed functional requirements. I adopted a bottom-up approach, starting to test small units of the

management layer, up to integrate all simple functions in a more complete verification that groups all components and verify the correct cooperation. Lastly, the implementation of an additional service over the IM has been developed as validation test of the performed work.

5.2.1. Unit Tests

The formalization of a battery of unit tests to verify certain aspects of the system will provide the following benefits:

- They allow us to clearly define the behavior of the code before deploying it (if we write the tests before the code).
- They maintain their value over time. The other ways of testing the code are not persistent: executing them again involves the same effort than performing for the first time.
- They allow regression testing whenever you modify the code, ensuring integrity.

Each method has been tested in order to verify the correct behavior in front of a normal flow of operations or in error situations.

The test code has been wrote using the unit testing framework JUnit [47], a subset of classes that permits to run the implemented code in a controlled way. These unit test have been developed as *white box* tests, also known as *transparent box testing* or *structural testing* where the code test is wrote knowing the internal structure of the element under prove. In this way the tester can verify the correct work flow of the unit, even in cause of faults.

The first step had been create unit test bundles for checking the correctness of managers behavior. For each manager I wrote a battery of unit tests validating the response of each implemented function singularly. I assumed the underlying level (client) was error free.

For each functional area, so for each manager, I verified the correct creation, deletion and retrieving of cloud elements since the testing environment is composed by the infrastructure manager and the operative cloud technology. Each test is programmed in order to not leave residual elements of its execution.

Test class	Unit Tests	Result
AuthManagerTest.java	*testGetAccess()*	Success
	testGetTenantList()	Success
	testGetSpecificTenant()	Success
	testGetAuthURL()	Success
	testVerifyAccessexpiration()	Success

FlavorManagerTest.java	*testGetList()*	Success
	testGetSpecific()	Success
	testCreate()	Success
	testDelete()	Success
	testCheckID()	Success
	testCheck()	Success
HostManagerTest.java	*testGetList()*	Success
	testCheckID()	Success
ImageManagerTest.java	*testGetList()*	Success
	testGetSpecific()	Success
	testCheckID()	Success
KeyPairManagerTest.java	*testGetList()*	Success
	testGetSpecific()	Success
	testCreate()	Success
	testDelete()	Success
	testCheckID()	Success
	testCheck()	Success
NetworkManagerTest.java	*testGetList()*	Success
	testGetSpecific()	Success
	testCreate()	Success
	testDelete()	Success
	testCheckID()	Success
	testCheck()	Success
SecurityGroupManagerTest.java	*testGetList()*	Success
	testGetSpecific()	Success
	testCheckID()	Success
SubnetManagerTest.java	*testGetList()*	Success
	testGetSpecific()	Success
	testCreate()	Success
	testDelete()	Success
	testCheckID()	Success
VirtualApplianceManagerTest.java	*testGetList()*	Success
	testGetSpecific()	Success
	testCreate()	Success
	testDelete()	Success
	testCheckID()	Success
	testCheck()	Success
VirtualMachineManagerTest.java	*testCreate()*	Success
	testGetSpecific()	Success
	testCheckID()	Success

	testSuspend()	Success
	testResume()	Success
	testDelete()	Success
	testResize()	Success
	testMigrate()	Success

Table 5.1: Unit test set

As can be seen in the table, all implemented modules work correctly. In each test class I have programmed a automatic set up that provides to the module an operating environment where its capabilities could be verified. After the execution this environment is dismantled leaving the system exactly as before the test. Listing 5.1 shows an example of unit test applied to the *Virtual Appliance Manager* in the creating function. We can see that the test verifies the behavior of the function in different cases:

- Case 1: the test tries to create a new VA (with a correct set of parameters)
- Case 2: the test tries to create a new VA with the same set of parameters used in the previous case. Then verifies if the returned identifier is the same.
- Case 3: a wrong parameter is introduced.
- Case 4: a null parameter is introduced.

For example in the *second case* the test verifies that when we try to create a new VA with the same set of parameters of an existing one, the manager does not duplicate it, but returns the already existing.

Listing 5.1: Virtual Appliance testCreate() function

```
@Test
public void testCreate(){

  //Case 1: all elements are correct
  va = new VirtualAppliance();
  va.setFlavorid(fm.getList(access).get(1).getId());
  va.setImageid(im.getList(access).get(1).getId());
  va.setName("Cid");
  secgroups = new ArrayList<String>();
  secgroups.add("default");
  va.setSecgroups(secgroups);
  try {
    id = vam.create(va, access);
  } catch (VirtualApplianceExceptionOnCheck e1) {
```

```
      assertTrue(false);
      e1.printStackTrace();
    } catch (VirtualApplianceExceptionOnCreate e1) {
      assertFalse(true);
      e1.printStackTrace();
    }

    //Case 2: The VA already exists
    try {
      assertEquals(id, vam.create(va, access));
    } catch (VirtualApplianceExceptionOnCheck e1) {
      e1.printStackTrace();
    } catch (VirtualApplianceExceptionOnCreate e1) {
      e1.printStackTrace();
    }

    //Case 3: One parameter does not exist
    va.setFlavorid("abcdefg");
    try {
      vam.create(va, access);
    } catch (VirtualApplianceExceptionOnCheck e1) {
      assertTrue(false);
      e1.printStackTrace();
    } catch (VirtualApplianceExceptionOnCreate e1) {
      assertTrue(true);
      e1.printStackTrace();
    }

    //Case 4: One parameter is null
    va.setFlavorid(null);
    try {
      vam.create(va, access);
    } catch (VirtualApplianceExceptionOnCheck e1) {
      assertTrue(true);
      e1.printStackTrace();
    } catch (VirtualApplianceExceptionOnCreate e1) {
      assertTrue(false);
      e1.printStackTrace();
    }
  }
```

5.2.2. Integration Tests

The development process has required continuous integration tests between the *Infrastructure Manager* and the cloud. The selected testing approach was the *Bottom Up* method: this approach firstly tests the system integrating its components

from the lowest level , then follows with the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.

This process has been followed throughout the whole development process: once I created a new element, I immediately tested it. If the verification was successful, the process continued up to the development of the next level of the architecture. In this way the system was growing stable on a strong foundation.

5.2.3. System Tests

Test Bench

Before starting the design and development of the IM, it was necessary to create a test bench that would allow us to study more thoroughly the technology and understand its potentialities and restrictions. Therefore we dedicated five computers to the installation of the platform in order to deploy a cloud infrastructure for testing processes.

Since the OpenStack software is open source and the community of developers is very active, the program is constantly updated due to several bugs that affect the various releases. These errors and the lack of updated documentation that persists incoherent with the version of the code have made the preparation of this workbench slower than planned.

I selected the *Openstack Folsom Release* right after its publication. This choice was made because of the advantages introduced by new features, as the new network service (still unstable probably until the next version). Even if the test version of the new release (code name *Grizzly*) had been released just a month after the announce of Folsom stable release, I considered premature to use a service in its first test edition. For this reason, I have updated only some parts of the release, so as not to upset the system already installed. It is necessary to mention that in the first milestone of the development cycle of the Grizzly release 399 bugs have been identified and fixed [48].

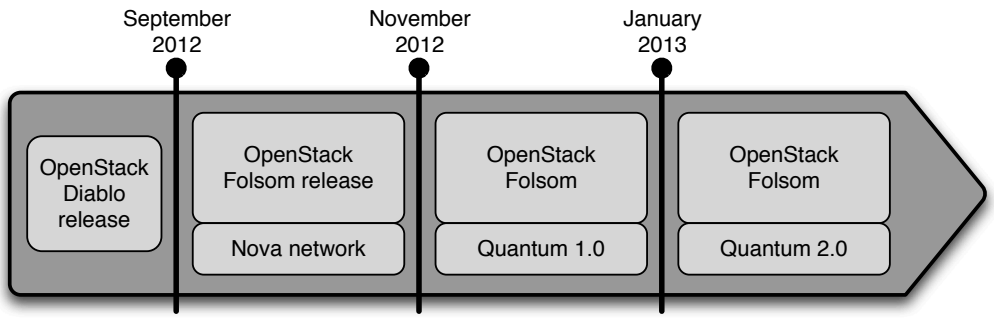

Figure 5.1: Graph representing the updates to the test cloud infrastructure

Workbench Architecture

I mounted 5 computers connected with a dedicated private LAN. One of these is connected to the Internet through a second physical network interface and became the main controller of the cloud infrastructure.

The *Cloud Controller* runs the main services while the others just run the compute service (2.7.1) and the network plugin (hereafter these machines will be called *compute nodes*). In picture 5.2 we can appreciate the physical architecture of our infrastructure.

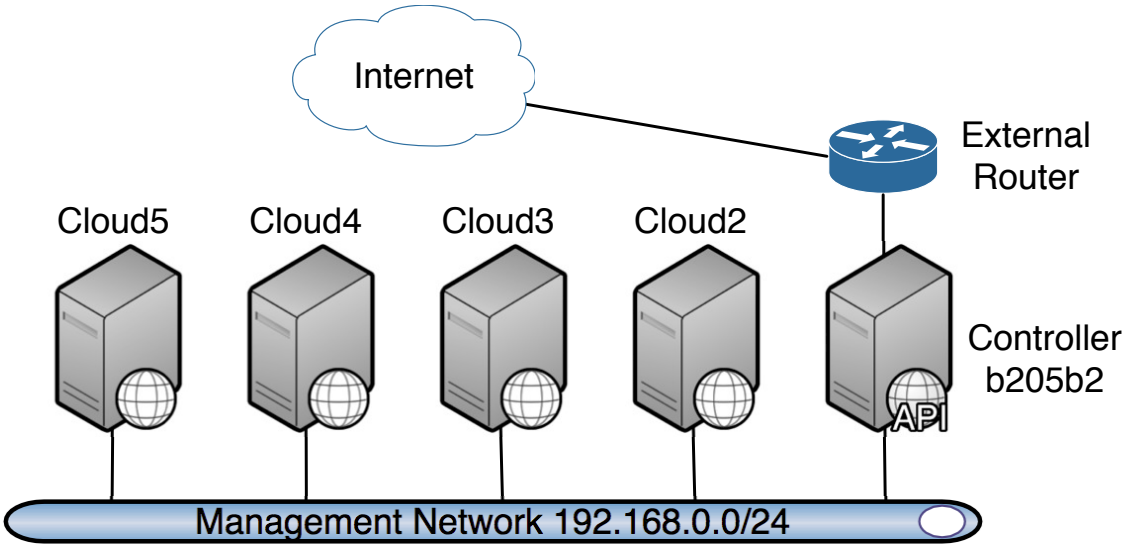

Figure 5.2: Test bench architecture

Test Results

The next set of tests tries to verify the correct interaction between the user and the *Infrastructure Manager*. Starting from the definition of use cases I made some system tests, that focus their attention in these specific operations. The last use case, *Creation of a Virtual Scenario* will be validated with the work described in chapter 7.

