UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JULIANO ARAUJO WICKBOLDT

Flexible and Integrated Resource Management for IaaS Cloud Environments based on Programmability

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Lisandro Z. Granville

Porto Alegre
December 2015

CIP — CATALOGING-IN-PUBLICATION

Wickboldt, Juliano Araujo

Flexible and Integrated Resource Management for IaaS Cloud Environments based on Programmability / Juliano Araujo Wickboldt. – Porto Alegre: PPGC da UFRGS, 2015.

125 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Lisandro Z. Granville.

1. Cloud Computing. 2. Cloud Networking. 3. Resource Management. I. Granville, Lisandro Z.. II. Título.

"Life is like riding a bicycle.

To keep your balance you must keep moving."

— ALBERT EINSTEIN

ACKNOWLEDGMENTS

First of all, I would like to thank my parents and brother for the unconditional support and example of determination and perseverance they have always been for me. I am aware that time has been short and joyful moments sporadic, but if today I am taking one more step ahead this is due to the fact that you always believed in my potential and encourage me to move on. To my grandpa Araujo and grandma Carmen, I thank you for teaching me what really matters in life. As a child, when I used to take vacations and enjoyed them with my grandparents in Pelotas, I have learned little things like respect, honesty, simplicity, and humility, which turned me into better person.

I would also like to thank my advisor, Lisandro, who in spite of the geographical distance, has always been incredibly available (*i.e.*, on-the-line) when I most needed. Lisandro, thanks for the technical and philosophical discussions, as well as for that “little push” when the time to “vacate the bush” came about. Although this phase is coming to an end I will try to stick around – especially now with Bernardo’s arrival much will change, surely for the better –, thus make sure to count on me whenever you need.

My sincere gratitude goes as well to all members of the computer networks group of UFRGS, which today, almost eight years later, it is much larger than it was when I started. So, I will avoid naming (or using embarrassing nicknames of) each student and professor whom I had the pleasure of interacting with, but I’m sure everyone knows the contribution they have had on my professional and personal development. I will also thank the people who started with me back then and have left the group, who nowadays carry UFRGS’s flag and especially the teachings and experiences acquired here all over Brazil and around the world. So thank you all for the companionship, the barbecues, the late nights on campus, the partnerships during travels and organizing events. I apologize for not being able to write at least one story here with each one of you, in which case this thesis would turn into a book.

Last but not least (in fact, I always save the best for last, as I do with the steak in our university restaurant), I thank Carolina, my, let’s say “wife”, for the patience, perseverance, and care. The path might not have been always clear, many of the choices seemed hazy, I failed many times, hit some others, but regardless of any mishap, you always supported me. Certainly, I would not have gotten this far without your support (and sometimes some professional help can’t harm). Thank you!

AGRADECIMENTOS

Primeiramente, devo meus sinceros agradecimentos aos meus pais e irmão pelo apoio incondicional e pelo exemplo de garra e perseverança que sempre foram para mim. Sei que o tempo tem sido curto e os encontros esporádicos, mas se hoje cumpro mais esta etapa com sucesso isso se deve muito ao fato de vocês sempre acreditarem no meu potencial e me incentivarem para seguir adiante. Ao vô Araujo e à vó Carmen, agradeço por me ensinarem os reais valores dessa vida. Desde criança, quando eu ainda tirava férias e as desfrutava com vocês em Pelotas, aprendi pequenas coisas como respeito, honestidade, simplicidade e humildade, que me fizeram ser hoje uma pessoa melhor.

Agradeço também ao meu orientador Lisandro, que apesar da distância geográfica, sempre esteve incrivelmente próximo (*i.e.*, *on-the-line*) nas horas que eu mais precisei. Lisandro, obrigado pelas discussões técnicas, filosóficas e por acertar até no momento de dar aquele "empurrãozinho" providencial quando é chegada a hora de desocupar a moita. Apesar de esta etapa estar chegando ao fim vou tentar ficar por perto – ainda mais agora com o Bernardo chegando muita coisa vai mudar, para melhor certamente –, então precisando estamos aí.

Meus sinceros agradecimentos vão também para todo o pessoal do grupo de redes da UFRGS, que hoje, quase 8 anos depois, é muito maior do que era quando entrei. Sendo assim, vou procurar não mencionar nominalmente (nem usar apelidos constrangedores para) cada aluno e professor com quem tive o prazer de interagir, mas tenho certeza que cada um sabe a contribuição que teve na minha formação profissional e pessoal. Vou agradecer também ao pessoal que começou comigo lá atrás e já saiu do grupo, que hoje levam pelo Brasil e pelo mundo afora o nome da UFRGS e, principalmente, os ensinamentos e experiências adquiridas aqui. Obrigado então a todos pelo companheirismo, pelos churrascos, pelas madrugadas no campus, pelas parcerias em viagens e organizando eventos. Me desculpem por não poder contar aqui ao menos uma história com cada um de vocês, de forma que essa tese viraria um livro.

Por fim, mas não menos importante (na verdade, sempre deixo o melhor para o final como faço com o bife do RU), agradeço a Carolina, minha, digamos, "esposa", pela paciência, perseverança e carinho. Nem sempre o caminho foi claro, muitas das escolhas pareciam nebulosas, errei várias vezes, acertei algumas, mas independente de qualquer contratempo, tu sempre me apoiou. Certamente, eu não teria chegado tão longe sem teu apoio (e as vezes alguma ajuda profissional também vai bem). Muito obrigado!

ABSTRACT

Infrastructure as a Service (IaaS) clouds are becoming an increasingly common way to deploy modern Internet applications. Many cloud management platforms are available for users that want to build a private or public IaaS cloud (*e.g.*, OpenStack, Eucalyptus, OpenNebula). A common design aspect of current platforms is their black-box-like controlling nature. In general, cloud management platforms ship with one or a set of resource allocation strategies hard-coded into their core. Thus, cloud administrators have few opportunities to influence how resources are actually managed (*e.g.*, virtual machine placement or virtual link path selection). Administrators could benefit from customizations in resource management strategies, for example, to achieve environment specific objectives or to enable application-oriented resource allocation. Furthermore, resource management concerns in clouds are generally divided into computing, storage, and networking. Ideally, these three concerns should be addressed at the same level of importance by platform implementations. However, as opposed to computing and storage management, which have been extensively investigated, network management in cloud environments is rather incipient. The lack of flexibility and unbalanced support for resource management hinders the adoption of clouds as a viable execution environment for many modern Internet applications with strict requirements for elasticity or Quality of Service. In this thesis, a new concept of cloud management platform is introduced where resource management is made flexible by the addition of programmability to the core of the platform. Moreover, a simplified object-oriented API is introduced to enable administrators to write and run resource management programs to handle all kinds of resources from a single point. An implementation is presented as a proof of concept, including a set of drivers to deal with modern virtualization and networking technologies, such as software-defined networking with OpenFlow, Open vSwitches, and Libvirt. Two case studies are conducted to evaluate the use of resource management programs for the deployment and optimization of applications over an emulated network using Linux virtualization containers and Open vSwitches running the OpenFlow protocol. Results show the feasibility of the proposed approach and how deployment and optimization programs are able to achieve different objectives defined by the administrator.

Keywords: Cloud Computing. Cloud Networking. Resource Management.

Gerenciamento de Recursos Flexível e Integrado para Ambientes de Nuvem IaaS baseado em Programabilidade

RESUMO

Nuvens de infraestrutura como serviço (IaaS) estão se tornando um ambiente habitual para execução de aplicações modernas da Internet. Muitas plataformas de gerenciamento de nuvem estão disponíveis para aquele que deseja construir uma nuvem de IaaS privada ou pública (*e.g.*, OpenStack, Eucalyptus, OpenNebula). Um aspecto comum do projeto de plataformas atuais diz respeito ao seu modelo de controle caixa-preta. Em geral, as plataformas de gerenciamento de nuvem são distribuídas com um conjunto de estratégias de alocação de recursos embutida em seu núcleo. Dessa forma, os administradores de nuvem têm poucas oportunidades de influenciar a maneira como os recursos são realmente gerenciados (*e.g.*, posicionamento de máquinas virtuais ou seleção caminho de enlaces virtuais). Os administradores poderiam se beneficiar de personalizações em estratégias de gerenciamento de recursos, por exemplo, para atingir os objetivos específicos de cada ambiente ou a fim de permitir a alocação de recursos orientada à aplicação. Além disso, as preocupações acerca do gerenciamento de recursos em nuvens se dividem geralmente em computação, armazenamento e redes. Idealmente, essas três preocupações deveriam ser abordadas no mesmo nível de importância por implementações de plataformas. No entanto, ao contrário do gerenciamento de computação e armazenamento, que têm sido amplamente estudados, o gerenciamento de redes em ambientes de nuvem ainda é bastante incipiente. A falta de flexibilidade e suporte desequilibrado para o gerenciamento de recursos dificulta a adoção de nuvens como um ambiente de execução viável para muitas aplicações modernas da Internet com requisitos rigorosos de elasticidade e qualidade do serviço. Nesta tese, um novo conceito de plataforma de gerenciamento de nuvem é introduzido onde o gerenciamento de recursos flexível é obtido pela adição de programabilidade no núcleo da plataforma. Além disso, uma API simplificada e orientada a objetos é introduzida a fim de permitir que os administradores escrevam e executem programas de gerenciamento de recursos para lidar com todos os tipos de recursos a partir de um único ponto. Uma plataforma é apresentada como uma prova de conceito, incluindo um conjunto de adaptadores para lidar com tecnologias de virtualização e de redes modernas, como redes definidas por software com OpenFlow, Open vSwitches e Libvirt. Dois estudos de caso foram realizados a fim de avaliar a utilização de programas de gerenciamento de recursos para implantação e otimização de aplicações através de uma rede emulada usando contêineres de virtualização Linux e Open vSwitches operando sob o protocolo OpenFlow. Os resultados mostram a viabilidade da abordagem proposta e como os programas de implantação e otimização são capazes de alcançar diferentes objetivos definidos pelo administrador.

Palavras-chave: Computação e Comunicação em Nuvem, Gerenciamento de Recursos.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
AWS	Amazon Web Services
CaaS	Connectivity as a Service
CDMI	Cloud Data Management Interface
CDN	Content Delivery Network
CIM	Common Information Model
CIMI	Cloud Infrastructure Management Interface
CLI	Command Line Interface
DHCP	Dynamic Host Configuration Protocol
EC2	Elastic Compute Cloud
ECA	Event-Condition-Action
GUI	Graphical User Interfaces
HPC	High-Performance Computing
HuaaS	Human as a Service
IaaS	Infrastructure as a Service
ICN	Information-Centric Networking
iSCSI	Internet Small Computer System Interface
ISP	Internet Service Providers
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
MaaS	Metal as a Service
MPLS	Multiprotocol Label Switching
NaaS	Network as a Service

NFS	Network File System
OCCI	Open Cloud Computing Interface
OCF	OFELIA Control Framework
OCNI	Open Cloud Networking Interface
OVF	Open Virtualization Format
PaaS	Platform as a Service
PBMN	Policy-based Network Management
QoE	Quality of Experience
QoS	Quality of Service
REST	Representational State Transfer
S3	Simple Storage Service
SaaS	Software as a Service
SDN	Software-Defined Networking
SLA	Service-Level Agreement
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
VLAN	Virtual Local Area Network
VPXI	Virtual Private eXecution Infrastructure
VXDL	Virtual Infrastructure Description Language
XML	eXtensible Markup Language

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Utility computing, a concept coined almost 50 years ago, regained attention in the late-1990's and mid-2000's with the dissemination of grid and afterwards cloud computing (FOSTER et al., 2008). Similarly to current electric power distribution networks, utility computing is about delivering computing resources in the form of services to users. Today, cloud computing adopts this concept, offering all kinds of computing resources as services. The level of abstraction on which the user perceives and interacts with such resources is determined by a business model known as Everything as a Service (XaaS) (LENK et al., 2009; BAUN et al., 2011). There are at least three consolidated models to provide services over clouds: Infrastructure as a Service (IaaS), which encompasses creating the basic abstractions to mimic physical infrastructures as sets of virtualized elements; Platform as a Service (PaaS), where programming and execution environments are offered to users/programmers with a more abstract notion of the underlying infrastructure; and Software as a Service (SaaS), which is an even more abstract model, where end-users are granted with direct access applications hosted on the cloud infrastructure (usually operating over IaaS or PaaS). Moreover, there are other less commonly adopted models, such as Human as a Service (HuaaS)[1] (BAUN et al., 2011) and Metal as a Service (MaaS) (VORAS; ORLIC; MIHALJEVIC, 2013). HuaaS is a model where crowds of people are used as resources to process or provide information – often referred to as crowdsourcing – on a voluntary or rewarded basis, particularly useful for tasks where human intelligence or creativity is required. MaaS, on the other hand, is a recent model which aims to provide similar features to IaaS clouds (*e.g.*, elasticity and reliability), but over physical servers, *i.e.*, not necessarily relying on virtualization.

In particular, providing IaaS through clouds has drawn much attention from key information technology players (*e.g.*, Amazon, Google, Microsoft, IBM, HP) in the last few years. In general, *public cloud* providers adopt a pay-per-use model, where customers can "rent" virtualized computing resources, use these resources for a given amount of time, and release them when they are not necessary any longer. Many organizations, however, use their own physical resources to build IaaS clouds in order to deploy their particular applications, in a model usually referred to as *private cloud*. It is also possible to combine both private and public resources forming the so-called *hybrid clouds* (ZHANG; CHENG; BOUTABA, 2010). The reasons for choosing private instead of public clouds

[1]Sometimes also referred to as Human Provided Service (HPS)

vary, but it commonly include situations such as: when the customer does not trust the cloud provider to meet specific security requirements, when an organization already has idle resources to deploy a private cloud and does not want to rent anything from third parties, or in the case of companies or individuals running non-profit or experimental applications, such as researchers or universities.

There are numerous cloud management platforms used for the deployment and maintenance of both public, private, and hybrid clouds. Some platforms have been designed to be used exclusively in the context of a particular cloud provider, such as Amazon EC2 (Amazon, 2013) for Amazon's cloud or Microsoft's Azure (Azure, 2014). On the other hand, other management platforms are developed by the open source software community and are free to be downloaded and used by any interested party (*e.g.*, OpenNebula (OpenNebula, 2008), Eucalyptus (Eucalyptus, 2009), OpenStack (Rackspace, 2010), and CloudStack (CloudStack, 2012)). For example, HP's public IaaS cloud[2] and PayPal's private SaaS cloud[3] are both deployed using the OpenStack platform.

In terms of resource management, a common design aspect present in most cloud platforms is the separation of management concerns in computing, storage, and networking (SOTOMAYOR et al., 2009). Computing management is tightly related to handling virtual machines, allocating processing power and memory, and managing operating system images. Storage management, in turn, enables the allocation of persistent – possibly distributed – data volumes over a data center. In general, storage management is still subdivided into block storage (*i.e.*, disk volumes) and object storage (*i.e.*, media files hosted in the cloud). Lastly, networking management is perhaps the most challenging concern, given the variety abstractions and implementations created in every cloud management platform. Conceptually, managing networks in clouds means enabling communication between virtual resources, which can range from configuring IP addresses for virtual machines to creating complex virtual network infrastructures with switches, routers, and firewalls. Ideally, these three resource management concerns (*i.e.*, computing, storage, and networking) should be addressed at the same level of importance, or, in other words, platforms should support complex virtual network topology configuration as well as handling live virtual machine migration. However, in practice, some platforms focus more in one or another concern (JENNINGS; STADLER, 2014).

Resource management operations in cloud platforms need to be performed throughout the life-cycle of cloud applications (SEMPOLINSKI; THAIN, 2010; MANVI; SHYAM,

[2]Further information at http://www.hpcloud.com/why-hp-cloud/openstack
[3]Further information at http://www.openstack.org/user-stories/paypal/

2014). In general, in most platforms this life-cycle includes at least the following four phases:

1. the *specification phase*, at which the application owner (*e.g.*, tenant) requests a set of virtual resources – sometimes referred to as *cloud slice* or just *slice* – which usually includes a number of virtual machines, operating system images, storage space, and connectivity parameters;
2. the *provisioning phase*, at which the cloud platform allocates the requested resources from the data center and makes them available to the application;
3. the *runtime phase*, at which the application is running and using the provisioned virtual resources, and when the slice might undergo optimization, *e.g.*, when elasticity is supported;
4. the *termination phase*, at which resources are finally released by the application back to the platform and become available again for future allocations.

A major problem with current cloud management platforms comes from their black-box-like centralized control design, which resembles cluster task schedulers. In such a design, from the slice specification to the termination phase, very few opportunities exist for the administrator to influence how resources are actually managed by the platform (*e.g.*, virtual machine placement or virtual link path selection). IaaS clouds could benefit from customization in resource allocation strategies under two different perspectives: (*i*) to achieve environment specific objectives and (*ii*) to enable application-oriented resource management. Resource management objectives may vary from optimizing simple environmental metrics, such as minimizing energy consumption or network links utilization, to complex application-related requirements, like avoiding co-location of conflicting types of applications to respect privacy or security concerns. Further than customizing a platform for one specific objective, it could be even more beneficial to achieve several coexisting objectives (one objective for each type of application, for instance) using a single cloud management platform.

As mentioned earlier in this thesis, resource management concerns (*i.e.*, computing, storage, and networking) should ideally be addressed at the same level of importance. However, as opposed to computing and storage management, which have been extensively investigated in the last few years, network management in cloud environments is rather incipient. There is no common abstraction to provide network elements as services, similarly to what happens to virtual machines or virtual disk volumes. Some cloud management platforms offer very low-level protocol dependent configurations, such as IP/DHCP address ranges, while others offer high-level network services, such as load balancers or firewalls. The lack of flexibility and unbalanced support for resource management in current platforms hinders the adoption of clouds as a viable execution environment for many modern Internet applications – particularly highly distributed and network-intensive ones – with strict requirements for elasticity or Quality of Service (QoS) (MORENO-VOZ-MEDIANO; MONTERO; LLORENTE, 2013).

In this thesis, a new concept of cloud management platform is introduced where resource allocation and optimization are made flexible. By adding concepts of programmability to the core of a platform with a simplified object-oriented API it is possible to enable administrators to write and run personalized programs for both application deployment and optimization. In addition, administrators can also use this API to customize metrics and configure events to trigger optimization whenever necessary. The proposed API allows for more integrated resource management by offering high-level abstractions and operations to handle all sorts of resources (*i.e.*, computing, storage, and networking), all at the same level of importance. Furthermore, administrators can use the API to collect information from the monitoring system in order to use this information when deploying or optimizing applications.

1.1 Hypothesis & Research Questions

To overcome the limitations exposed in the context of resource management among cloud platforms, particularly in terms of flexibility, this thesis presents the following hypothesis.

Hypothesis: **a cloud platform incorporating the concepts of programmable and integrated resource management can enable the flexibility required to support modern applications and to adapt to environment specific needs**

In order to guide the investigations conducted in this thesis, the following research questions (RQ) associated with the hypothesis are defined and presented.

RQ I. *How to enable clouds to deal with network as a first order manageable resource?*

RQ II. *How to enable the detailed specification of applications needs?*

RQ III. *How can a cloud platform enable flexible integrated resource management for a wide range of applications and environments?*

One may note that the proposed research questions are intentionally broad and could lead to a multitude of investigations and outcomes. At the end of this study, at least one possible answer to each of the proposed research questions is identified. Of course, the answers provided in this thesis are supported by a substantial amount of results and analysis. However, it does not mean that different approaches could not accomplish similar results.

The methodology employed to show the feasibility of the proposed approach is based on the development of a cloud management platform and evaluated in two case studies. The platform developed supports a wide range of virtualization technologies through Libvirt (Libvirt, 2012), advanced networking through software-defined networking (SDN) with OpenFlow (MCKEOWN et al., 2008), and integration with a configurable cloud monitoring framework (CARVALHO et al., 2012). The evaluations carried out with the platform developed are conducted on an emulated environment, using Linux Containers virtualization with LXC (LXC, 2012), Open vSwitches (Open vSwitch, 2012) running OpenFlow, and Mininet (Mininet, 2013).

The first case study intends to show the deployment of an information-centric networking (ICN) application, based on the NetInf (KUTSCHER et al., 2011) architecture. A resource management program developed within the proposed platform is implemented to deploy this specific type of application. The results show a promising path towards end-to-end deployment – from detailed specification and resource allocation to monitoring and adaptation – of network-intensive applications in cloud environments.

The second case study shows that by using the proposed programmable cloud management platform an administrator is able to write, in a just few lines of code, optimization programs that achieve completely distinct objectives. In this case study three different optimization programs are implemented and executed. Results show how the approach enables flexible resource management in cloud platforms, from deployment to optimization of cloud slices.

1.2 Main Contributions

Throughout the development of this study many contributions are expected, both in terms of conceptual advancements in the state-of-the-art of resource management in the context of cloud computing and in delivering tools for overcoming technological challenges. Some of these contributions are listed as follows:

1. Adding more flexibility in resource management to make clouds a more suitable platform for applications and to assist administrators to achieve their environment specific objectives;
2. Adding support for advanced network configuration and management to clouds based on modern networking paradigms;
3. Creating a programmability environment based on an object-oriented API, which allows high-level abstractions for all sorts of manageable resources.

1.3 Thesis Roadmap

The remainder of this thesis is organized as follows.

In Chapter 2, the most important background concepts and studies related to this thesis are reviewed. Initially, a brief overview of the evolution of cloud computing is presented focusing mainly on resource management aspects. Afterwards, discussions are presented about virtualized elements and their associated operations, cloud interfaces and standardization efforts, the most important cloud management platforms proposed, and specific tools and libraries in the context of cloud resource management. Finally, additional academic efforts dealing with more particular research challenges in this context are presented.

In Chapter 3, the key concepts that drive this research are presented. These concepts are organized in the form of a conceptual architecture which includes the main components of a cloud management platform proposed to enable more flexible and integrated resource management in the context of IaaS clouds. Overall, the conceptual architecture is divided into six sections: *Graphical User Interface (GUI)*, *Slice Space*, *Programmable Logic*, *Event Space*, *Unified API*, and *Drivers*. This chapter also shows how the conceptual architecture interacts with a specialized external monitoring infrastructure in order to manage both physical and virtual resources.

In Chapter 4, the cloud management platform – which is called Aurora Cloud Manager – developed as a proof of concept is detailed. Initially, an overview of the interactions between the Aurora platform with the external monitoring infrastructure is presented. Afterwards, the initial specification of a *Cloud Slice* is introduced. Then, the implementation of core components of the conceptual architecture as well as the technology employed are discussed.

In Chapter 5, two case studies are presented to show the feasibility of the new concept of cloud management platform proposed in this thesis. The first case study aims to show the deployment of a network-intensive application based on the NetInf architecture. The second case study focuses on the optimization of resources according to different objectives and discusses the results obtained with three resource optimization programs developed within the proposed platform.

In Chapter 6, some final remarks and conclusions are presented. In addition, answers to the fundamental research questions proposed are discussed and justified. Moreover, opportunities to develop future work are identified and detailed.

2 BACKGROUND & RELATED WORK

Cloud computing is a term coined around 2007. Already in 2008, Vaquero *et al.* (2008) were on the pursuit of a consolidated definition of the term by analyzing a large number of previous definitions. These authors revealed a traditional approach to cloud computing emphasizing the provisioning of virtual computing and storage resources. This is tightly related with the fact that cloud computing initially emerged as an evolution of clusters and grids in the high-performance computing (HPC) community. Networking was then only considered as the means to interconnect virtual resources. Benson *et al.* (2011) also highlighted some of the limitations of cloud platforms in supporting robust networking functions. Recently, Moreno-Vozmediano *et al.* (2013) pointed out that current IaaS clouds are still too infrastructure-oriented and lack advanced service-oriented capabilities. For example, the authors emphasize the current poor support for service-oriented QoS metrics. Also, the definition of complex elasticity rules, based on both infrastructure-level and QoS metrics, is not well supported.

Many cloud management platforms were developed in research projects or directly by the open source software community. Examples of some major platforms currently available for organizations that want to build their own private or public clouds, are: Eucalyptus (NURMI et al., 2009), OpenNebula (SOTOMAYOR et al., 2009), OpenStack (Rackspace, 2010), and CloudStack (CloudStack, 2012). Given this variety of platforms, many standardization efforts also moved towards open standards besides Amazon's proprietary EC2/S3 interfaces to allow interoperability and information exchange among platforms and with end users. This chapter discusses some of the main initiatives towards providing IaaS through clouds, keeping the focus of discussions around virtualized resource management.