T01 - Create a Virtual Appliance

The definition of a Virtual Appliance requires four parameters: a name, an image, a set of virtual resources, and a set of security groups. The test consist of four attempts of creation:

1. The insertion of a new VA with correct parameters.

2. The insertion of a new VA with the same virtual resources set but different image.

3. The insertion of a new VA with exactly the sames parameters of the previous case.

4. The creation of a new VA with a wrong parameter.

In each case we should obtain a different result. In the first case we expect the IM creates normally the new VA. In the second case, the IM should reuse the virtual resources set created in the previous phase (we should see the same identifier). The last two tests should not create a new VA, since in the first case the IM should return the already created VA, and in last one we must obtain a Infrastructure Manager Exception.

Running the test I obtain 2 VAs with the parameter that I have inserted. The IM has created a Flavor in the cloud that specifies the requested virtual resources. We can see the results:

Listing 5.2: T01 allocated virtual resources

```
+---+-----------+-----+----+----+----+----+------+------+-----+
|ID |     Name  | RAM |Disk| Eph|Swap|Vcpu| rxtx |Public|extra|
+---+-----------+-----+----+----+----+----+------+------+-----+
| 1 |InfManager1| 100 | 1  | 0  |    |  1 | 1.0  | True | {}  |
+---+-----------+-----+----+----+----+----+------+------+-----+
```

Moreover, the VA database has been populated with only 2 entries, even if we triggered four attempts. Since in the third attempt the request corresponds to an existent VM, the IM returns the identifier of the equivalent element. In this way I verified the IM does not introduces useless redundancy in the information base. In the last case we obtained an exception indicating which parameter was wrong. In the following listing (5.3) we can appreciate the two new VA referring to the same set of virtual resources.

Listing 5.3: VA database content

```
+----+---------+------+--------------------+----------+
| ID |  NAME   | VRID |       IMAGEID      | SECGROUP |
+----+---------+------+--------------------+----------+
| 1  | Test1.1 |  1   | 6306009a-a797-4108 | default  |
+----+---------+------+--------------------+----------+
```

T02 - Create a Virtual Network

The next step is to test the possibility of creating a new network in the cloud infrastructure. In this case we need to specify the name of the new network and the network address together with the number of bits of the network mask. The system should return the identifier of the element and we should see the new active network. The test will follow the next flow:

1. Create a new network with network address 10.10.10.0/24 named public.
2. Create a new network with the same parameter.
3. Create a new network with a wrong parameter.

In the former test we should obtain the identifier of the new virtual element and we should see the network active in the infrastructure. In the second case, the returned identifier should be the same as before, since the new network is equal to the former. Lastly we should obtain a Infrastructure Manager Exception.

Effectively the results correspond to what I suggested and we can appreciate the new Virtual Network is active. Listings 5.4 and 5.5 show the network to which is associated the subnet 10.10.10.0/24.

Listing 5.4: T02 network definition

```
+--------------------+--------+----------------------+
| id                 | name   | subnets              |
+--------------------+--------+----------------------+
| 20172d4d-84dd-4ac1 | public | 539786e7-0e14-4d64   |
+--------------------+--------+----------------------+
```

Listing 5.5: T02 subnet definition

```
+------------------+-------------------------------------------+
| Field            | Value                                     |
+------------------+-------------------------------------------+
| allocation_pools |{"start":"10.10.10.2","end":"10.10.10.254"}|
| cidr             | 10.10.10.0/24                             |
| dns_nameservers  |                                           |
| enable_dhcp      | True                                      |
| gateway_ip       | 10.10.10.1                                |
| host_routes      |                                           |
| id               | 539786e7-0e14-4d64                        |
| ip_version       | 4                                         |
| name             |                                           |
| network_id       | 20172d4d-84dd-4ac1                        |
```

```
| tenant_id          | bcb940507f64477784                          |
+--------------------+---------------------------------------------+
```

T03 - Start a Virtual Machine

After defining the two previous elements we are ready to start our first Virtual Machine. The new instance will be started from the previously created VA and attached to the network aforementioned. At the end of the process we should see an active Virtual Machine attached to the before mentioned LAN.

I will proceed to run following operations:

1. Create a VM with the aforementioned VA and virtual network.
2. Create a second VM introducing the same set of parameters.
3. Attempt to pass wrong values.

At the end of the process I expect to see two running instances that have the same characteristics but the identifiers. In the last test case I obtain an Infrastructure Manager Exception explaining the source of the fault.

In the following listing we can see the result of the test: two running Virtual Machines attached to the network that I have created before. Manually I tried to log in the first VM through a SSH connection and the result was successful.

Listing 5.6: T03 running Virtual Machines

```
+--------------------+--------------+--------+------------------+
| ID                 | Name         | Status | Networks         |
+--------------------+--------------+--------+------------------+
| aa3a4aee-05ed-42c5 | Test1.1459488| ACTIVE | public=10.10.10.4|
| 919174d9-adc9-4b45 | Test1.14e2917| ACTIVE | public=10.10.10.5|
+--------------------+--------------+--------+------------------+
```

T04 - Suspend and Resume a Virtual Machine

Once we have obtained a set of running VMs I try to test the *suspend* and *resume* functions. In this case I try to suspend the running Virtual Machines and verify their state. If the VM passed to the *"suspended"* mode I try to resume, in order to restore the *"active"* state.

Executing the test we can see that the state of the two VMs changes without problems, and the two instances pass to the suspended mode. With the resume function, VMs come back to the previous state. If the resume function is applied to a running machine, the IM returns an Exception.

Listing 5.7: T04 suspended Virtual Machines

```
+--------------------+-------------+---------+------------------+
| ID                 | Name        | Status  | Networks         |
+--------------------+-------------+---------+------------------+
| aa3a4aee-05ed-42c5 |Test1.1459488|SUSPENDED| public=10.10.10.4|
| 919174d9-adc9-4b45 |Test1.14e2917|SUSPENDED| public=10.10.10.5|
+--------------------+-------------+---------+------------------+
```

T05 - Change virtual resources Virtual Machine

One of the most interesting features of the Infrastructure Manager is the possibility of change the set of virtual resources to a VM. This characteristic may be very useful when a VM needs more capacity to provide its services (think about a web server with a temporary overload). In this case I will change the resource set, attaching to the *server* a *vRAM* of 512MB.

Executing the test I can see that the virtual machine has changed the set of virtual resources.

T06 - Stop a Virtual Machine

After all these tests there remains the last task: to turn off the virtual machine. I executed a double test:

1. I delete the two running VM that I created in T03.
2. I try to delete an nonexistent virtual instance.

In the former case, as supposed, the IM turns off the two running VM and resources are freed. In the second case, I receive a Infrastructure Manager Exception that informs me the inserted value is incorrect.

Mapping Between Use Cases and Tests

In the next table I resume the mapping between system tests and use cases:

Test Identifier	Use Case	Description
T01	UC04	Create a Virtual Appliance
T02	UC05	Create a Virtual Network
T03	UC01	Start a VirtualMachine from a VA definition
T04	UC03	Suspend and resume a VM
T05	UC06	Change virtual resources to a VM
T06	UC02	Stop a VM

Table 5.2: Integration tests mapped to use cases

5.3. Metrics and Statistics

Including the code developed within the body of the thesis was considered inappropriate. It is however important to consider, both in terms of effort in terms of quality, the performed work; for this purpose we have used a set of metrics showing some characteristics that describe the effective quality of the implemented code. Firstly I report the general statistics of the Infrastructure Manager implementation:

Besides the amount of code, it is also interesting to estimate its quality through specific metrics. Below I report a selection of the metrics considered most relevant for object-oriented development [49]:

- Cyclomatic Complexity
- Weighted methods per class
- Efferent couplings
- Lack of cohesion in Methods
- Number of levels
- Number of fields
- Number of parameters

Next we will provide more details about each of the selected indexes and analyze the graphs of the results of these tests applied to our code.

Packages	23
Classes	103
Functions	602
Lines of code	3964
Test functions	84
Lines of test code	1140

Table 5.3: General statistics

5.3.1. Cyclomatic Complexity

This metric is an indication of the number of 'linear' segments in a method (i.e.sections of code with no branches) and therefore can be used to determine the number of tests required to obtain complete coverage. It can also be used to indicate the psychological complexity of a method.

A method with no branches has a Cyclomatic Complexity of 1 since there is

1 arc. This number is incremented whenever a branch is encountered. In this implementation, statements that represent branching are defined as: 'for', 'while', 'do', 'if', 'case', 'catch' etc. The sum of Cyclomatic Complexities for methods in local classes is also included in the total for a method.

Cyclomatic Complexity is a procedural rather than an object oriented metric. However, it still has meaning for this kind of programs at the method level.

In our specific case the cyclomatic complexity exceeds the threshold only in the implementation of the Infrastructure manager and precisely in the method in charge with the creation of VM since in this phase the system needs to do several checks on the validity of the parameters.

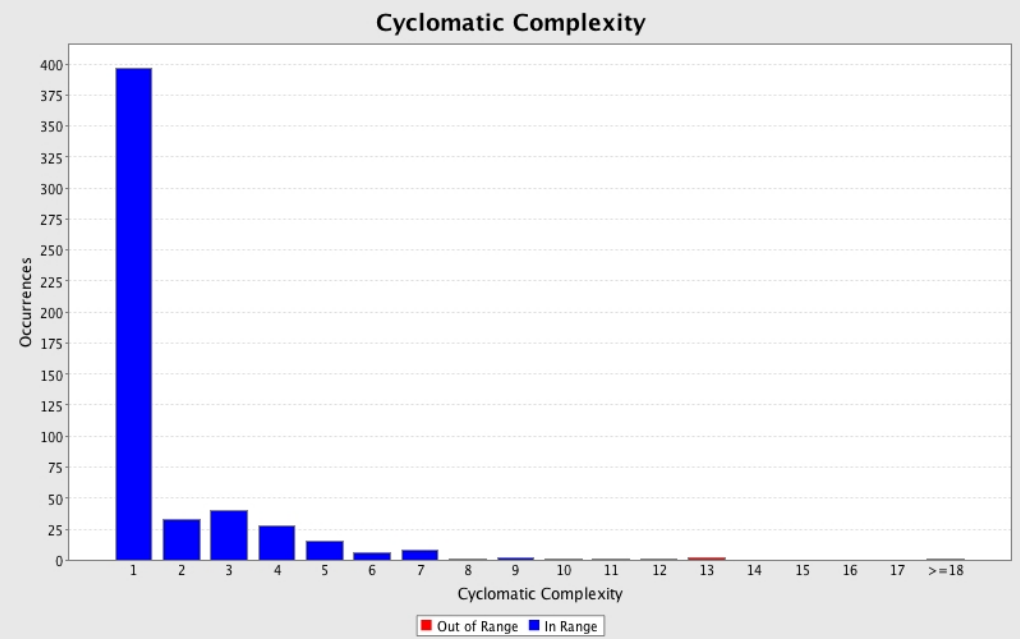

Figure 5.3: Cyclomatic complexity

5.3.2. Weighted Methods per Class

This metric is the sum of complexities of methods defined in a class. It therefore represents the complexity of a class as a whole and this measure can be used to indicate the development and maintenance effort for the class. In order to reduce this value, classes with a large Weighted Methods Per Class can often be split into two or more classes.

According to the previous metric, the most complex class is the Infrastructure Manager Implementation which interacts with the underlying level of managers. In

this class resides highest complexity, since, as designed, this is the conductor of the entire system.

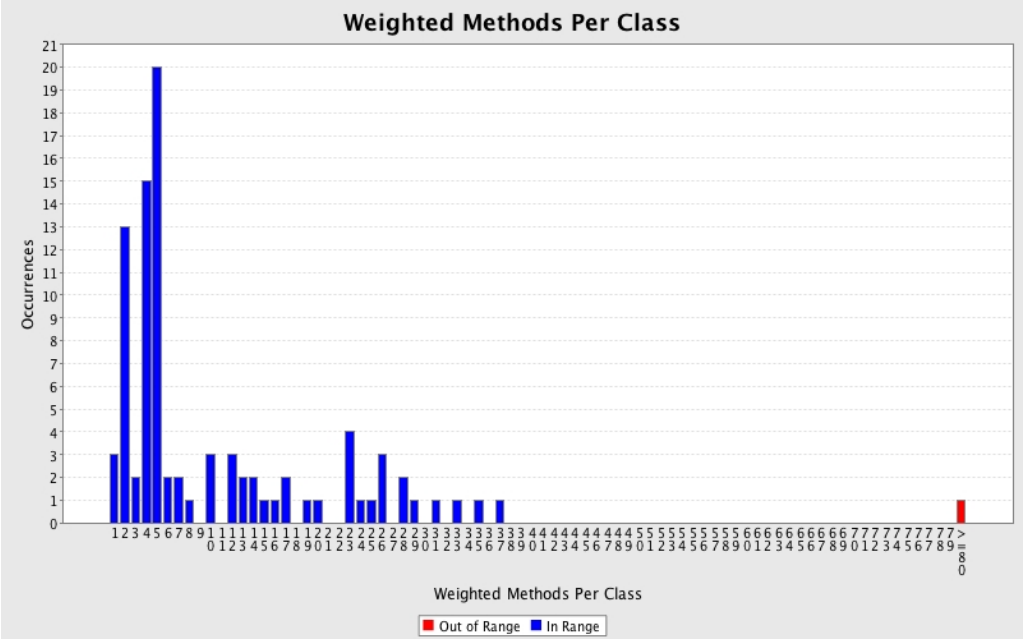

Figure 5.4: Weighted methods per class

5.3.3. Efferent Couplings

This metric is a measure of the number of types the class being measured 'knows' about. This includes: inheritance, interface implementation, parameter types, variable types, thrown and caught exceptions. In short, all types referred to anywhere within the source of the measured class.

A large efferent coupling can indicate that a class is unfocussed and also may indicate brittleness, since it depends on the stability of all the types to which it is coupled. When used to measure couplings between packages (coming in a later release) this measure can be used together with others to calculate 'abstractness', 'stability' and 'distance from the main line'.

A typical way to reduce this value is extracting classes from the original class decomposing it into smaller classes.

In our system the higher value belongs to the Infrastructure Manager Implementation since it must cooperate with all managers. This correlations makes high the value of *Efferent couplings* for this class. On the other hand, since each module is very specialized, the majority of classes has low values.

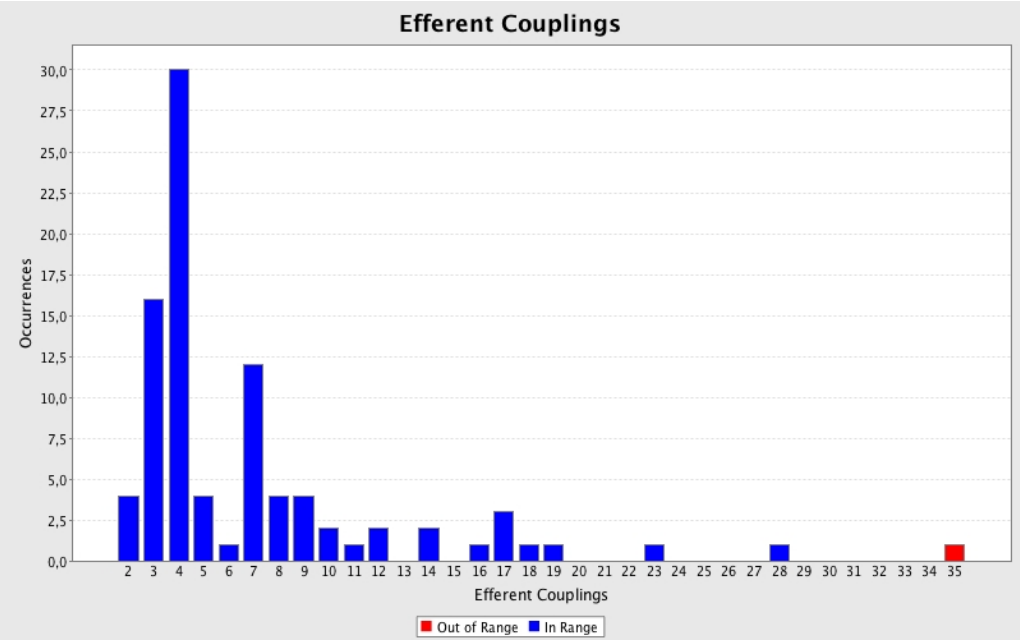

Figure 5.5: Efferent couplings

5.3.4. Lack of Cohesion in Methods

Chidamber and Kemerer define Lack of Cohesion in Methods as the number of pairs of methods in a class that don't have at least one field in common minus the number of pairs of methods in the class that do share at least one field. When this value is negative, the metric value is set to 0.

According to the definition, classes that have a high level of lack of cohesion are the defined POJOs or rather the information model classes of the *Quantum Client*: in this classes I defined only attributes and auxiliary methods (setters and getters) which never access to more than one attribute at a time. This is the reason why they have an high level of lack of cohesion, but in this case the value is not important, since it is only due to the method of calculation. POJOs correspond to the highest values of the graph. For the rest, from the graph we can notice that the cohesion of the system is high.

5.3.5. Number of Levels

This metric is an indication of the maximum number of levels of nesting in a method. The idea of the metric is that a large Number Of Levels increases complexity and reduces comprehensibility. In addition, such methods generally (but

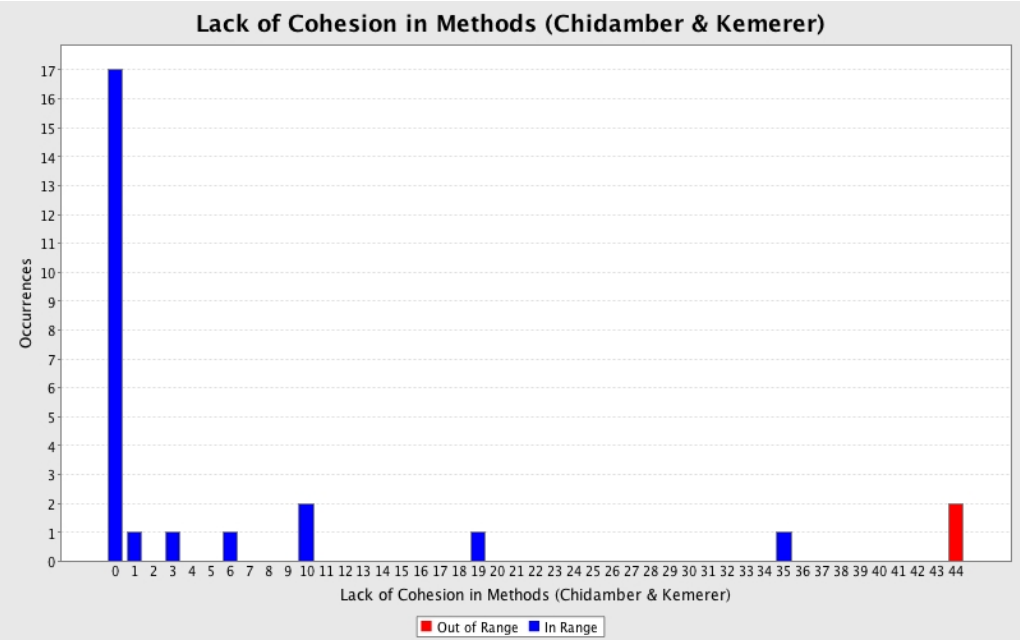

Figure 5.6: Lack of cohesion in Methods

not necessarily) operate at the lowest level of abstraction or at mixed levels of abstraction, both of which contribute to the confusion. All methods that have a large Number Of Levels can be simplified by extracting private methods or by creating a Method Object. Both of these result in naming of the different parts of the original method, thus raising the level of abstraction.

The VirtualApplianceManager class has the higher level of levels of nesting (5), followed by the InfrastructureManagerImplementation and CSV (implementation of the Consistency interface) classes (4). Certainly we can find the source of this result in the high number of parameter checks carried out in the aforementioned classes; each of these tests is translated in a new level of nesting and in the worst case we obtain a value of 5, which is located just above the threshold.

5.3.6. Number of Fields

This metric represents the number of fields in a class. Although a large number of fields is not necessarily an indication of bad code, it does suggest the possibility of grouping fields together and extracting classes, taking appropriate methods as well. This gives the group of fields with their associated operations a name thus improving the semantics of the object model and a better distribution of system intelligence.