The remainder of this chapter initially discusses which virtualized elements and abstractions are commonly provided by IaaS clouds and which operations can be performed to handle these elements (Section 2.1). Afterward, some of the most important interfaces and standards used to model and exchange information among heterogeneous cloud environments are described (Section 2.2). Then, four of the main open source cloud platforms available and their main characteristics are detailed (Section 2.3). After, tools and libraries that can be used to interact with cloud platforms and virtualized resources are discussed (Section 2.4). Finally, other academic efforts targeted to design solutions for clouds filling particular gaps usually left behind by commercial implementations are presented, including a brief discussion on the use of experimental testbeds in this context (Section 2.5).

2.1 Virtualized Elements & Operations

Virtualization plays a key role in modern IaaS cloud environments by improving resource utilization and reducing costs (BELOGLAZOV; BUYYA, 2010). Typical elements that can be virtualized in these environments include computing and storage. Recently, virtualization has been extended also to the networking domain in order to overcome limitations of current cloud environments, such as poor isolation and increased security risks (BARI et al., 2013). Also, a set of operations need to be available to interact with and manage these virtualized elements. Conceptually, a set of virtualized resources of any type (*i.e.*, computing, storage, or network) allocated to a customer (a.k.a tenant) or an application is called a virtual infrastructure, also sometimes referred to as a *cloud slice*. This section describes the main elements that can form virtual infrastructures in a cloud environment and present a non-exhaustive list of the main operations performed over these resources derived from a plethora of existing implementations (Amazon, 2013; Eucalyptus, 2009; Rackspace, 2010; OpenNebula, 2008; CloudStack, 2012).

2.1.1 Computing

The virtualization of computing resources (*e.g.*, CPU and memory) is achieved by server virtualization technologies, or hypervisors (*e.g.*, VMWare, Xen, QEMU), which allow multiple virtual machines to be consolidated into a single physical machine. The benefits of server virtualization for cloud computing include performance isolation, improved application performance, and enhanced security (BARI et al., 2013).

Cloud providers deploy their infrastructures in data centers composed of several servers interconnected by a network. In the IaaS model, virtual machines are instantiated and allocated to tenants on-demand. Server virtualization adds flexibility to the cloud because virtual machines can be dynamically created, terminated, and migrated to different locations without affecting existing tenants. In addition, the capacity of a virtual machine (*i.e.*, allocated CPU, memory, and disk) can be adjusted to reflect changes in tenants' requirements without hardware changes.

Cloud providers have flexibility to decide where to allocate virtual machines in physical servers considering diverse criteria such as cost, energy consumption, or performance. In this regard, several virtual machine allocation schemes have been proposed in the literature that leverage the flexibility of virtualization to optimize resource utilization (ZHU; AGRAWAL, 2012; FRINCU; CRACIUN, 2011; RAO et al., 2011a; ISLAM et al., 2012; RAO et al., 2011b; LI et al., 2011).

To support the management of virtual machines a number of operations are available through most hypervisors and cloud platforms. Some of these operations are related to basic tasks, such as creating, removing, or modifying virtual machines and their properties. In some systems it is also possible to perform cloning operations to copy the properties from one virtual machine to another. Another set of operations are more related to provisioning of resources, such as deploying, undeploying, or migrating virtual machines. These operations can abstract the complexity of memory, CPU, or disk (*e.g.*, copying guest operating system image) allocation processes. Many operations are also often included to manipulate the state of a virtual machine and its guest operating system, such as start, stop, suspend, resume, snapshot, and restore (from snapshot).

Additionally, image management operations are also made available to create, remove, and upload guest operating system images for the subsequent deployment of virtual machines. Properly managing image placing and transferring can significantly affect the performance of resource provisioning as a whole in cloud environments and can be a quite challenging task to accomplish (PENG et al., 2012).

2.1.2 Storage

Storage virtualization consists of grouping multiple (possibly heterogeneous) storage devices that are seen as a single virtual storage space. There are two main abstractions to represent storage virtualization in clouds: virtual volumes and virtual data objects. The virtualization of storage as virtual volumes is important in this context because it simplifies the task of assigning disks to virtual machines. Furthermore, many implementations also include the notion of virtual volume pools, which represent different sources of available virtualizable storage spaces to allocate virtual volumes from (*e.g.*, separate local physical volumes or a remote Network File System (NFS)). On the other hand, cloud storage of virtual data objects, enables scalable and redundant creation and retrieval of data objects directly into/from the cloud. This abstraction is also often accompanied by the concept of containers, which in general serve to create a hierarchical structure of data objects similar to files and folders on any operating system.

Storage virtualization for both volumes and data objects is of utmost importance to enable the elasticity property of cloud computing. For example, virtual machines can have their disk space adjusted dynamically to support changes in cloud application requirements. Such adjustment is too complex and dynamic to be performed manually and, by virtualizing storage, cloud providers offer a uniform view to their users and reduce the need for manual provisioning. Also, with storage virtualization, cloud users do not need to know exactly where their data is stored. The details of which disks and partitions contain which objects or volumes is transparent to users, which also facilitates storage management for cloud providers. Furthermore, this transparency also gives the cloud provider the ability to decide on the best mechanisms to enable availability and redundancy to volumes and data objects according to the needs of each application.

Regarding the context of virtualization for volume storage there are several possible associated operations to handle these virtual resources. Basic operations include creating, removing, and resizing volumes (and pools, when supported), which depending on the implementation would allocate the resources on physical media right away or on demand. It should also be common to find operations to attach and detach virtual volumes into/from virtual machines, since that is the customary way to provide access to these storage spaces to end users.

In the case of storage virtualization for data objects there are similar operations to create and delete these objects and their associated containers. Moreover, many implementations include means to transfer the actual data in and out of the cloud environment, to allow publishing content and retrieving it back. Furthermore, advanced operations are sometimes available to stream the content out to the general public or even options for massive scale distribution employing concepts of Content Delivery Networks (CDN) (PATHAN; BUYYA; VAKALI, 2008; BUYYA et al., 2009).

2.1.3 Networking

Cloud infrastructures rely on local and wide area networks to connect the physical resources of their data centers (*i.e.*, servers, switches, and routers). Such networks are still based on the current IP architecture that has a number of problems. These problems are mainly related to the lack of isolation, which can allow a virtual infrastructure or application to interfere with another, resulting in poor performance or in security problems. Another issue is the limited support for innovation, which hinders the development of new architectures that could suit better cloud applications. As opposed to computing and storage management, which can be considered fairly stable, networking within cloud environments is still quite challenging.

Historically, cloud platforms have implemented solutions based on IP networks (*e.g.*, VLAN and MPLS) mainly trying to add a level of isolation among virtual machines. In general, a group of virtual machines was set to belong to the same flat network segment, which was configured to be isolated from the rest of the network, regardless of where virtual machines were positioned in the data center. Many scattered abstractions for more complex network elements were added on demand to these isolated virtual networks, such as dedicated DHCP servers for IP auto-configuration, firewalls to provide a controllable entry-point, and load balancers to distribute requests among pools of virtual machines.

Nowadays, to overcome the limitations of network architectures, network virtualization is being also considered in the context of clouds. Network virtualization has been a hot topic of investigation in recent years (KHAN et al., 2012; CHOWDHURY; BOUTABA, 2009) mainly in Internet Service Provider (ISP) networks. Similarly, in virtualized cloud networks multiple virtual networks need to share a physical substrate and can run isolated protocol stacks. A virtual network is part of a virtual infrastructure that includes abstractions for virtual network nodes (*i.e.*, switches and routers) and virtual links. In the context of clouds, virtual network nodes and links are employed mainly to interconnect virtual machines, but are not only limited to that. The advantages of virtualization of cloud networks include network performance isolation (*e.g.*, separate scheduling queues and algorithms), improved security, and the possibility to introduce new protocols and addressing schemes without changing the underlying substrate.

Regarding the operations available to manage virtualized network elements, there is not one widely accepted set of standardized operations in the context of clouds. It varies according to the complexity of network elements supported by each environment. For example, in environments where only basic isolation is provided over ordinary IP networks, management operations are also basic. In these environments, operations such as, creating/configuring a flat network (*e.g.*, assigning a VLAN tag and DHCP range) and associating virtual machines to this network, should be commonly available. Where more complex network virtualization abstractions are supported, at least operations for creating/removing virtual network nodes and links and associating these elements to the rest of the virtual infrastructure (*i.e.*, connecting virtual machines and storage) should be present. Also, explicit operations for deploying, undeploying, and migrating virtual routers and switches are desirable, since positioning these elements in relation to the underlying substrate tend to be at least as complex as it is for virtual machines. Moreover, operations for establishing, disabling, and configuring the communication in virtual links should be useful to control, for example, path selection and QoS parameters.

2.1.4 Management

Offering management as a resource that can be allocated and even virtualized is not as common as it is for computing, storage, and networking, but is certainly as important. By analogy, one can think of when a tenant establishes a new virtual infrastructure as acquiring a set of physical servers and network devices and installing them at a local facility. In a traditional physical network a management infrastructure would also have to be setup to monitor and operate the newly acquired devices. In a virtual environment this necessity remains the same, but in this case virtual management infrastructures can be created dynamically in association with other types of virtualized resources.

At the virtual level, each virtual infrastructure can independently operate its own management protocols and monitoring tools. For example, one tenant can use SNMP to manage his/her virtual infrastructure, while another can use NETCONF or Web services. At the data center level, cloud providers need to manage their physical nodes and network employing whatever tools and techniques they find most suitable. Moreover, cloud providers can also use information from hypervisors in order to obtain status of virtualized elements and the resources they consume. However, the effective management of virtual infrastructures may require combining information from all three levels, *i.e.*, physical, virtual, and hypervisors.

Monitoring is an example of a management aspect that can be virtualized and is particularly important in the context of this thesis. Once a new virtual infrastructure is created, a set of monitoring tools can be configured (MONTES et al., 2013) in order to start monitoring this set of virtual resources right away. This set of configurations and the corresponding monitored metrics are referred to as a *monitoring slice* (CARVALHO et al., 2012). Therefore, every virtual infrastructure is coupled with a monitoring slice to monitor resource status and utilization.

Operations are also necessary to instantiate monitoring slices associated with virtual infrastructures. For example, basic operations to start and stop monitoring need to exist in order to deploy or remove the appropriate monitoring infrastructure configurations required to monitor a given virtual device. In addition, after virtual devices are being monitored, it is also important to be able to read information about a specific device from monitoring infrastructure. Furthermore, some advanced operations would be also interesting to have, such as creating personalized metrics and events to be triggered in specific situations (*e.g.*, when a virtual machine crashes or upon virtual link saturation).

2.2 Cloud Interfaces & Standardization Efforts

Today, there are many heterogeneous cloud platforms that support the provisioning of virtualized infrastructures under a plethora of different specifications and technologies. Each cloud provider chooses the platform that suits them best or designs their own platforms to provide differentiated services to their consumers. The problem with this heterogeneity is that it hinders interoperability and causes vendor lock-in for consumers. In order to allow the remote management of virtual elements, many platforms already offer specific interfaces (*e.g.*, Amazon EC2/S3, Elastic Hosts, Flexiscale, Rackspace Cloud Servers, and VMware vSphere) to communicate with external applications.

To cope with this variety of technologies and support the development of platform agnostic cloud applications a few proposals took basically two different approaches: (*i*) employing proxy-style APIs in order to communicate with many providers using a set of technology specific adapters and (*ii*) creating standardized generic interfaces to be implemented by cloud platforms. The first approach has a drawback of introducing an additional layer of indirection in cloud systems, which incurs in overhead and increases latency. Nevertheless, there are libraries and tools that are widely employed, such as Apache Deltacloud and Libcloud, which are further discussed in Section 2.4. The second approach, on the other hand, represents a more conceptually elegant solution to the problem, when proposing some sort of *lingua franca* to communicate among cloud systems. The problem in standardization is always making everyone agree onto the same standard (ORTIZ, 2011). Ideally, a standardized interface should be open and extensible to allow widespread usage and adoption by cloud management platforms and application developers. This section reviews some of the most important efforts towards the definition of open standard interfaces to support virtualization and interoperability in the cloud.

2.2.1 Virtual Infrastructure Description Language (VXDL)

The Virtual Infrastructure Description Language (VXDL)[1] originated from the HPC community, where its main purpose was to describe a Virtual Private eXecution Infrastructure (VPXI) (KOSLOVSKI; PRIMET; CHARAO, 2009). A VPXI, in turn, is defined as a time-limited interconnection of virtual computing resources through a virtual private overlay network, which is conceptually similar to what in this thesis is called simply as a virtual infrastructure. Initially, VXDL was mostly employed in the process of translating the workflow of HPC applications into a description of a virtual infrastructure that can be suitable to efficiently execute these applications (HUU et al., 2011).

Currently, VXDL is a language being considered for modeling and describing complete virtual infrastructures also in the context of IaaS clouds. This language is being proposed as a standard form of communication between the cloud users and providers (KOSLOVSKI; SOUDAN; VICAT-BLANC, 2012). Moreover, VXDL is currently supported by the VXDL Forum, which is a not-for-profit consortium embodied by members of the network industry and academia (VXDL Forum, 2011). Besides describing the elements that compose virtual infrastructures, VXDL also includes the notion of a time limit or lifetime for these elements, allowing the representation and scheduling of changes over time in virtual resource allocations. Nevertheless, VXDL does not intend to specify the operations performed over virtual elements.

VXDL defines the concept of a virtual infrastructure as four key points: (*i*) simplicity and abstraction of complex virtual infrastructures for high-level protocol and technology independent resource manipulation; (*ii*) interconnections to represent that virtual elements are not independent and how they should be able to interact; (*iii*) timeliness to represent the dynamic aspect of virtual infrastructures, such as time intervals or phases where resources need to be reserved to a given application; and (*iv*) attributes representing the expected requirements associated with virtual elements, such as parameters for Quality of Experience (QoE), Quality of Service (QoS), or Service-Level Agreement (SLA) definitions.

The basic virtual elements that can compose a virtual infrastructure currently included in the VXDL specification are: *vNode* used to represent computing nodes, *e.g.*, virtual machines; *vStorage* used to represent virtual storage devices, *e.g.*, virtual volumes; *vRouter* used to represent virtual network devices, *e.g.*, virtual switch or router; *vLink*

[1]Formerly called Virtual eXecution Description Language (VXDL)

used to represent logical connections between two endpoints of the virtual infrastructure; and *vAccessPoint* used to represent an entry point to connect the virtual infrastructure with other networks, *e.g.*, a virtual firewall or proxy.

2.2.2 Open Cloud Computing Interface (OCCI)

The Open Cloud Computing Interface (OCCI) (OGF, 2012) introduces a set of open community led specifications to deal with cloud service resource management (EDMONDS et al., 2012). OCCI is supported by the Open Grid Forum and was originally conceived to create a remote management API for IaaS platforms allowing interoperability for common tasks, such as deployment, scaling, and monitoring virtual resources. Besides the definition of an open API, this specification also introduces a RESTful Protocol to exchange all sorts of management information and actions. The current release of OCCI is not anymore focused only in IaaS, but includes other cloud business models, such as PaaS and SaaS.

The current version of the specification[2] is designed to be modular and extensible, thus it is split into three complementary documents. The OCCI Core document (GFD.183) describes the formal definition of the OCCI Core Model. This document also describes how renderings can be used to interact with the core model (including associated behaviors) and how it can be expanded through extensions. The second document is OCCI Infrastructure (GFD.184), which contains the definition of the OCCI infrastructure extension for the IaaS domain. This document also defines additional resource types, their attributes, and actions that each resource type can perform. The third document, OCCI HTTP Rendering (GFD.185), defines means of interacting with the OCCI Core Model through the RESTful OCCI API. Moreover, this document defines how the OCCI Core Model can be communicated and serialized over HTTP protocol.

The OCCI Infrastructure document describes the modeling of virtual resources in IaaS as three basic element types: (*i*) Compute that are information processing resources, (*ii*) Storage that are intended to handle information recording, and (*iii*) Network representing L2 networking elements (*e.g.*, virtual switches). Also, there is an abstraction for the creation of Links between resources. Links can be of two types, *i.e.*, Network Interface or Storage Link, depending on the type of resource they connect. It is also possible to use this specification to define Infrastructure Templates, which are intended to be short-

[2]As of the ending of 2013 the current version of OCCI is v1.1 (release date April 7, 2011)

hand to predefined virtual resource specifications (*e.g.*, small, medium, and large virtual machine configurations). Moreover, the OCCI HTTP Rendering document complements these definitions by specifying management operations, such as creating, retrieving, updating, and deleting virtual resources. Furthermore, it details general requirements for the transmission of information over HTTP, such as security and authentication.

OCCI is currently implemented in many popular cloud management platforms, such as OpenStack, OpenNebula, and Eucalyptus. There are also base implementations in programming languages, such as *rOCCI* in Ruby and jclouds in Java, and automated compliance tests with *doyouspeakOCCI*. One particular effort aims to improve the inter-cloud networking standardization by proposing an extension to OCCI, called Open Cloud Networking Interface (OCNI) (MEDHIOUB; MSEKNI; ZEGHLACHE, 2013). There is also a reference implementation of OCNI called *pyOCNI*, written as a Python framework including JSON serialization for resource representation.

2.2.3 Open Virtualization Format (OVF)

The Open Virtualization Format (OVF) (DMTF, 2013b), currently in version 2.0.1, was introduced late in 2008 within the Virtualization Management initiative of the Distributed Management Task Force (DMTF), aiming to provide an open and extensible standard for packaging and distribution of software to be run in virtual machines. Its main virtue is to allow portability of virtual appliances onto multiple platforms through so-called OVF Packages, which may contain one or more virtual systems. The OVF standard is not tied to any particular hypervisor or processor architecture. Nevertheless, it is easily extensible through the specification of vendor-specific metadata included in OVF Packages.

An OVF Package is a core concept of the OVF specification, which consists of several files placed into one directory describing the structure of the packed virtual system. An OVF Package includes one OVF Descriptor, which is an XML document containing metadata about the package contents, such as product details, virtual hardware requirements, and licensing. The OVF Package may also include certificates, disk image files, or ISO images to be attached to virtual systems.

Within an OVF Package, an Envelope Element describes all metadata for the virtual machines included in the package. Among this metadata a detailed Virtual Hardware Description (based on Common Information Model (CIM) classes) can specify all types of virtual hardware resources required by a virtual system. This specification can be abstract or incomplete, allowing the virtualization platform to decide how to better satisfy the resource requirements, as long as required virtual devices are realized. Moreover, OVF Environment information can be added to define how the guest software and the deployment platform interact. This environment allows the guest software to access information about the deployment platform, such as values specified for the properties defined in the OVF Descriptor.

This standard is present in many hypervisor implementations and has been shown to be very useful for migrating virtual systems information among many hypervisors or platforms, since it allows precise description of virtual machines and virtual hardware requirements. However, it is not within the objectives of OVF to provide detailed specification for complete virtual infrastructures (*i.e.*, detailing interconnections, communication requirements, and network elements).

2.2.4 Cloud Infrastructure Management Interface (CIMI)

The Cloud Infrastructure Management Interface (CIMI) (DMTF, 2013a) standard is another DMTF proposal within the context of the Cloud Management initiative. This standard defines a model and protocol for management interactions between IaaS cloud providers and consumers. CIMI's main objective is to provide consumers with access to basic management operations on IaaS resources (virtual machines, storage, and networking) facilitating portability between different cloud implementations that support this standard. CIMI also specifies a RESTful protocol over HTTP using both JSON or XML formats to represent information and transmit management operations.

The model defined in CIMI includes basic types of virtualized resources, where Machine Resources are used to represent virtual machines, Volume Resources for storage, and Network Resources for virtual network devices and ports. Besides, CIMI also defines a Cloud Entry Point type of resource, which is intended to represent a catalog of virtual resources that can be queried by a consumer. A System Resource in this standard gathers one or more Network, Volume, or Machine Resources, and can be operated as a single resource. Finally, a Monitoring Resource is also defined to track progress of operations, metering, and monitoring of other virtual resources.

The protocol relies on basic HTTP operations (*i.e.*, PUT, GET, DELETE, HEAD, and POST) and uses either JSON of XML to transmit the message body. To manipulate virtual resources there are four basic CRUD (Create, Read, Update, and Delete) operations. It is also possible to extend the protocol by creating or customizing operations to manipulate the state of each particular resource. Moreover, the CIMI specification can also be integrated with OVF, in which case virtual machines represented as OVF Packages can be used to create Machine Resources or System Resources.

At the time of this writing, implementations of the CIMI standard are not so commonly found as OCCI or OVF are. One specific implementation worth noting is found within the Apache Deltacloud[3] project, which exposes a CIMI REST API to communicate with external applications supporting manipulation of Machine and Volume Resources abstractions.

2.2.5 Cloud Data Management Interface (CDMI)

The Cloud Data Management Interface (CDMI) (SNIA, 2012) is a standard specifically targeted to define an interface to access cloud storage and to manage data objects. CDMI is comparable to Amazon's S3 (SNIA, 2013), with the fundamental difference that it is conceived by the Storage Networking Industry Association (SNIA) to be an open standard targeted for future ANSI and ISO certification. This standard also includes a RESTful API running over HTTP to allow accessing capabilities of cloud storage systems, allocating and managing storage containers and data objects, handling users and group access rights, among other operations.

The CDMI standard defines a JSON serializable interface to manage data stored in clouds based on several abstractions. Data objects are a fundamental storage component analogous to files within a file system, which include metadata and value (contents). Container objects are intended to represent grouping of data, analogous to directories in a regular file system, this abstraction links together zero or more Data objects. Domain objects represent the concept of administrative ownership of data stored within cloud systems. This abstraction is very useful to facilitate billing, to restrict management operations to groups of objects, and to represent hierarchies of ownership. Queue objects provide first-in, first-out access to store or retrieve data from the cloud system. Queuing provides a simple mechanism for controlling concurrency when reading and writing Data objects in a reliable way. To facilitate interoperability this standard also includes mech-

[3] http://deltacloud.apache.org/cimi-rest.html

anisms for exporting data to other network storage platforms, such as iSCSI, NFS, and WebDAV.

Regarding implementations, CDMI it also not so commonly found deployed in most popular cloud management platforms. SNIA's Cloud Storage Technical Working Group (TWG) provides a Reference Implementation for the standard, which is currently a working draft and provides support only for version 1.0 of the specification. Some independent projects, such as CDMI add-on for OpenStack Swift and the CDMI-Serve in Python, have implemented basic support for the CDMI standard but do not present much recent activity.

Besides all the aforementioned efforts to create new standardized interfaces for virtual resource management in cloud environments, other approaches, protocols, and methods have been proposed as well and may be of interest in particular situations (ESTEVES; GRANVILLE; BOUTABA, 2013). Moreover, many organizations, such as OASIS, ETSI, ITU, NIST, and ISO, are currently engaged with their cloud and virtualization related working groups on developing standards and recommendations. The interested reader should look at DMTF's maintained wiki page Cloud-Standards.org[4] to keep track of future standardization initiatives.

One last point to emphasize in the current picture of standardization and interfaces for cloud management is that many platforms already support some default interoperability interfaces, such as OCCI, CIMI, and also the current *de facto* standard Amazon EC2. The use of such interfaces enables other applications to remotely send requests to a given cloud system in order to allocate virtual resources. Nevertheless, that only eliminates the need to access the user interface to request such resources. Still, if an administrator really needs to change the resource allocation strategies employed at the core of a cloud platform to map virtual resource requests into a set of physical resources, it would be necessary to dig into the source code of such platform – which is usually accessible via open source licenses, but not really easy to change.

2.3 Cloud Management Platforms

This section lists some of the most important efforts currently targeted to build tools for cloud management. The following open source cloud management platforms are

[4]http://cloud-standards.org/

in fact complete solutions to deploy and operate private or public, and sometimes even hybrid clouds. The main technical characteristics of the platforms presented in this section are summarized in Table 2.1, including programming language used for development, compatibility with some of the interfaces and standards previously mentioned, support for network management and monitoring, and latest releases information to indicate development activity. In the following sections, detailed descriptions of each platform and their main characteristics are presented.