In our case the number of fields never goes beyond the threshold: the higher

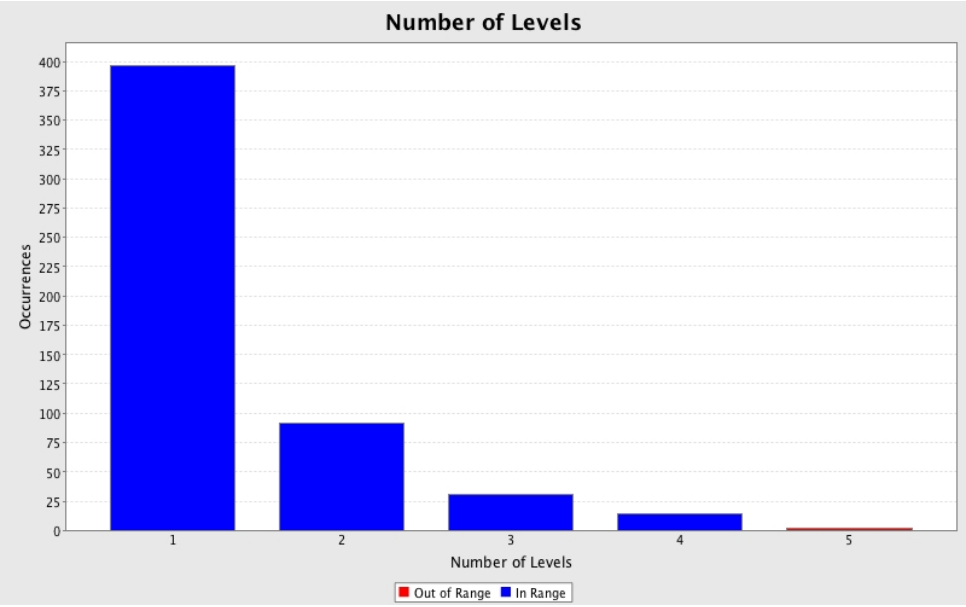

Figure 5.7: Number of levels

value is related to POJOS classes that are used in the communication with the Quantum controller. In general, the majority of classes is composed predominantly of methods and the system maintains an acceptable level of attributes per class.

5.3.7. Number of Parameters

This metric is a count of the number of parameters to a method. Methods with a large Number Of Parameters often indicate that classes are missing from the model. Most methods that have a large number of parameters can be simplified by grouping parameters into sets and making classes from those sets. This leads to increase the maintainability.

In our case the the Infrastructure Manager is composed by functions that receive a small number of parameters. This value is a bit misleading, since usually parameters are structured objects that incorporate a grater set of elements.

5.3.8. Conclusions

As evidenced by the previous set of graphs, the code maintains a good level of quality in all the analyzed characteristics, giving the work an additional value. In this way it is presumed that the maintenance of such system should not introduce many problems, either at the time to integrate it with other systems, either at the time to change it in order to adapt it to other cloud solutions.

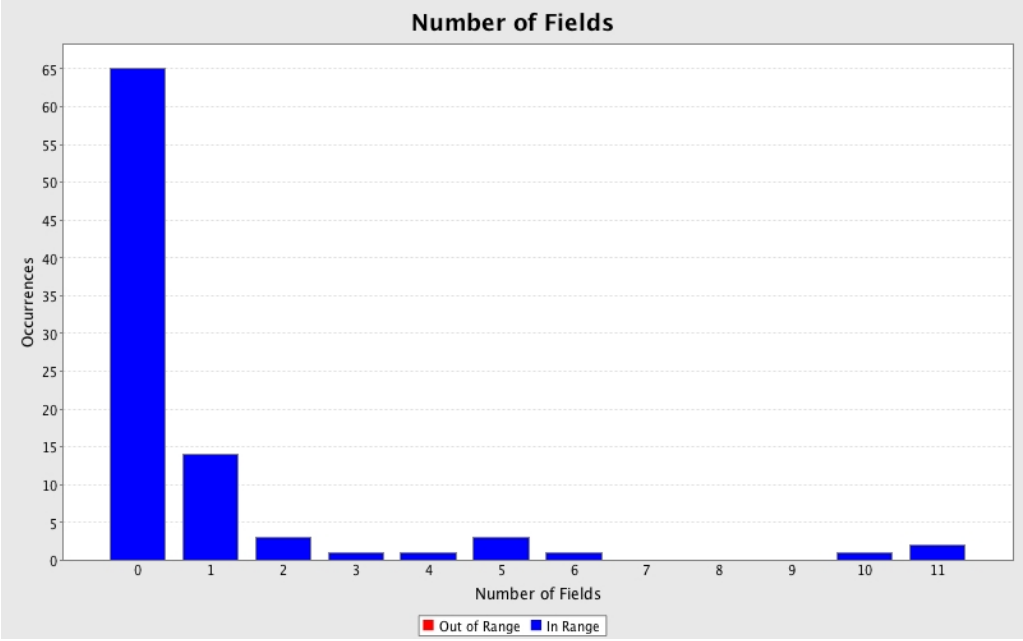

Figure 5.8: Number of fields

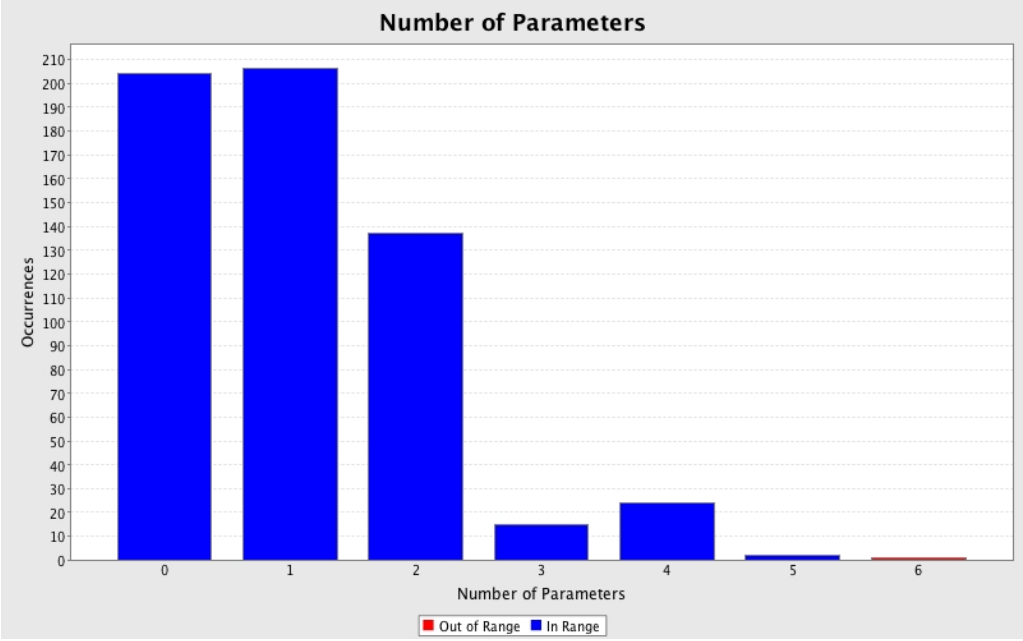

Figure 5.9: Number of parameters

Chapter 6

OpenStack Experience and Configuration Troubleshooting

Since the installation and deployment of the Openstack infrastructure has resulted to be more complex than planned, I have decided to devote a chapter to show which were the troubles that slowed the work.

6.1. Operational Problems

First of all I must mention the incomplete alignment between the code and the documentation files that caused diversions from the correct implementation. Following the provided official guide I encountered many faults that fortunately had already been treated and most of times solved in mailing lists of the developer's community.

The *launchpad.net* web site in real practice has been of paramount importance in solving specific problems and support, since at a given time it has been necessary to interact actively to resolve a configuration error that was not documented.

Our first successful attempt consists in the installation of the Openstack Folsom release, with the default network managing service *Nova Network* (deprecated in next version, due to the advent of the new standalone network manager Quantum). I choose to use the default hypervisor KVM even it was not the only suitable.

The installation has not been too problematic but the correct configuration of the computational service has been a path made of continuous tests and changes into the configuration files. A specific scenario needs a specific configuration that could not be standardized; so the definition of such file (nova.conf) led me throughout developers mailing lists and forums till the generation of the configuration set available in the appendix of this document (Appendix A.1).

The testing of the infrastructure saw the installation of several releases (stable

and not), which had implied the removal of the previously installed software. In the Openstack documentation no section is dedicated to the uninstalling process that, unfortunately, does not ends with a simple Linux *purge command*. When the admin decides to remove OpenStack, he has to take care about some hidden OpenStack residual, which usually raises error in a future installation of the IaaS.

It has occurred many times to redeploy the computational service. This operation has led to some problems caused by residual elements leaved by the previous deployment. As I mentioned, each service works with a database (in our case a *mysql* database) that should be deleted or cleared manually. Unfortunately, the database is not the only residual element the service installation leaves. Sometimes the compute service may crash leaving a headless instance in the cloud controller: these files create cases of conflict, since, in the new deployment, the enumeration of VM's restart from the beginning. In this case, on booting a new VM, the new instance may end in the *error* state and in the Nova compute service log file will appear error messages that refers to an already existing VM. I solved the problem using with a script that deeply cleaned the system and recreated the database (available at appendix A).

The second big problem I encountered concerns the metadata service. If enabled in the OpenStack configuration, a VM on booting can contact a metadata server (for default set at 169.254.169.254) in order to retrieve some additional files needed for the initial instance configuration (i.e. the injection of the SSH key pair). Even if in the documentation it was not mentioned, metadata forwarding must be handled by the gateway, and since the Openstack *Nova* service does not do any setup in this mode, it must be done manually. All requests to 169.254.169.254 port 80 will need to be forwarded to the API server. For this I inserted a set of *iptables* rules in order to permit VMs to reach such service (iptables rules are available in the appendix A.4). After that VMs could contact the metadata server and a SSH connection with public and private key authentication could be performed.

The umpteenth problem I encountered concerns a feature that from our point of view is very important: the possibility of choosing the physical machine on which the new VM will run. At the exit of Folsom release, the documentation still relied primarily based on the previous version and the above commands did not get the required result.

This was a true case of no documentation, since in the transition to the new version these commands have been changed radically and in spite of the guide bore the name of the new release, it had not been completed yet. An important note refers to the elected scheduler: if you need this kind of functionality you need to specify in the configuration file the use of the *filter scheduler* a specific scheduler that allows to indicate this kind of parameters. Even in this case the answer was given by one of the developers on the Launchpad web application.

The documentation also omits that in order to enable the possibility of resize

VMs resources the migration service must be configured, even if the *resize_to_the same_host* flag is set to true in the *nova.conf* file. This operation requires the creation of a shared repository accessible through the NFS protocol.