Table 2.1: Technical characteristics of cloud management platforms

	Eucalyptus	**OpenNebula**	**OpenStack**	**CloudStack**
Programming Language	Mostly C and Java	C++ (Integration APIs in Ruby, JAVA, and Python)	Python	Mostly Java
Compatibility and Interoperability	Fully integrated with AWS	AWS, OCCI, and XML-RPC API	Nova and Swift are feature-wise compatible to EC2 and S3 (applications need to be adapted though), OCCI support (under development)	CloudStack REST API (XML or JSON)
Supported Hypervisors	vSphere, ESXi, KVM, any AWS-compatible clouds	KVM, Xen, and VMWare	QEMU/KVM over libvirt (fully supported), VMware and XenAPI (partially supported), many others at non-stable development stages	XenServer/XCP, KVM, and/or VMware ESXi with vSphere
Identity Management	Role-Based Access Control mechanisms with Microsoft Active Directory or LDAP systems	Sunstone, EC2, OCCI, SSH, x509 certificates, and LDAP	Local database, EC2/S3, RBAC, Token-based, SSL, x509 or PKI certificates, and LDAP	Internal or LDAP
Resource usage control	Resource quotas for users and groups	Resource quotas for users and groups	Configurable quotas per user (tenant) defined by each project	Usage Server separately-installed provides records for billing, resource limits per project
Networking	Basic support with 4 operating modes	IP/DHCP ranges customizable by users, many options for administrator require manual configuration	Several options via Neutron component, extensible with plug-ins	Two operating modes, several networking-as-a-service options in advanced configurations
Monitoring	CloudWatch	Internal, gathers information from hypervisors	Simple customizable dashboard relies on information provided by other components	Some performance indicators available through the API are displayed to users and administrator
First Release	1.0 (Released: May 29th, 2008)	1.0 (Released: July 24th, 2008)	Austin (October 21st, 2010)	4.0.0-incubating (Released: November 6th, 2012)
Current Major Release	4.1 (Released: January 29th, 2015)	4.12 (Cotton Candy) (Released: March 11th, 2015)	Kilo (Released: April 30th, 2015)	4.5 (Released: August 23rd, 2015)
License	GPL v3.0	Apache v2.0	Apache v2.0	Apache v2.0

Source: by author (2015).

2.3.1 Eucalyptus

Eucalyptus started as a research project in the Computer Science Department at the University of California, Santa Barbara in 2007, within a project called Virtual Grid Application Development Software Project (VGrADS) funded by the National Science Foundation. This is one of the first open source initiatives to build a cloud management platform to allow users to deploy their own private clouds (NURMI et al., 2009; Eucalyptus, 2009). Currently, Eucalyptus is in version 4.1 and comprises full integration with Amazon Web Services (AWS) – including EC2, S3, Elastic Block Store (EBS), Identity and Access Management (IAM), Auto Scaling, Elastic Load Balancing (ELB), and CloudWatch – enabling both private and hybrid cloud deployments.

Eucalyptus architecture is based on four high-level components: (*i*) Node Controller executes at physical hosts of the infrastructure and is responsible for controlling the execution of virtual machine instances; (*ii*) Cluster Controller works as a front end at the cluster-level (*i.e.*, Availability Zone) managing virtual machine execution and scheduling on Node Controllers, controls cluster-level Service Level Agreements (SLAs), and also as manages virtual networks; (*iii*) Storage Controller exists both at cluster-level and at cloud-level (Walrus) and implements is a put/get Storage-as-a-Service solution based on Amazon's S3 interface, providing a mechanism for storing and accessing virtual machine images and user data; and (*iv*) Cloud Controller is the entry-point into the cloud for users and administrators, it implements an EC2-compatible interface and coordinates other components to perform high-level tasks, such as authentication, accounting, reporting, and quota management.

For networking, Eucalyptus offers four operating modes: (*i*) Managed in which the platform manages layers 2 and 3 virtual machine isolation, employing a built-in DHCP service. This mode requires a switch to forward a configurable range of VLAN-tagged packets; (*ii*) Managed (no VLAN) in which only layer 3 virtual machine isolation is possible; (*iii*) Static, where there is no virtual machine isolation, employs a built-in DHCP service for static IP assignment; and (*iv*) System, where there is also no virtual machine isolation and, in this case, no automatic address handling since Eucalyptus will rely on an existing external DHCP service. In version 4.0, Eucalyptus introduced a new functionality to support network configuration through the Edge Networking Mode. In such mode, a stand-alone component is placed at Node Controllers specifically designed to manage network configuration upon request of the centralized Cluster Controller.

2.3.2 OpenNebula

In its early days OpenNebula was a research project at the Universidad Complutense de Madrid. The first version of the platform was released open source in 2008 within the European Union's Seventh Framework Programme (FP7) project called RESERVOIR – Resources and Services Virtualization without Barriers (2008-2011) (SOTOMAYOR et al., 2009). Nowadays, OpenNebula (version 4.12 Cotton Candy released November

3rd, 2014) is a platform used mostly for the deployment of private clouds, but is also capable of interfacing with other systems to work as hybrid or public cloud environment.

OpenNebula is conceptually organized in a three-layered architecture (MORENO-VOZMEDIANO; MONTERO; LLORENTE, 2012). At the top, the Tools layer comprises higher-level functions, such as cloud level virtual machine scheduling, providing CLI and GUI access for both users and administrators, managing and supporting multitier services, elasticity and admission control, and exposing interfaces to external clouds through AWS and OCCI. At the Core layer, vital functions are performed, such as accounting, authorization, and authentication, as well as resource management for computing, storage, networking, and virtual machine images. Also at this layer, the platform implements monitoring of resources by retrieving information available from hypervisors to read the state of virtual machines and manages federations enabling access to remote cloud infrastructures, which can be either partner infrastructures governed by a similar platform or public cloud providers. At the bottom, the Drivers layer implements infrastructure and cloud drivers to provide an abstraction to communicate with the underlying technology or to enable access to remote cloud providers.

OpenNebula allows administrators to setup multiple zones and create federated virtual data centers considering different federation paradigms (*e.g.*, cloud aggregation, bursting, or brokering), in which case each zone operates their network configurations independently. From the user point of view, setting up a network in the OpenNebula platform is restricted to the creation of a DHCP IP range that will in turn be automatically configured in each virtual machine. The administrator can change the way virtual machines connect to the physical ports of the host machine using one of many options, *i.e.*, VLAN 802.1Q to allow isolation, EBtables and Open vSwitch permit implementation of traffic filtering, and VMware VLANs which isolate virtual machines running over VMware hypervisor. It is also possible to deploy Virtual Routers from OpenNebula's Marketplace to work as an actual router, DHCP, or DNS server.

2.3.3 OpenStack

OpenStack started as a joint project between Rackspace Hosting and NASA around mid 2010, aiming to provide a cloud-software solution to run over commodity hardware (Rackspace, 2010). Right after the first official release (beginning of 2011), OpenStack was quickly adopted and packed within many Linux distributions, such as Ubuntu, Debian, and Red Hat. Today, it is the cloud management platform with the most active community counting on more than 13,000 registered people from over 130 countries. OpenStack is currently developed in 9 parallel core projects (plus 4 incubated) all coordinated by the OpenStack Foundation, which is embodied by 9,500 individuals and 850 different organizations.

The OpenStack architecture consists of a myriad of interconnected components, each one developed under a separate project, to deliver a complete cloud infrastructure management solution. Initially, only two components were present, Compute (Nova) and Object Storage (Swift), which respectively provide functionality for handling virtual machines and a scalable redundant object storage systems. Adopting an incremental approach, incubated/community projects were gradually included in the core architecture, such as Dashboard (Horizon) to provide administration GUI access, Identity Service (Keystone) to support a central directory of users mapped to services, and Image Service (Glance) to allow discovery, registration, and delivery of disk and server images. The current release of OpenStack (Kilo) includes advanced network configuration with Neutron, persistent block-level storage with Cinder, a single point of contact for billing systems through Ceilometer, and a service to orchestrate multiple composite cloud applications via Heat.

As for networking, a community project called Quantum started in April 2011 and was targeted to further develop the networking support of OpenStack by employing virtual network overlays in a Connectivity as a Service (CaaS) perspective. From release Folsom on, Quantum was added as a core project and renamed to Neutron. Currently, this component allows administrators to employ from basic networking configuration of IP addresses, allowing for dedicated static address assignment or via DHCP, to complex configuration with SDN technology like OpenFlow. Moreover, Neutron provides the platform with the ability of adding plug-ins to introduce more complex functionality, such as quality of service, intrusion detection systems, load balancing, firewalls, and virtual private networks.

2.3.4 CloudStack

CloudStack started as a project from a startup company called VMOps in 2008, later renamed Cloud.com, and was first released as open source in mid 2010. After Cloud.com was acquired by Citrix, CloudStack was relicensed to Apache 2.0 and incubated by the Apache Software Foundation in April 2012. Ever since, the project has developed a powerful cloud platform to orchestrate resources in highly distributed environments for both private and public cloud deployments (CloudStack, 2012).

CloudStack deployments are organized into two basic building blocks, a Management Server and a Cloud Infrastructure. The Management Server is a central point of configuration for the cloud (these servers might be clustered for reliability reasons). It provides Web user interface and API access, manages the assignment of virtual machines to hosts, allocates public and private IP addresses to particular accounts, manages images, among other tasks. The Cloud Infrastructure is composed of distributed Zones (typically, data centers) hierarchically organized into Pods, Clusters, Hosts, Primary, and Secondary Storage. The CloudStack Cloud Infrastructure may also optionally include Regions (perhaps geographically distributed), to aggregate multiple Zones, and each Region is controlled by a different set of Management Servers, turning the platform into a highly distributed and reliable system. Moreover, a separate Python tool called CloudMonkey is available to provide CLI and shell environments for interacting with CloudStack-based clouds.

CloudStack offers two types of networking configurations: (*i*) Basic, which is an AWS-style networking providing a single network where guest isolation can be achieved through layer-3 means such as security groups and (*ii*) Advanced, where more sophisticated network topologies can be created. CloudStack also offers a variety of Networking as a Service (NaaS) features, such as creation of VPNs, firewalls, and load balancers. Moreover, this tool provides the ability to create a Virtual Private Cloud, which is a private, isolated part of CloudStack that can have its own virtual network topology. Virtual machines in this virtual network that can have any private addresses since they are completely isolated from others.

By analyzing the evolution of cloud management platforms, one may note that their primary focus is on virtualization of computing and storage resources (SEMPOLINSKI; THAIN, 2010). This biased evolution may be explained by the fact that platforms for cloud computing management emerged from the cluster and grid computing communities, thus the network used to be considered only as means to transfer data or jobs rather than an allocable type of resource. The majority of these platforms offers basic network configuration via DHCP servers only to configure IP addresses for virtual machines. Initially, isolation of traffic between sets of virtual machines was a primary concern, so many platforms started supporting the configuration of VLAN tags, for instance. Only recently, platforms started adding advanced networking capabilities based on concepts from the network virtualization area, such as employing NaaS and CaaS models. Despite all this recent interest in adding network-related features to already established cloud platforms, implementations still diverge in defining the most appropriate abstractions to represent virtual network elements.

A second aspect that is worth mentioning regards the ability of a cloud administrator to get a cloud management platform to operate resources in line with the cloud provider's objectives. Most of the aforementioned platforms are indeed open source, which in theory allows their source code to be downloaded and modified to almost any purpose. That does not necessarily mean that digging into the code of a complex platform is an easy task to do, specially as a resource management activity on a daily basis. Rational resource management is at the heart of the cloud business and should be placed right in the hands of the person who understands both business and application requirements, as well as the technicality of the underlying infrastructure. Besides adapting a platform to the cloud provider's infrastructure level objectives, it might also be the case that a platform needs to adapt to many different application specific objectives. This thesis argues that it is utterly important that a cloud platform allows the flexibility necessary to have its resource management strategies easily customized for both application and environment specific objectives.

2.4 Specific Tools & Libraries

Next, some tools and libraries mainly designed to deal with the diversity of technologies involved in cloud environments are described. Unlike cloud platforms, these tools do not intend to offer a complete solution for cloud providers. Nevertheless, they play a key role in integration and allow applications to be written in a more generic manner in terms of virtual resource management.

2.4.1 Libcloud

Libcloud is a client Python library for interacting with the most popular cloud management platforms (Libcloud, 2012). This library originally started being developed within Cloudkick (extinct cloud monitoring software project, now part of Rackspace) and today is an independent free software project licensed under the Apache License 2.0. The main idea behind Libcloud is to create a programming environment to facilitate developers on the task of building products that can be ported across a wide range of cloud environments. Therefore, much of the library is about providing a long list of drivers to communicate with different cloud platforms. Currently, Libcloud supports more than 26 different providers, including Amazon's AWS, OpenStack, OpenNebula, and Eucalyptus, just to mention a few.