Fortunately, the documentation has improved with the updating process[1]. Finally I remark that in a multihost deployment scenario all compute nodes must be aligned with their configuration and all of them must run the *Nova Network* service.

6.2. The Quantum Service Installation

Since the network customization capabilities offered by the *Nova Network* service were so limited, I planned the changeover to the new standalone and full of prospective network service *Quantum*. Unfortunately the transition has not been so friendly.

After the release of the last stable release I decided to deploy the recently introduced module expanding the already installed system so as to take advantage of all the new introduced features. The Quantum project was created to provide a rich and tenant-facing API for defining network connectivity and addressing in the cloud. The Quantum project gives administrator the ability to leverage different networking technologies to power their cloud networking.

However, with regret I noticed the new module still has significant limitations. In order to support overlapping IP addresses between virtual networks, the Quantum DHCP and L3 agents use Linux network namespaces by default. But Quantum overlapping IPs do not work with *Nova Security Groups* or *Nova Metadata Server* that is considered fundamental for our system. As a result I have been forced to not use the Linux namespaces so that I can exploit the two services mentioned above. Limited by these aspects I configured Quantum to use tunnels and not vLAN since our switch can not bear this feature.

Following the installation guide of the official webpage, I got a completely inoperative system, since the Nova Compute service that I configured to work with the new network module, had broken. The error was given by a specific driver that could not be loaded, even if the documentation was clear over this. I changed the parameter and I managed to get the nova service running.

Then I tested the service obtaining an unexpected error: virtual machines were not receiving the network configuration since the XML template, passed by the compute service to libvirt, was getting populated with an empty network bridge field. Taking advantage of the launchpad I asked the community that focused the problem: the Nova compute service was not loading the drivers for virtual network

[1]The history of each document can be accessed at the specific section *"Document Change History"*

interfaces (VIF). Through the OpenStack developers mailing list[2] I found the correct parameter and subsequently I solved a typo present in the documentation that made the parameter not working and the system crash (Errors and correction available at appendix A.2)

After solving the issue I experienced another problem: the l3_agent was continuously reporting an error in its log due to the impossibility of defining an iptables rule. This error had been classified as a system bug and reported in a page[3] within the launchpad web application. The bug had been fixed in the new (temporary release) of Openstack Grizzly-"g1"[4] and changes are pending to be uploaded to the Ubuntu Quantal release. So, even if using an unstable version is not advisable, I changed the Quantum module with the expectancy of resolving the problem, and indeed I managed to get a working system where VMs could obtain the correct Network configuration.

When I tested the developed Quantum Client I noticed that the interchanged messages between Java client and Quantum API differ from the API reference[5]. This occurrence made me readapt the developed code to the real message protocol. But a more significant problem arose. Testing the developed client I reached the conclusion that the transition between Nova network and Quantum had left some problems in the Nova service.

6.2.1. A suggested Openstack Improvement

While testing in a more comprehensive way the new infrastructure, I fell into an integration problem that hinders our programmatic interaction with the cloud infrastructure. In the Nova Network service only two networks are defined as fixed and no others names could be used, their names are *private* and *public*. This feature is a bit restricting, but now the matter refers to the message protocol: when requesting a VM's state information, the cloud controller returns a JSON message describing the instance. In the *addresses* object (see listing 6.1) I find some JSON elements referenced by a key which value is the name of the network.

Listing 6.1: Addresses' part of a Nova service's message.

```
{
  "addresses": {
      "private": [{"version": 4, "addr": "10.1.1.6"}],
```

[2]https://lists.launchpad.net/openstack/msg18632.html

[3]https://bugs.launchpad.net/quantum/+bug/1069966

[4]https://launchpad.net/quantum/+milestone/grizzly-1

[5]http://docs.openstack.org/api/openstack-network/2.0/content/

```
        "public": [{"version": 4, "addr": "192.168.0.134"}],
        "lan02": [{"version": 4, "addr": "10.1.22.3"}],
        "lan01": [{"version": 4, "addr": "10.1.11.3"}]
    }
}
```

As long as the names of the networks were fixed there was no problem because the values of the message could be easily accessed in programmatic form. But with the arrival of Quantum, the names of the networks have started to be variable, since the user can choose any value. This substantial change would have to make desist to continue to use this parameter as a key value since by now reference values are not fixed and more difficult to access in an automatic form.

In other words, as it is now, the fragment of the JSON response under *addresses* does not conform to a fixed JSON schema. So each time a network name changes, also the JSON message template changes. I found this problem while trying to parse this message programatically. As it is structured now, a programmer can not build an object able to represent the *addresses* element, since every network, inside it, is parsed as an element of a different type, with the name of the network used as the type.

The problem is not syntactic, but semantic and I think it must be fixed. It seems that in all other messages of the API the key is fixed and this is the first case of dynamic keys. My suggestion is: don't use network names as key, but specify them as values. This improvement will ease a lot the programmatical interaction between nova and an external client. In this way programmers can map the JSON object to, for example, a Java object, regardless the name of the network. A solution proposal is offered in listing 6.2 and a bug report had been opened in Launchpad[6].

Listing 6.2: Possible solution for Nova service's message.

```
The message is conform to a fixed JSON schema that does not
   matters the name of the network.
{
"addresses":  [
   {"network": "private", "version": "4", "addr": "10.1.1.6" },
   {"network": "public", "version": "4", "addr":
      "192.168.0.134"},
   {"network": "lan02", "version": "4", "addr": "10.1.22.3"},
   {"network": "lan01", "version": "4", "addr": "10.1.11.3"}
  ]
}
```

[6]https://bugs.launchpad.net/nova/+bug/1084560

6.2.2. General Advice

I end this section with a series of useful tips when you want to deploy a cloud infrastructure based on OpenStack. First of all be sure to work with updated systems, which satisfy the OpenStack's requirements: it is necessary to pay attention to the *libvirt* version, since some configuration settings vary according to this parameter.

We must not forget the importance of debugging tools that the software offers, since they are the main source of information about what's going on. Be sure that you have configured authentication parameters of all modules that access the API (Glance, Nova, Quantum...). These values must be configured on the specified file *api_paste.ini* and authentication through Keystone (when used) must be specified.

Sometimes, due to some problems it is necessary to clean up the file system from residues of instances that no longer exist. Cleaning up the database, making sure the correct repopulation, can sometimes solve problems of inconsistency.

For any other problem, I suggest you contact the OpenStack community that will surely lead you to a correct configuration of your infrastructure.

Chapter 7

Virtual Networks over Openstack - VNO

The last part of the development cycle covers the creation of the Virtual Networks over Openstack service (VNO). The name has been suggested by the Virtual Networks over linuX project (VNX), an open-source virtualization tool designed to help building virtual network testbeds automatically. Within this chapter I try to define a possible service architecture that could be implemented using the previously described system.

7.1. Virtual Networks over linuX - VNX

VNX is an open source project developed by the Telecommunication and Internet Networks and Services (RSTI) research group of the Telematics Engineering Department (DIT) of the Technical University of Madrid (UPM).

VNX is a tool for testing network applications/services over complex testbenchs made of virtual nodes and networks, as well as for creating complex network laboratories to allow students to interact with realistic network scenarios [50]. It is composed by two main parts: one is a VNX specification language (based on XML) used to describe the virtual network scenario, including virtual machine and virtual networks specifications. The other is the VNX program that analyses the specification file and creates the virtual environment. This program could be installed over a single server, but it comes with a distributed version that allows the deployment of virtual scenarios over clusters of Linux servers.

A recent publication [39] affirms that in this situation, cloud technologies seem to be a natural environment for these scenario-based tools to work with in the future. Some other similar tools provide a minimal integration with clouds technologies, as they allow connecting the virtual scenarios created with virtual machines deployed

over private or public clouds. However, none of them are able to deploy a complete virtual network scenario over a cloud infrastructure. Moreover, cloud technologies do not provide in general the necessary primitives to allow the user to define the topology.

But with the deployed *Infrastructure Manager* and the capabilities of the OpenStack infrastructure this prevision may become a reality.

I will not enter into the details of the VNX software since it is not the aim of this work.

7.2. The VNO Service

The idea is to integrate the VNX service with the deployment of virtual machines over OpenStack. The new service would be able to analyze the VNX specification file and deploy the virtual network scenario over the cloud using primitive functions offered by the *Infrastructure Manager*. This is an additional element that will validate the previous work, offering a possible new use case and providing a starting point for a possible future project. The VNO service will benefit from all cloud characteristics presented in the former chapter, and in particular from the distribution of resources over the infrastructure.

7.2.1. The Architecture

Since this is a prototype, the service architecture is simple but extensible. The structure is composed of four main blocks that are installed above the infrastructure manager system (figure 7.1). The *Service* block, an implementation of the *Service Interface*, is the main module that analyzes and starts the virtual scenario. The *Parser* is in charge of the unmarshalling of the configuration file, returning handleable objects to the service module. The last module is the *Connector* or rather the link between the VNO service and the *Infrastructure Manager*. Even in this case the architecture follows a hierarchical design where the *Service* module acts as root.

The Connector

I will start from the lower layer or rather the *Connector*. This is the linker to the Infrastructure Manager that enables the deployment of the virtual scenario. It calls the IM's primitives in order to start all virtual instances specified by the higher levels.

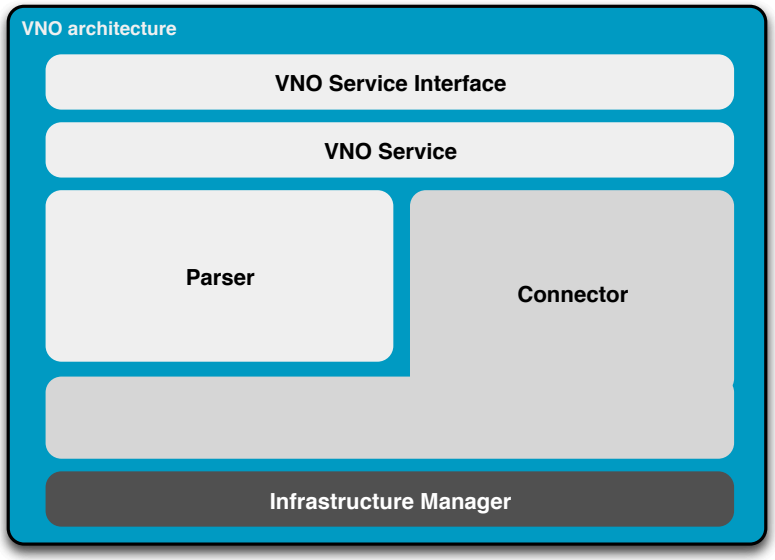

Figure 7.1: VNO architecture

The Parser

The parser is the element responsible of unmarshalling the specification file. The VNX specification file contains a set of XML tags that define the characteristic that VMs should have. Now, the level of customization of virtual instances offered by VNX is considerably lower than that offered by the IM; for this reason a default adaptation must be done. With default adaptation I mean that many variables that are required to start a new VM are not specified in the VNX XML file and for this reason they must be set as default value. In this version of the code, the parser just focus on VMs and networks specifications: a future implementation, accordingly with a proper IM improvement, may lead to an exhaustive and complete system able to add new features to VNX standard.