Moreover, this library also provides a unified Python API, offering a set of common operations to be mapped to the appropriate calls to the remote cloud system. These operations are divided into four abstractions: (*i*) Compute, which enables operations for handling virtual machines (*e.g.*, list/create/reboot/destroy virtual machines) and its extension Block Storage to manage volumes attached to virtual machines (*e.g.*, create/destroy volumes, attach volume to virtual machine); (*ii*) Load Balancer, which includes operations for the management of load balancers as a service (*e.g.*, create/list members, attach/detach member or compute node) and is available in some providers; (*iii*) Object Storage, which offers operations for creating an Amazon S3-like environment for handling data objects in a cloud (list/create/delete containers or objects, upload/download/stream object) and its extension to CDN for providers that support these operations (*e.g.*, enable CDN container or object, get CDN container or object URL); and (*iv*) Domain Name System (DNS), allows management operations for DNS as a service (*e.g.*, list zones or records, create/update zone or record) in providers that support it, such as Rackspace Cloud DNS.

2.4.2 Deltacloud

Deltacloud follows a very similar philosophy compared to Libcloud. It is also an Apache Software Foundation project – left incubation in October 2011 and is now a top-level project – and is similarly targeted to provide an intermediary layer to let applications communicate with several different cloud management platforms. Nevertheless, instead of providing a programming environment through a specific programming language, Deltacloud enables management of resources in different clouds by the use of one of three supported RESTful APIs (DeltaCloud, 2011): (*i*) Deltacloud classic, (*ii*) DMTF's CIMI, (*iii*) Amazon's EC2.

Deltacloud implements drivers for more than 20 different providers and offers several operations divided into two main abstractions: (*i*) Compute Driver, which includes operations for managing virtual machines, such as create/start/stop/reboot/destroy virtual machine instances, list all/get details about hardware profiles, realms, images, and virtual machine instances; and (*ii*) Storage Driver, providing operations similar to Amazon S3 to manage data objects stored in clouds, such as create/update/delete buckets (analogous to folders), create/update/delete blobs (analogous to data files), and read/write blobs data and attributes.

2.4.3 Libvirt

Libvirt is a toolkit for interacting with multiple virtualization providers/hypervisors to manage virtual compute, storage, and networking resources. It is a free collection of software available under GNU LGPL and is not particularly targeted to cloud systems. Nevertheless, Libvirt has shown to be very useful to handle low level virtualization operations and is actually used under the hood by cloud platforms like OpenStack to interface with some hypervisors. Libvirt supports several hypervisors (*e.g.*, KVM/QEMU, Xen, VirtualBox, and VMware), creation of virtual networks (*e.g.*, bridging or NAT), and storage on IDE, SCSI, and USB disks and LVM, iSCSI, and NFS file systems. It also provides remote management using TLS encryption, x509 certificates, and authentication through Kerberos or SASL.

Libvirt provides a C/C++ API with bindings to several other languages, such as Python, Java, PHP, Ruby, and C#. This API includes operations for managing virtual resources as well as retrieving information and capabilities from physical hosts and hypervisors. Virtual resource management operations are divided into three abstractions: (*i*) Domains, which are common virtual machine-related operations, such as create, start, stop, and migrate; (*ii*) Storage, for managing block storage volumes or pools; and (*iii*) Network, which includes operations such as creating and connecting virtual machines to bridges, enabling NAT, and DHCP. Note that network operations are all performed within the scope of a single physical host, *i.e.*, it is not possible to connect virtual machines in separate hosts to the same bridged network, for example.

In summary, the first two libraries (*i.e.*, Libcloud and Deltacloud) aim to integrate different cloud technologies. They both operate at an even higher level of abstractions than the interfaces and standards presented in Section 2.2, adding a level of indirection between the cloud platform operating the resources and the application requesting them. Again, it is possible to visualize that there is no common set of abstractions among different libraries. Libvirt, on the other hand, is a programming interface that operates very close to the underlying infrastructure integrating mainly different hypervisor technologies. The person using this programming interface still needs to deal with many technology specific parameters and there is only rudimentary support for management of virtualized network elements.

2.5 Additional Related Work

This section starts discussing the need to translate application goals to an actual allocation of virtual resources. Afterwards, some academic efforts that aim to make cloud platforms more customizable are presented. Finally, the use of experimentation testbeds as a solution to manage virtual resources is also discussed.

2.5.1 Application Goal Translation

Although the focus of this thesis is mainly on providing IaaS, it is important to keep in mind that what runs over these virtualized infrastructures is actually an application. Many approaches have been proposed to “translate” the initial specification of an application – the input from the user to the cloud platform – into actual resource allocation. Most of the aforementioned commercial platforms adopt an oversimplified approach, using forms or wizards, asking questions like: how many virtual machine instances? how much RAM memory? how much disk space? and which operating system image should be deployed? Then, the cloud platform finds a way to allocated virtual resources onto the available physical ones.

Some recent studies propose ways to allow a high-level specification of application constraints (SUN et al., 2012; WUHIB; STADLER; LINDGREN, 2012; ESTEVES et al.,

2013) and complex methods of translation to resource allocation. In this thesis, allowing the application to be specified in very high levels of abstraction is not a major concern. However, means are included for the user to specify a detailed virtual infrastructure where the application can be deployed – further explained in Chapter 3.

2.5.2 Customizability and Elasticity

To turn cloud platforms more customizable, Guo *et al.* (2010) take a service composition approach to create cloud platforms by employing a process description language called Lightweight Coordination Calculus (LCC) to coordinate system components. Resource allocation and optimization are performed respectively by the *Scheduler* and *SLA & Billing Manager* components, which can interact with other components in any way the developer decides to implement. Although interesting, the approach does not yet consider networking as a first-order resource. Moreover, the notion of virtual infrastructure composed of a set virtual devices is vague. It is not possible to specify, for example, a virtual topology or allocation of network resources such as links or virtual routers. Network configuration is done via DHCP and virtual machines reside inside the same subnet as they belong into the same virtual machine pool.

As an evolution of Guo *et al.*'s work, a lightweight approach to provisioning resources based on application containers, instead of heavy virtualization, was proposed by He *et al.* (2012). This approach shows promising performance improvements to certain types of applications, because of the fast virtualization schema adopted. However, to scale up an application, the authors assume that when a container has consumed all allocated resources, creating a new container will improve application performance. In fact, that is the most common approach, and many authors consider that scaling up and down is a matter of allocating more or less resources, *i.e.*, when an application is overloaded the straightforward solution is to spawn new virtual machines (ALI-ELDIN; TORDSSON; ELMROTH, 2012). This thesis argues that the need for optimization may depend on the application specific behavior and on the conditions of underlying infrastructure. Moreover, auto-scaling is not only necessary for virtual machines, but for all sorts of virtual resources that may help on optimizing applications (HASAN et al., 2012). Thus, this thesis advocates for easily programmable optimization strategies and metrics inside the core of cloud platforms.

2.5.3 Experimentation Testbeds

There is a number of testing platforms (testbeds) that provide IaaS for researchers to run all sorts of experiments in large scale infrastructures. OFELIA Control Framework (OCF) (OCF, 2011), from the FP7 OFELIA project (KÖPSEL; WOESNER, 2011), allows researchers to reserve resources and run experiments over OpenFlow networks (MCKEOWN et al., 2008). Another example is ProtoGENI (ProtoGENI, 2012), from the GENI project (GENI, 2011), which allows the creation of a virtual infrastructure of interconnected virtual resources, aggregating physical resources available from many partner federations.

Although these platforms are able to provide complex virtual infrastructure allocations, they do not strictly follow the cloud computing model. The notion of an application to run on a cloud is rather vague, the approach is much more resource oriented. On the other hand, they do implement more complex network configuration functionality than most of the previously mentioned platforms, following a NaaS model. Also, resources are "leased" for testing purposes and there is usually much bureaucracy and strict rules to access them. It is important to notice that, in most cases, the user/researcher has many options to choose in terms of where they want their resources to be positioned. Usually, researchers have a notion of the underlying topology and hardware resources available too. That is generally not the way most cloud providers would operate; cloud providers usually try to optimize resource placement according to well-defined objectives (*e.g.*, low energy consumption) and hide such placement details from users.

2.6 Summary

This chapter presented the state-of-the-art of resource management in IaaS clouds both in terms of fundamental concepts of virtualization and tool support to operate virtual elements in real life systems. It is possible to notice that many solutions address a specific set of problems and tools present an overlapping set of features, while many issues remain only marginally or not addressed. For example, cloud platforms lack integrated support for all types of resources (computing, storage, and networking) developed at a level that allows complex configuration and optimization of them all. Moreover, management as an allocable resource to allow end users to gain full control of their virtualized elements is rarely mentioned. Further than that, despite the fact that many open source platforms are available, reprogramming the core of resource allocation strategies in those platforms can be a complex task and is something a cloud administrator would certainly not want to do on a daily basis. Therefore, the next chapter presents the conceptual building blocks of our proposed solution to add more flexibility to cloud environments providing administrators with new means of customizing resource management for their specific environment or applications' needs.

3 KEY CONCEPTS & ARCHITECTURE

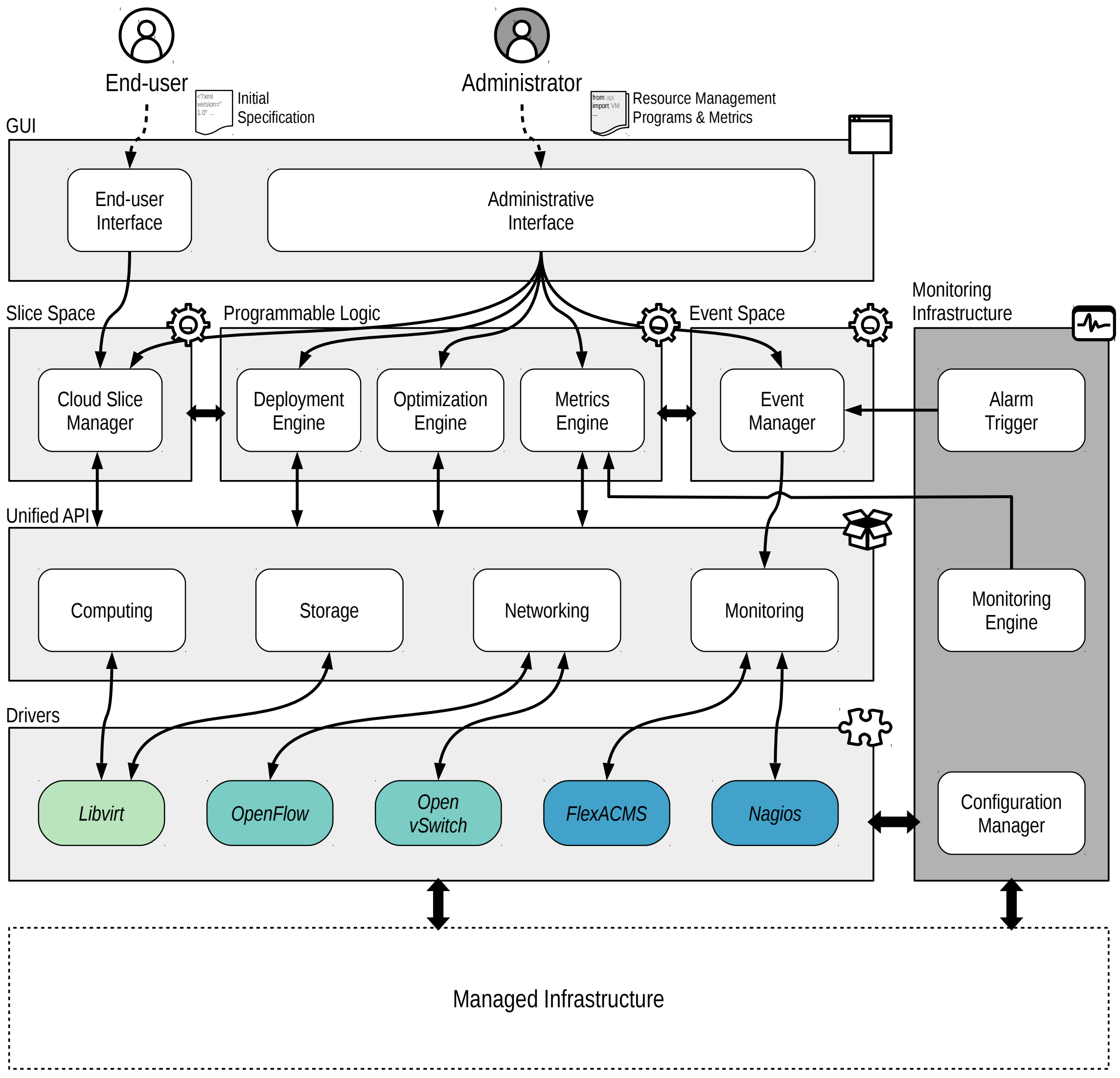
This chapter presents the main concepts that drive this research. Figure 3.1 depicts the conceptual architecture that organizes the main components of a cloud management platform proposed to enable more flexible and integrated resource management in the context of IaaS clouds. Overall, the conceptual architecture is divided into six sections (light gray boxes): *Graphical User Interface (GUI)*, *Slice Space*, *Programmable Logic*, *Event Space*, *Unified API*, and *Drivers*. Each of these sections groups together conceptually similar components. Moreover, the architecture interacts with a specialized external *Monitoring Infrastructure* (dark gray box) in order to obtain advanced monitoring features. Also, both the conceptual architecture – through its drivers section – and the *Monitoring Infrastructure* interact to manage the physical and virtual elements at the *Managed Infrastructure*. All components of the architecture and interactions among them are detailed throughout this chapter.

The remainder of this chapter presents the concepts involved in the architecture of Figure 3.1 from a top-down perspective, following closely the typical process of request, establishment, and maintenance of virtual infrastructures over IaaS clouds. Initially, Section 3.1 describes the main parties involved in this process and discusses a few variants in the business relationship between these parties. Section 3.2 details how an *Initial Specification* document should describe a virtual infrastructure to be deployed over an IaaS cloud environment. Section 3.3 discusses how *Resource Management Programs & Metrics* are structured, whilst Sections 3.4 and 3.5 argue on how these programs and metrics play their role in the management cycle of virtual infrastructures. Finally, the proposed *Unified API* and *Drivers* that implement technology specific protocols and commands are described respectively in Sections 3.6 and 3.7.

3.1 Involved Parties

There are at least two parties involved in the process of request, establishment, and maintenance of virtual infrastructures over IaaS clouds: a tenant and a cloud provider. In the conceptual architecture depicted in Figure 3.1 the tenant is represented by the *End-user* actor and the cloud provider by its *Administrator*. The tenant is the person or organization interested in “buying” virtual resources from a cloud provider for deploying a service or application on top of a cloud-based infrastructure.

Figure 3.1: Conceptual architecture of a cloud management platform



Source: by author (2015).

Cloud service business models offer a number of variations for the interaction between tenants and cloud providers (STRØMMEN-BAKHTIAR; RAZAVI, 2011), depending on how the tenant perceives and interacts with the service and the level of trust between parties. In some situations – such as in IaaS private clouds – both provider and tenant are part of the same organization, in which case access control and billing mechanisms may be relaxed. Moreover, in situations like this further details on the underlying infrastructure may be exposed to the *End-user* to be used at virtual infrastructure level. Since the focus of this thesis is on making resource management more flexible and integrated from the cloud *Administrator* point of view, this relationship is kept as simple as possible. Some aspects, such as billing or protecting the cloud provider from the tenant and vice-versa, are left out of scope. The sole responsibility of *End-users* is to describe their resource needs in the form of an *Initial Specification* (further detailed in Section 3.2) and it is up to the cloud provider to allocate and maintain the requested virtual resources accordingly.

Regarding the organization of cloud providers, again business models are flexible to accommodate a wide range of possibilities to providing virtual infrastructures as services to tenants (BUYYA et al., 2009). Some contexts consider, for example, the existence of service brokers that act on behalf of tenants to submit requests, often composing services from many providers as an integrated solution to their clients. Another example appears in the context of IaaS hybrid clouds, in which part of the resources are under the responsibility of the local provider and another part can be obtained from third parties. In this thesis the model for service providing is also simplified. A single provider scenario is assumed (*i.e.*, the provider has full control over all the resources available for provisioning services) and the tenant interacts directly with the provider using the platform's GUI. Integrating service provisioning with other cloud management platforms, for example via standardized interfaces (*e.g.*, OCCI or CIMI), to form a hybrid cloud scenario would not be impracticable. However, the level of control over resource management allowed by such platforms via remote interfaces is reduced, so *Resource Management Programs & Metrics* (see Section 3.3) would have to operate with limited capabilities.

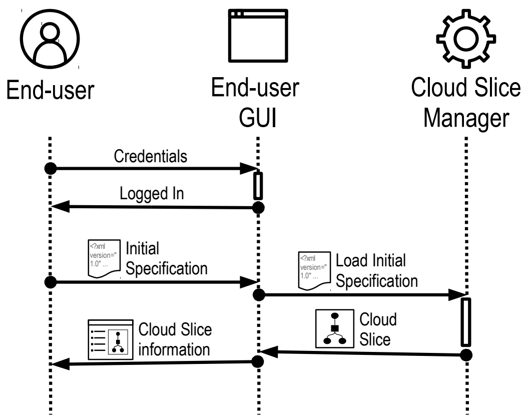
In general, the primary role of an *Administrator* in any cloud environment is to maintain both virtual and physical infrastructures up and running. In this thesis the *Administrator* is also attributed with the task of writing or editing *Resource Management Programs & Metrics* (further detailed in Section 3.3). Depending on each scenario and the complexity that these programs and metrics might acquire this task can be in fact performed by an *Administrator* or a third actor, say a *Cloud Developer*, could be envisioned. This actor would be the type of professional with extensive working knowledge from both application and infrastructure levels, thus capable of being the interface between the *End-user* and the *Administrator* for matters of adapting the service management according to the tenant's demand. The addition of this *Cloud Developer* role is very much in line with the emerging DevOps development philosophy (HÜTTERMANN, 2012).

3.2 Initial Specification of a Cloud Slice

As a first step to establish a virtual infrastructure over an IaaS cloud, the *End-user* needs to specify what are the resource requirements for his/her application or service to be carried out properly. Figure 3.2 details in a sequence diagram the interaction between the

End-user and the conceptual architecture of a cloud platform in order to input an *Initial Specification* of the resources required by an application or service.

Figure 3.2: Sequence diagram for the initial specification of a cloud slice



Source: by author (2015).

Initially, the *End-user* interacts with the *GUI* to enter log in information in order to gain access to the system. Afterwards, the *End-user* is able to input an *Initial Specification* document. This document should reflect, as much as possible, the requirements of the service or application that will run over the virtual infrastructure it describes. In the context of this thesis the contents of an *Initial Specification* are expected to be a description of all the resources that compose a single virtual infrastructure, including all possible types of allocable resources (*e.g.*, computing, storage, and networking) and their interconnections. A few standards are under way that can be used for describing complete virtual infrastructures, such as VXDL, OCCI, and CIMI, which have been extensively discussed in Section 2.2 of this thesis. Of course, not all applications have obvious requirements that can be easily mapped to a virtual infrastructure setup. Therefore, an intuitive interface to help with the specification of such requirements is desirable, but it is also out of the scope of this thesis. Recent studies are actually focused in finding means to describe high-level application requirements to be then decomposed into cloud resource allocations, some of which have been briefly discussed in Section 2.5.

This description of resources provided by the *End-user* is called *Initial Specification* because it is used only as a first description of the set of resources to be allocated. The actual allocation may differ from what has been specified depending on decisions taken during the deployment phase (further detailed in Section 3.4). Likewise, other information can be added or edited by the *End-user* or the *Administrator* once the *Initial Specification* is loaded into the system (*e.g.*, additional information on the management infrastructure required by the application). Moreover, optimizations may occur after the resources are allocated in order to enable elasticity, for example (further detailed in Section 3.5).

After the submission of the *Initial Specification* document, the *Cloud Slice Manager* component of the architecture parses this document, turning it into an internal representation. In this work this internal representation is called a *Cloud Slice*. The concept of *slice* is well known in both server and network virtualization environments, but among cloud related proposals this concept varies in interpretation. In the specific context of this research a *Cloud Slice* is defined as an aggregation of all different kinds of resources that compose the virtual infrastructure over which an application is deployed (*i.e.*, computing, storage, networking, and management resources). A *Cloud Slice* is dynamic by principle. It can be created and destroyed upon request of an *End-user* at anytime. Also, a *Cloud Slice* can be modified by moving, adding, or removing virtual resources during application runtime, which is desirable for resource optimization or to improve the performance of the application itself.

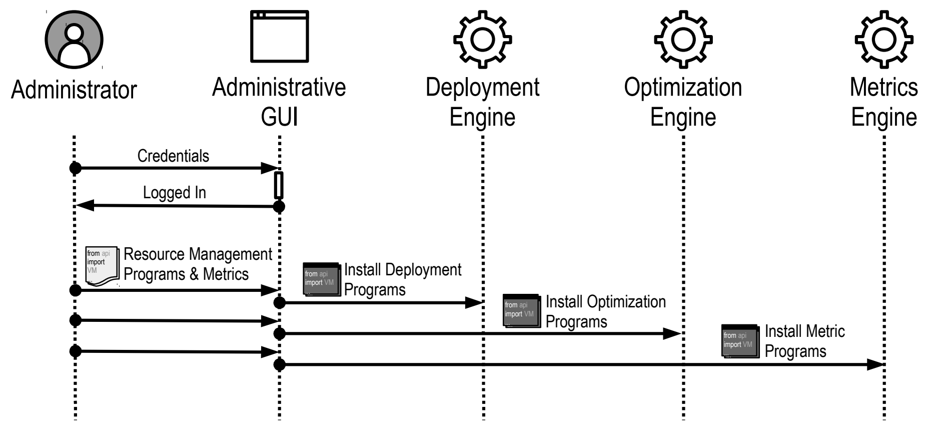
At the end of this step, the *Cloud Slice* is created and all internal control structures are ready for the actual deployment. The *End-user* is granted with access to all the information related to his/her *Cloud Slice* through the *GUI*. This is the time to check resource configuration, topology, and any other *Cloud Slice* information that has been submitted in the *Initial Specification* is received correctly. It is important to notice that, at this moment, no resource is yet allocated. Actual resource allocation happens only later at the deployment phase (detailed in Section 3.4).

3.3 Resource Management Programs & Metrics

After the creation of a *Cloud Slice*, the components in sections *Programmable Logic*, *Event Space*, and *Slice Space* of the architecture interact to organize resource management throughout the life-cycle of the application. The three main components of the architecture, which add flexibility to the core of the cloud management platform, are *Deployment Engine*, *Optimization Engine*, and *Metrics Engine*. To operate, these three components depend on *Resource Management Programs & Metrics* written by the *Administrator*. Figure 3.3 depicts the sequence diagram of interaction between the *Administrator* and the cloud platform to customize resource management via programmability.

First of all, the *Administrator* logs in to the *Administrative GUI* providing the appropriate credentials. After gaining access to the system, the *Administrator* is able to

Figure 3.3: Sequence diagram for installation of resource management programs and metrics



Source: by author (2015).

upload *Resource Management Programs & Metrics* through the same interface. These programs are of three distinct types: *Deployment Programs*, *Optimization Programs*, and *Metric Programs*. Each program is plugged in to a specific component of the platform as a regular software module and will be available for execution at the appropriate moment.

Deployment Programs are intended to calculate and allocate the resources required to deploy one particular *Cloud Slice* at a time. The logic employed by the program to calculate the resource allocation can be based on any algorithm one can implement with a standard programming language supported by the cloud platform. There are no specific constraints on how the program performs the deployment, the only requirement is that the program implements a specific interface for receiving information from the platform and returns, at the end of deployment, the expected return format to inform the final status of the deployment operation. When a *Deployment Program* is loaded and executed, the *Deployment Engine* is responsible for informing which *Cloud Slice* needs to be deployed – therefore no scheduling among *Cloud Slices* needs to be implemented in these programs – and controlling the execution of the program until its end. To retrieve information from the current status of the infrastructure (both virtual and physical) and to perform resource allocation operations *Deployment Programs* rely on direct calls to the *Unified API* (further detailed in Section 3.6).

Optimization Programs are implemented and installed by the *Administrator* similarly to how *Deployment Programs* are. The main difference between these two types of programs is their purposes. *Optimization Programs* are installed into the *Optimization Engine* component to be executed whenever the need to optimize resource allocation arises, as opposed to *Deployment Programs* that execute only during *Cloud Slice* deployments.