The Service Implementation

The Service implementation is the working center of the service since it receives and manages the set of objects that compose the scenario. In listing 7.1 we can see an example of virtual machine specification. We can notice that a VM is defined by a name, an operative system, a file system type, an amount of memory and by a list of network configurations where the interfaces of the VM are defined. It is obvious that a lot of parameters are missing if we want to deploy this element over the cloud.

Once obtained these values, the *Service Implementation* calls the *Connector*

passing the retrieved information. The *Connector* will contact the Infrastructure Manager that will deploy the VM. With more than one VM definition, the system works sequentially treating all specifications once at a time. With just one element the example may seems trivial, but in appendix B a more exhaustive example is provided.

Naturally it is important to remake that this system benefits of the IM service that ease substantially the deployment task. In figure 7.2 we can see the graphical representation of the workflow.

Listing 7.1: Example of VM in the VNX specification file (partial)

```
<vm name="h2" type="libvirt" subtype="kvm" os="linux">
  <filesystem type="cow">/usr/share/vnx/filesystems/
     rootfs_ubuntu</filesystem>
  <mem>128M</mem>
  <if id="1" net="Net0">
    <ipv4>10.0.0.3/24</ipv4>
  </if>
  <route type="ipv4" gw="10.0.0.1">default</route>
</vm>
```

7.2.2. The Test

At the end of the development we tested the service over workbench passing to the service the VNX configuration file retrieved online in the VNX documentation[1]. Such configuration file is available in the appendix B. An important VNO limitation is the impossibility of create virtual routers since the Openstack platform now is not providing this feature when the metadata service is enabled. For this reason we change partially the XML file, even if in VNX virtual routers are virtual machines with a linux OS simulating the router behavior.

At the end of the execution we can found in the cloud controller the following result (listing 7.2). A graphical view is offered in figure 7.3.

Listing 7.2: Example of VM specification in the VNX specification file

```
+--------------------------------------+------+--------+--------+
| ID                                   | Name | Status |Networks|
+--------------------------------------+------+--------+--------+
| fe82f5ba-ffd8-4fbb-8f25-6f1a2f3ab2fa | h1   | ACTIVE | Net0
   =10.0.0.3 |
| e05ef7bd-bcbd-4b55-a4ba-46c30bb9867b | h2   | ACTIVE | Net0
   =10.0.0.4 |
```

[1]Available online at http://web.dit.upm.es/vnxwiki/index.php/Vnx-tutorial-ubuntu

```
| aff66a55-b1c8-492c-976c-ebc4b4341cac | h3   | ACTIVE | Net2
   =10.0.2.4 |
| e9314b4a-aa2d-40a9-bb17-8b1b99fb120c | h4   | ACTIVE | Net2
   =10.0.2.3 |
+--------------------------------------+------+--------+--------+
```

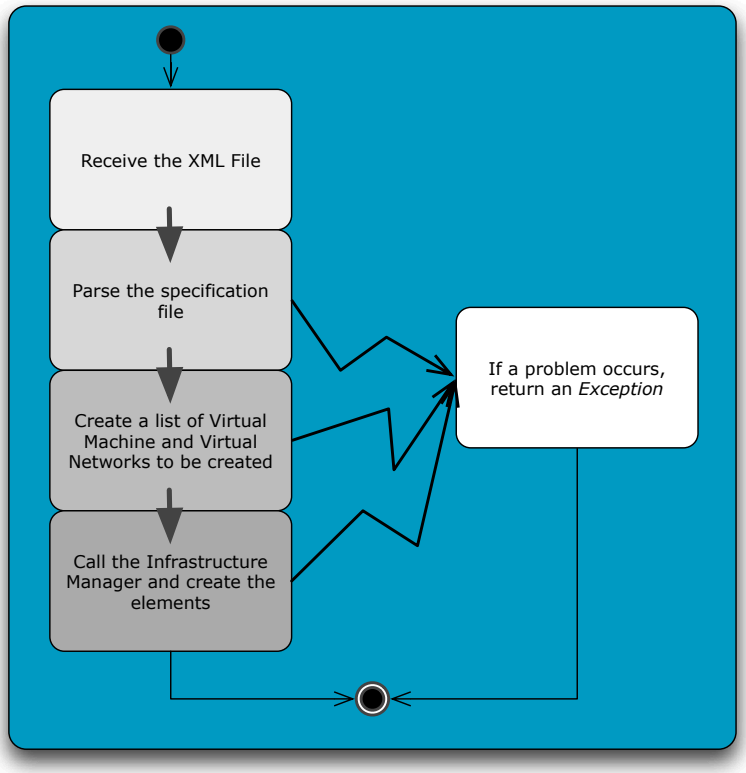

Figure 7.2: VNO service activity diagram

How we can see in the figure 7.3, the scenario is still incomplete: it lacks a virtual router able to route the network traffic outside the cloud. This improvement can be reached only when a stable release of the Openstack Network Service could provide the possibility of virtualize this typology of devices without restrictions.

Conclusions and Possible Extensions to the Project

This is only the starting point of a new project that does not make part of this master thesis. Significant improvements may be made, with the aim of creating a system able to deploy advanced network topologies, reaching and surpassing the actual level of the tool.

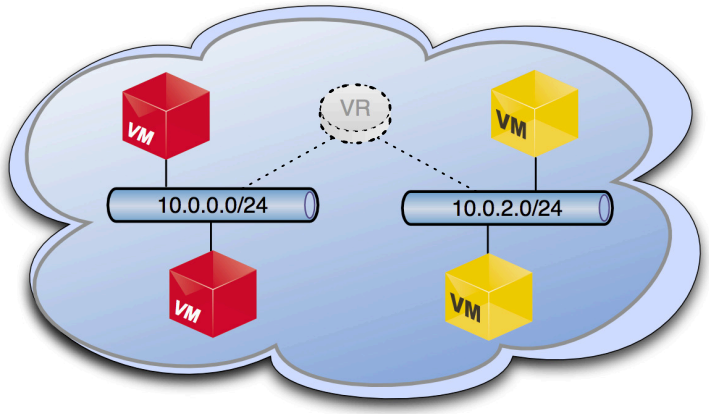

Figure 7.3: VNO service test result, the creation of a Virtual Router is the future goal and it is not still present

We can notice the XML file contains more parameters than those handled by us. An example is the *"exec"* tag. This label brings together a series of commands that are executed on the virtual machine in the ignition phase. This concept seems to find a natural mapping with the metadata service of OpenStack where the user can pass to the virtual machine a script or a initial configuration file that is executed on booting.

VNX allows the user to load a file system tree into the new VM. Even in this case this feature sees a similar service in the Openstack volumes service where the user can create new storage volumes and attach them to the running instance. By automating this process we could create this feature also in VNO.

Another possible improvement is the introduction of different access interfaces such as REST or web interfaces improving the accessibility of the service.

Up to now we have tried to adapt the VNX format to the new system. Another idea is to change slightly the standard used for the configuration file, while keeping backwards compatibility, but inserting new elements that could fit with the new features: these new features refers to the possibility of choosing adopted OS image for each VM, or the computing power allocated to each instance.

Also an improvement is to provide mapping policies that guide the process of allocating VMs to specific *Hosts* over the cloud.

These are only a small set of characteristics that describe the great potentialities of the new Virtual Networks over OpenStack service.

Chapter 8

Results and conclusions

The aim of this chapter is to review the results by extracting appropriate conclusions and pointing possible future research lines.

To do this, I begin with a review of the goals set at the beginning of the project, to analyze their fulfillment. After that I summarize some of the limitations and difficulties encountered during development and finally, I enumerate some possible future lines of work.

8.1. Results

During this *Master Thesis* I have established the need for a management architecture for private IaaS clouds able to support several solutions (to avoid data and vendor lock-in), working alone or together, and several user interfaces. To this end I have decided to use of a generic information model developed inside the research group that captures all the relevant information for the infrastructure, and a modular architecture that can be adapted to fit several IaaS products and needs.

To validate my approach I have developed and tested a sample implementation with support for one cloud solution (OpenStack). In previous chapters I have described the details of this implementation, analyzing step by step each component of the developed system.

The IM (*Infrastructure Manager*) allows the deployment of VMs over a private cloud infrastructure, leaving open the possibility to use a different solution, and allows the creation of virtual network scenarios in an automated, albeit limited, way.

I have put my work trough test using a set of specific cases that verify the correct behavior of the system. Additionally, I have implemented an additional service (VNO), which uses some services provided by the *Infrastructure Manager*.

8.2. Conclusions

The completion of this *Master Thesis* has put me in an environment where I have been able to strengthen my knowledge and learn many concepts in the fields of software development, software systems, network integration and web services.

During the development of the project I have improved my skills and knowledge in the relatively new and potentially important field of cloud computing. Also, I've been able to practice and improve my abilities in the use of various tools and technologies, like integrated development environments, virtualization tools, cloud platforms and application servers.

Finally, this work has allowed me to deepen my knowledge about the Java programming language, especially with web services technologies. More importantly, I have managed to familiarize with the OpenStack environment and all its services.

8.3. Future works

During the developing period of the system different ideas for new features and improvements have emerged. Unfortunately, not all of them could be developed in the time frame of this work. In this section I describe some of these ideas for future work, which were not incorporated in this thesis due to the limited timescale and resources.

The first future aim is to develop support for at least another cloud solution: This way we will be more able to test a federation of several private clouds and achieve a true working common management interface for multiple technologies. Related to this, another possibility for improvement would be the implementation of more user interfaces, which would allow the access to the service through the use of more technologies.

The implementation of the VNO service still needs a lot of effort in order to reach an adequate level of customization of the scenarios. This objective is tied with the improvement of the OpenStack IaaS network features, especially its system for managing virtual networks. The level of integration between the two services (VNX and VNO) can be greatly improved, but the amount of work required for this might represent another *Master Thesis* in on itself.