Moreover, *Optimization Programs* can operate under two different scopes: slice specific or global. Slice specific *Optimization Programs* need to be associated with a particular *Cloud Slice* and will perform changes only to the virtual resources allocated to this particular *Cloud Slice*, even though these programs can read information from any resource of the infrastructure. Global scope *Optimization Programs* can be used to optimize resource allocation across several *Cloud Slices*. These programs are mostly useful to optimize the overall resource consumption regarding the needs of the provider. Determining when an *Optimization Program* needs to be executed is part of the *Cloud Slice Optimization Cycle* (described in Section 3.5).

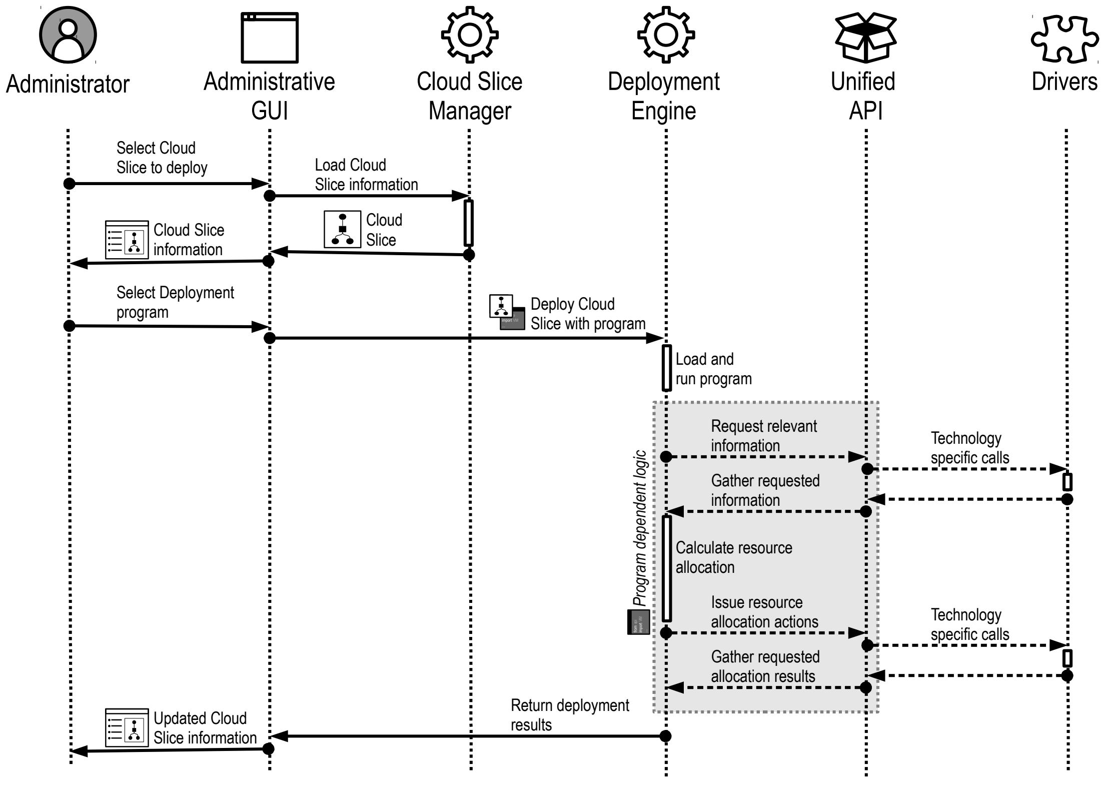
Metric Programs are intended to allow the *Administrator* to write customized metrics to be used specifically for monitoring resources and help gathering relevant information during *Cloud Slice* deployment or optimization. Similarly to other types of programs, *Metric Programs* need to implement a specific interface to be loaded and executed so that the metric value can be collected. However, the return value of a metric can be of any kind, ranging from numbers, booleans, or text, to lists or serializable information objects. To calculate the value of a metric, the program can employ any algorithm and rely on any information available from the *Unified API* to gather the state of one or more resources or even aggregate information from other *Metric Programs*. The conceptual architecture (Figure 3.1) also comprises the remote access to metric values. Therefore, a web service-like interface is made available, so that *Metric Programs* can have their computed values collected from outside the platform (from the external *Monitoring Infrastructure*, for example). Similarly to *Optimization Programs*, *Metric Programs* can as well be of global or slice specific scope. Slice specific metrics can be employed even to get information from inside the running application to be used, for example, to optimize resource allocation considering a particular application-oriented objective.

It is worth noting that writing and installing *Resource Management Programs & Metrics* can happen either before or after the *Initial Specification of a Cloud Slice* described earlier in this thesis. The existence of these programs is only required before the deployment or optimization of *Cloud Slices*. Even slice specific programs, after installed, can be associated at anytime with one or many *Cloud Slices* to fulfill their purpose. Another practical aspect regarding programs is to avoid having the *Administrator* to write all *Resource Management Programs & Metrics* from scratch, which could be a cumbersome task to perform. Ideally, a cloud management platform following the concepts of the proposed architecture should be shipped already with a basic set of general purpose programs and metrics. Therefore, only customizations to these already installed programs and metrics would be necessary to adapt resource management to fit environment or application specific needs. More detailed examples on how *Resource Management Programs & Metrics* are written are presented in Chapter 5.

3.4 Cloud Slice Deployment

Having fulfilled the first two steps to establish a virtual infrastructure over an IaaS cloud – creation of a *Cloud Slice* from an *Initial Specification* and completing the installation of at least one *Deployment Program* – it is then possible to allocate actual resources to run the *End-user's* application. For the deployment of a *Cloud Slice* to be accomplished several components of the proposed conceptual architecture need to interact, as depicted in the sequence diagram of Figure 3.4.

Figure 3.4: Sequence diagram for deployment of Cloud Slices using resource management programs


Administrator
Administrative GUI
Cloud Slice Manager
Deployment Engine
Unified API
Drivers
Select Cloud Slice to deploy
Load Cloud Slice information
Cloud Slice
Cloud Slice information
Select Deployment program
Deploy Cloud Slice with program
Load and run program
Program dependent logic
Request relevant information
Technology specific calls
Gather requested information
Calculate resource allocation
Issue resource allocation actions
Technology specific calls
Gather requested allocation results
Return deployment results
Updated Cloud Slice information

Source: by author (2015).

Although in this step most of the work is performed by customizable programs within the *Deployment Engine* component, the *Administrator* still plays a central role in selecting an appropriate program to deploy a specific *Cloud Slice*. Therefore, to start the deployment procedure, the *Administrator* needs to indicate one *Cloud Slice* that he/she wants to deploy and which *Deployment Program* should be employed for this procedure. This need for manual intervention is one side effect of having the flexibility to work with many resource management programs within a single platform. It is, of course, important to provide *Administrators* with the ability to interact and influence the process of resource management. However, for scalability reasons it is also fundamental to ensure that manual labor will not become a bottleneck to the system as a whole.

To make this part of the whole process more automated there are at least two possible alternatives: (*i*) having only one “default” *Deployment Program* active at a time or (*ii*) associating a class of applications to a specific *Deployment Program*. The first alternative can be considered the easy way out and may be used only to follow the specific objectives of the environment (*i.e.*, not application-specific). If the *Administrator* needs to change this objective, it is only a matter of selecting a new *Deployment Program* to be the “default” active one. The second alternative, on the other hand, adds further complexity to the process. First, it would be necessary to define classes of applications and appropriate programs to deploy each of them. Then it is also necessary to define which application falls into which class. For the sake of simplicity, this task could be attributed to someone with knowledge about the application's characteristics, *i.e.*, the *End-user* or to a *Cloud Developer* (in a scenario where one exists). Otherwise, a method to automatically determine the class of the application based, for example, on information of the *Initial Specification* could be proposed. So far in this research, the *Administrator* is assumed to be responsible for this manual selection. Future versions of the architecture may consider adding another component to perform the *Deployment Program* selection logic accordingly.

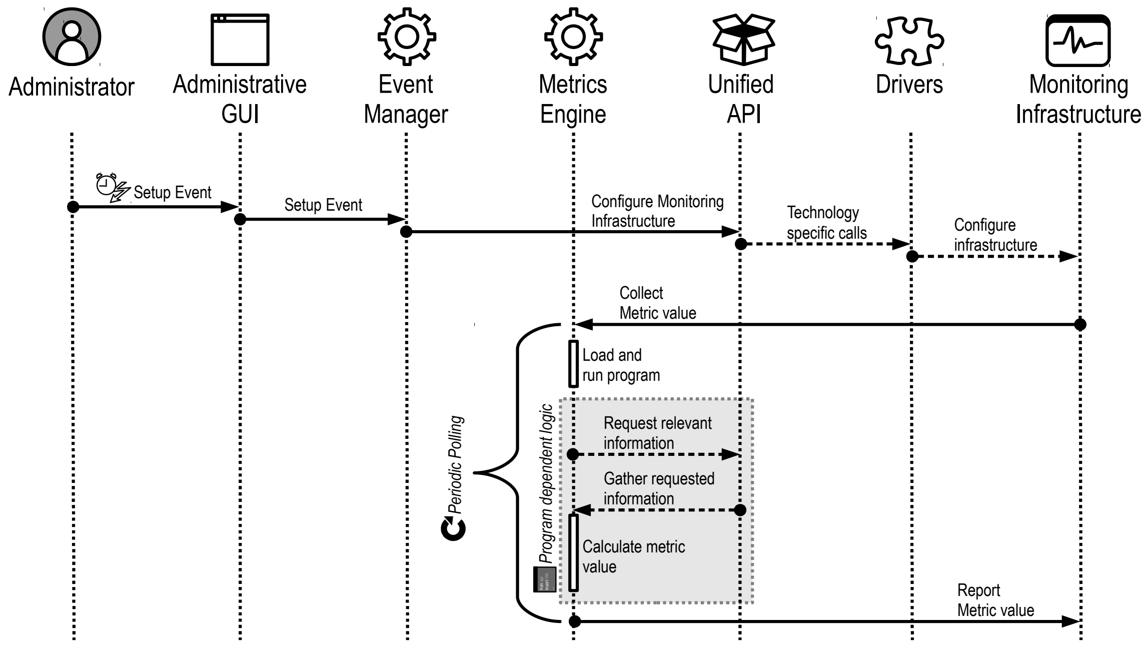
After the selection of a *Deployment Program*, the *Deployment Engine* component is responsible for loading and running the indicated program. As mentioned earlier in this thesis, the logic to deploy a *Cloud Slice* depends on the implementation the *Administrator* chooses to use, there is no specific method or algorithm enforced. Nevertheless, in general, a *Deployment Program* needs to perform at least the operations presented as an example in the shaded area of Figure 3.4. Initially, there are a set of calls to the *Unified API* in order to read the current status of the infrastructure and resource consumption. Then, resource allocation is calculated considering the information obtained (*e.g.*, find the best host for a virtual machine or the best route for a virtual link). Ideally this allocation should be somehow optimized according to an objective, either of the infrastructure or the application. Of course the allocation can be optimized later with *Optimization Programs*, but changes in a running application can always lead to service disruption. Finally, resources will be provisioned also issuing calls to the *Unified API* (*e.g.*, start a virtual machine or establish a virtual link). This process can naturally be also interactive, for example, where the program allocates a set of resources and reads again the status to recalculate future allocations.

While the deployment procedure is carried out, the *Deployment Engine* is mainly responsible for two tasks: (*i*) invoking the *Deployment Program* informing which *Cloud Slice* needs to be deployed and (*ii*) controlling the execution of the program. This invocation is performed through the interface mentioned in Section 3.3. Controlling the deployment execution means restraining the access to resources that the program can access for reading information or making changes. It is important that the deployment of one *Cloud Slice* does not affect others already deployed, although information from other *Cloud Slices* can be obtained and used for resource allocation calculations. Moreover, the *Deployment Engine* should also detect and report failures that can happen during the deployment procedure. The deployment of a *Cloud Slice* should be treated as an atomic operation, *i.e.*, the required resources should not be partially provisioned. For example, in concurrent deployments, one program might gather information about a set of resources right before a second program performs an allocation, which might cause the first program to run with outdated information and fail totally or partially. Despite the importance of fault control, mechanisms to provide such atomicity feature are currently left out of the scope of this thesis. Finally, at the end of deployment, the *Deployment Program* may gather and return statistics to the *Deployment Engine*, which in turn, will