I have also left as future work the integration of the *Infrastructure Manager* with a reasoning module able to automate the administration of a private cloud platform, thus completing the creation of a full autonomous manager for private cloud infrastructures.

Finally, the last line of work I would like to follow is the application of the presented architecture to the management of public cloud offerings, since the interest in hybrid clouds that mix public and private IaaS is growing steadily.

Appendix A

Configuration example for OpenStack

A.1. File nova.conf with nova network

In this appendix I propose the set of configuration files used during the development of the *Infrastructure Manager*. Since one of the most troubled phases had been the OpenStack deployment, I attach the configuration set I used, in order to assist the reader in case he wants to deploy its own IaaS.

Listing A.1: nova.conf

```
[DEFAULT]

#LOGSTATE
verbose=True
logdir=/var/log/nova
state_path=/var/lib/nova
lock_path=/var/lock/nova

#VIF
libvirt_vif_driver=nova.virt.libvirt.vif.
   LibvirtHybridOVSBridgeDriver
libvirt_ovs_bridge=br-int
libvirt_vif_type=ethernet

#SCHEDULER
scheduler_driver=nova.scheduler.multi.MultiScheduler
volume_scheduler_driver=nova.scheduler.chance.ChanceScheduler
compute_scheduler_driver=nova.scheduler.filter_scheduler.
   FilterScheduler
scheduler_available_filters=nova.scheduler.filters.
   standard_filters
```

```
scheduler_default_filters=AvailabilityZoneFilter,RamFilter,
    ComputeFilter

#NETWORK
dhcpbridge_flagfile=/etc/nova/nova.conf
dhcpbridge=/usr/bin/nova-dhcpbridge
force_dhcp_release=True
firewall_driver=nova.virt.libvirt.firewall.IptablesFirewallDriver
linuxnet_interface_driver=nova.network.linux_net.
    LinuxOVSInterfaceDriver

#QUANTUM
network_api_class = nova.network.quantumv2.api.API
quantum_url = http://192.168.0.1:9696
quantum_auth_strategy = keystone
quantum_admin_tenant_name = service
quantum_admin_username = user
quantum_admin_password = password
quantum_admin_auth_url = http://b205b2.dit.upm.es:35357/v2.0

# Change my_ip to match each host
my_ip=192.168.0.1

#VOLUMES
volumes_path=/var/lib/nova/volumes
iscsi_helper=tgtadm

#COMPUTE
libvirt_use_virtio_for_bridges=True
connection_type=libvirt
libvirt_type=kvm
compute_driver=nova.virt.libvirt.LibvirtDriver
root_helper=sudo nova-rootwrap /etc/nova/rootwrap.conf
allow_resize_to_same_host=True

#API
ec2_private_dns_show_ip=True
api_paste_config=/etc/nova/api-paste.ini
multi_host=True
enabled_apis=ec2,osapi_compute,osapi_volume,metadata
metadata_host=192.168.0.1
metadata_port=8775
dmz_cidr=192.168.0.1/32

#DATABASE
sql_connection=mysql://<user>:<password>@192.168.0.1/nova

# GLANCE
image_service=nova.image.glance.GlanceImageService
```

```
glance_api_servers=192.168.0.1:9292

#MESSAGES
rabbit_host = 192.168.0.1

#AUTHENTICATION
auth_strategy=keystone

[keystone_authtoken]
auth_host = 192.168.0.1
auth_port = 35357
auth_protocol = http
auth_uri = http://192.168.0.1:5000/
admin_tenant_name = service
admin_user = user
admin_password = password

#MIGRATION
vncserver_listen=0.0.0.0
live_migration_flag=VIR_MIGRATE_UNDEFINE_SOURCE,
    VIR_MIGRATE_PEER2PEER,VIR_MIGRATE_LIVE

#RESIZE
resize_confirm_window = 1
```

I will focus over some important values of the nova.conf configuration file:

- The authentication method must be set to "*keystone*".
- In the case the user needs to force the scheduling of VM over specific hosts (this is the case of the *Infrastructure Manager*), he needs to specify the *filter scheduler* option since the default one does not provide this possibility.
- The *resize_confirm_window* avoids a confirmation message in the resize operation.
- If the user needs a multi-host scenario, the *multi_host* flag must be set with the *true* value. Over multi host scenarios the resize and migrate functions are enabled only if a common nfts folder is correctly configured.
- The data base may not necessarily reside in the same machine in which the cloud controller has been installed.
- The *libvirt_vif_driver* must be set according to the *libvirt* version.

The Quantum service has been deployed using *Open vSwitch - OVS* [51]. Using the OVS quantum plugin in a deployment with multiple hosts requires the using of

either tunneling or vlans in order to isolate traffic from multiple networks. This is not the only available choice.

Make particularly attention to the following parameters that presents errors in the official documentation.

Listing A.2: Driver configuration errors

```
Configuration parameters described in the documentation:

libvirt_vif_driver=nova.virt.libvirt.vif.
   LibvirtHybirdOVSBridgeDriver
compute_driver=libvirt.LibvirtDriver

Correct parameters:

libvirt_vif_driver=nova.virt.libvirt.vif.
   LibvirtHybridOVSBridgeDriver
compute_driver=nova.virt.libvirt.LibvirtDriver
```

A.2. Cleanup script

The *bash* script I propose ease the user to redeploy the OpenStack infrastructure without incurring in errors caused by residual elements of the previous installation.

Listing A.3: Script for cleaning the system from nova residuals: cleanup.sh ROOTPSW

```
#!/bin/bash

rm -f /var/log/libvirt/qemu/instance*
rm -f /etc/libvirt/nwfilter/nova*
rm -f /var/lib/nova/instances/instance-*
rm -f /var/log/libvirt/qemu/inst*
rm -f /etc/libvirt/nwfilter/nova-*

sudo mysql -uroot -p$1 -e 'DROP DATABASE nova;'
sudo mysql -uroot -p$1 -e 'CREATE DATABASE nova;'
sudo mysql -uroot -p$1 -e "GRANT ALL PRIVILEGES ON nova.* TO '
   nova'@'%';"
sudo mysql -uroot -p$1 -e "SET PASSWORD FOR 'nova'@'%' = PASSWORD
   ('nova');"
```

A.3. IPTables rules

If the metadata service is enabled, it is necessary to set a collection of iptables rules that routes the VM's requests to the controller.

Listing A.4: Ipitables rules for metadata

```
iptables -t nat -A PREROUTING -d 169.254.169.254/32 -p tcp -m tcp
    --dport 80 -j DNAT --to-destination <APISERVER>:8775

iptables -t nat -A OUTPUT -d 169.254.169.254/32 -p tcp -m tcp --
   dport 80 -j DNAT --to-destination <APISERVER>:8775
```

Appendix B

VNX example of configuration file

In this second appendix I propose an example of configuration file used to deploy a virtual scenario, through the VNO service.

B.1. VNX configuration file

The XML specification file comes from the official documentation of the VNX project. Since I try to mantain the compatibility between the two services, I use an example of configuration file offered in the official documentation of VNX [50]. The only modification I made was the deletion of router definitions, since VNO does not yet support such functionality.

The XML file defines 4 virtual machines (h1,h2,h3,h4) connected to different virtual networks.

Listing B.1: VNX configuration file

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
~~~~~~~~~~~~~~~~~~~~
VNX Sample scenarios
~~~~~~~~~~~~~~~~~~~~

Name:        tutorial_ubuntu
Description: A simple tutorial scenario made of 6 Ubuntu virtual
   machines (4 hosts: h1, h2, h3 and h4;
             and 2 routers: r1 and r2) connected through three
                virtual networks. The host participates
             in the scenario having a network interface in Net3.

This file is part of the Virtual Networks over LinuX (VNX)
   Project distribution.
```

```
(www: http://www.dit.upm.es/vnx - e-mail: vnx@dit.upm.es)

Departamento de Ingenieria de Sistemas Telematicos (DIT)
Universidad Politecnica de Madrid
SPAIN
-->

<vnx xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/vnx-2.00.xsd
     ">
  <global>
    <version>2.0</version>
    <scenario_name>tutorial_ubuntu</scenario_name>
    <automac/>
    <vm_mgmt type="none" />
    <vm_defaults>
      <console id="0" display="no"/>
      <console id="1" display="yes"/>
    </vm_defaults>
  </global>

  <net name="Net0" mode="virtual_bridge" />
  <net name="Net1" mode="virtual_bridge" />


  <vm name="h1" type="libvirt" subtype="kvm" os="linux">
    <filesystem type="cow">/usr/share/vnx/filesystems/
       rootfs_ubuntu</filesystem>
    <mem>384M</mem>
    <console id="0" display="yes"/>
    <console id="1" display="no"/>
    <if id="1" net="Net0">
      <ipv4>10.0.0.2/24</ipv4>
    </if>
    <route type="ipv4" gw="10.0.0.1">default</route>

    <filetree seq="vnxtxt" root="/tmp">conf/txtfile</filetree>

    <!-- Start xeyes application -->
    <exec seq="xeyes"     type="verbatim" ostype="xexec">xeyes</
       exec>

    <!-- Start xeyes application and wait until it is closed -->
    <exec seq="xeyes2"    type="verbatim" ostype="xsystem">xeyes
       </exec>

    <!-- Start gedit, maximize the window and show a text file
       -->
```

```
  <exec seq="vnxtxt"    type="verbatim" ostype="system">chmod
     666 /tmp/vnx.txt</exec>
  <exec seq="vnxtxt"    type="verbatim" ostype="xexec">gedit /
     tmp/vnx.txt</exec>
  <exec seq="vnxtxt"    type="verbatim" ostype="xexec">sleep 3;
      wmctrl -r vnx.txt -b add,maximized_vert,maximized_horz</
     exec>
  <exec seq="vnxtxtoff" type="verbatim" ostype="system">pkill
     gedit; rm /tmp/vnx.*</exec>

  <!-- Start firefox and connect to h3 web server -->
  <exec seq="www-h3"    type="verbatim" ostype="xexec">firefox
     http://10.0.2.2</exec>
  <exec seq="www-h3-off" type="verbatim" ostype="system">pkill
     firefox</exec>

  <!-- Start calculator  -->
  <exec seq="calc"      type="verbatim" ostype="xexec">
     gcalctool</exec>
  <exec seq="calcoff"   type="verbatim" ostype="system">pkill
     gcalctool</exec>

</vm>

<vm name="h2" type="libvirt" subtype="kvm" os="linux">
  <filesystem type="cow">/usr/share/vnx/filesystems/
     rootfs_ubuntu</filesystem>
  <mem>128M</mem>
  <if id="1" net="Net0">
    <ipv4>10.0.0.3/24</ipv4>
  </if>
  <route type="ipv4" gw="10.0.0.1">default</route>
</vm>

<vm name="h3" type="libvirt" subtype="kvm" os="linux">
  <filesystem type="cow">/usr/share/vnx/filesystems/
     rootfs_ubuntu</filesystem>
 <mem>128M</mem>
 <if id="1" net="Net1">
    <ipv4>10.0.2.2/24</ipv4>
  </if>
  <route type="ipv4" gw="10.0.2.1">default</route>
  <!-- Copy the files under conf/tutorial_ubuntu/h3 to vm /var/
     www directory -->
  <filetree seq="start-www" root="/var/www">conf/
     tutorial_ubuntu/h3</filetree>
  <!-- Start/stop apache www server -->
  <exec seq="start-www" type="verbatim" ostype="system">chmod
     644 /var/www/*</exec>
```

```
    <exec seq="start-www" type="verbatim" ostype="system">service
        apache2 start</exec>
    <exec seq="stop-www" type="verbatim"  ostype="system">service
        apache2 stop</exec>
  </vm>

  <vm name="h4" type="libvirt" subtype="kvm" os="linux">
    <filesystem type="cow">/usr/share/vnx/filesystems/
       rootfs_ubuntu</filesystem>
    <mem>128M</mem>
    <if id="1" net="Net1">
      <ipv4>10.0.2.3/24</ipv4>
    </if>
    <route type="ipv4" gw="10.0.2.1">default</route>
    <!-- Copy the files under conf/tutorial_ubuntu/h4 to vm /var/
       www directory -->
    <filetree seq="start-www" root="/var/www">conf/
       tutorial_ubuntu/h4</filetree>
    <!-- Start/stop apache www server -->
    <exec seq="start-www" type="verbatim" ostype="system">chmod
       644 /var/www/*</exec>
    <exec seq="start-www" type="verbatim" ostype="system">service
        apache2 start</exec>
    <exec seq="stop-www" type="verbatim"  ostype="system">service
        apache2 stop</exec>
  </vm>
</vnx>
```

Glossary

API Application Programming Interface.

CLI Command Line Interface.

CPU Central Process Unit.

CSV Comma Separated Values.

DHCP Dynamic Host Configuration Protocol.

EC2 Elastic Compute CLoud.

EMF Eclipse Modeling Framework.

EMOF Essential Meta Object-Facility.

GCE Google Compute Engine.

GIM General Information Model.

HTTP HyperText Transfer Protocol.

IaaS Infrastructure as a Service.

IM Infrastructure Manager.

IP Internet Protocol.

KOALA Karlsruhe Open Application (for) cLoud Administration.

KVM Kernel-based Virtual Machine.

L3 Layer 3 - refers to the network layer of the OSI model.

LDAP Lightweight Directory Access Protocol.

libvirt A toolkit to interact with the virtualization capabilities.

NIST National Institute of Standard and Technology.

OCCI Open Cloud Computing Interface.

OMG Object Management Group.

OS Operating System.

OVF Open Virtualization Format .

OVS Open Virtual Switch.

PaaS Platform as a Service.

qcow2 QEMU Copy On Write disk.

RAM Random Access Memory.

S3 Simple Storage Service.

SaaS Software as a Service.

SOAP Simple Object Access Protocol.

SSH Secure Shell.

Tomcat Open source web server and servlet container.

VDI Virtual Disk Image.

VHD Virtual Hard Disk.

VM Virtual Machine.

VMDK Virtual Machine Disk.

VNO Virtual Networks over OpenStack.

VNX Virtual Networks over linuX.

Bibliography

[1] David M. Chess Jeffrey O. Kephart Jeffrey O. Kephart. Autonomic computing: Ibm's perspective on the state of information technology. *IEEE Computer IEEE Computer Society*, pages 41–50, 2003.

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

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

[4] Twenty-one experts define cloud computing | virtualization journal. Available online at http://virtualization.sys-con.com/node/612375. Retrieved On 27-09-2012.

[5] Celorio Pascual A. Desarrollo de un sistema para monitorización y análisis de plataformas de computación en la nube. Master's thesis, Universidad Politecnica de Madrid - ETSIT, 2012.

[6] Timothy Grance Peter Mell. The nist definition of cloud computing. *Special Publication 800-145*, September 2011.

[7] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Amazon elastic compute cloud (Amazon EC2). Available online at http://aws.amazon.com/ec2/. Retrieved on 26-09-2012.

[9] Dejan Milojifçifá, Ignacio M. Llorente, and Ruben S. Montero. Opennebula: A cloud management tool. *Internet Computing, IEEE*, 15(2):11 –14, march-april 2011.

[10] Openstack - open source cloud computing software. Available online at http://www.openstack.org/software/. Retrieved on 14-09-2012.

[11] Heroku - how it works. Available online at http://www.heroku.com/how. Retrieved on 26-09-2012.

[12] A. Zahariev. Google app engine. In *TKK T-110.5190 Seminar on Internetworking*, pages 1–5, 2009.

[13] Cloud foundry - architectural overview. Available online at http://docs.cloudfoundry.com/infrastructure/overview.html. Retrieved on 26-09-2012.

[14] Google docs - google docs homepage. Available online at http://www.google.com/google-d-s/documents/. Retrieved on 14-09-2012.

[15] M. Manzano, J. A. Hernandez, M. Uruena, and E. Calle. An empirical study of cloud gaming. In *Network and Systems Support for Games (NetGames), 2012 11th Annual Workshop on*, pages 1 –2, nov. 2012.

[16] Gartner's 2012 hype cycle for emerging technologies identifies "Tipping point" technologies that will unlock long-awaited technology scenarios. http://www.gartner.com/it/page.jsp?id=2124315.

[17] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755–768, May 2012.

[18] J. Baliga, R.W.A. Ayre, K. Hinton, and R.S. Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149 –167, January 2011.

[19] Eucalyptus cloud homepage. Available online at http://www.eucalyptus.com. Retrieved on 26-09-2012.

[20] About nimbus - nimbus. Available online at http://www.nimbusproject.org/about/. Retrieved on 27-09-2012.

[21] G. von Laszewski, J. Diaz, Fugang Wang, and G.C. Fox. Comparison of multiple cloud frameworks. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 734 –741, june 2012.

[22] LLC OpenStack. Openstack install and deploy manual. Available online at http://docs.openstack.org/folsom/openstack-compute/install/apt/content/, 2012. Retrieved on 14-09-2012.

[23] Damien Cerbelaud, Shishir Garg, and Jeremy Huylebroeck. Opening the clouds: qualitative overview of the state-of-the-art open source vm-based cloud management platforms. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 22:1–22:8, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[24] P. Sempolinski and D. Thain. A comparison and critique of eucalyptus, opennebula and nimbus. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 417 –426, 30 2010-dec. 3 2010.

[25] Comparativa de las plataformas cloud abiertas: OpenStack, OpenNebula, eucalyptus y CloudStack | revista cloud computing. Available online

at http://www.revistacloudcomputing.com/2013/01/comparativa-de-las-plataformas-cloud-abiertas-openstack-opennebula-eucalyptus-y-cloudstack/. Retrieved on January 2013.

[26] Bernd Harzog. Microsoft launches new azure features, new iaas offering, and performance management partnerships | the virtualization practice. http://www.virtualizationpractice.com/news-microsoft-launches-new-azure-features-new-iaas-offering-and-performance-management-partnerships-16365/.

[27] Git homepage. Available online at http://git-scm.com/. Retrieved on 28-09-2012.

[28] Dropbox - simplify your life. Available online at https://www.dropbox.com/. Retrieved on 20-10-2012.

[29] Panda Security. Panda cloud antivirus. Available online at http://www.cloudantivirus.com. Panda Cloud Antivirus Free Edition, Retrieved on 20-10-2012.

[30] C. Baun, M. Kunze, and V. Mauch. The KOALA cloud manager: Cloud service management the easy way. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, pages 744 –745, July 2011.

[31] Christian Baun and Marcel Kunze. The KOALA cloud management service: a modern approach for cloud infrastructure management. In *Proceedings of the First International Workshop on Cloud Computing Platforms*, CloudCP '11, page 1:1,Äì1:6, New York, NY, USA, 2011. ACM.

[32] Liutong Xu and Jie Yang. A management platform for eucalyptus-based iaas. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 193 –197, sept. 2011.

[33] Puppet labs: IT automation software for system administrators. Available online at http://puppetlabs.com/. Retrieved on 15-01-2013.

[34] Xen. Available online at http://wiki.xen.org/wiki/Main_Page.

[35] libvirt: The virtualization API. Available online at http://libvirt.org/.

[36] Anatomy of the libvirt virtualization library. Available online at http://www.ibm.com/developerworks/linux/library/l-libvirt/. Retrieved 21-11-2012.

[37] Eclipse modeling - EMF - home. Available online at http://www.eclipse.org/modeling/emf/.

[38] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[39] D. Fernandez, A. Cordero, J. Somavilla, J. Rodriguez, A. Corchero, L. Tarrafeta, and F. Galan. Distributed virtual scenarios over multi-host linux environments. In *2011 5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management (SVM)*, pages 1 –8, October 2011.

[40] Russ Miles and Kim Hamilton. *Learning UML 2.0*. O'Reilly, 2008.

[41] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, November 2008.

[42] OMG's MetaObject facility. Available online at http://www.omg.org/mof/.

[43] Douglas Crockford <douglas@crockford.com>. The application/json media type for JavaScript object notation (JSON). Available online at http://tools.ietf.org/html/rfc4627.

[44] Jackson JSON processor - homepage. Available online at http://jackson.codehaus.org/. Retrieved October 2012.

[45] Log4J2 guide - apache log4j 2. Available online at http://logging.apache.org/log4j/2.x/. Retrieved on 20-12-2012.

[46] Eclipse - the eclipse foundation open source community website. Available online at http://www.eclipse.org/. Retrieved on 27-10-2012.

[47] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 231–255. Springer Berlin Heidelberg, January 2006.

[48] What to expect from grizzly-1 milestone. Available online at http://fnords.wordpress.com/2012/11/23/what-to-expect-from-grizzly-1-milestone/. Retrieved 11 Jan 2013.

[49] State of flow: EclipseMetrics::projects. Available online at http://www.stateofflow.com/projects/16/eclipsemetrics. Retrieved on 20-01-2013.

[50] VNX homepage. Available online at http://web.dit.upm.es/vnxwiki/index.php/Main_Page.

[51] Open vSwitch homepage. Available online at http://openvswitch.org/. Retrieved on November 2012.

Author: Mattia Peirano

Date: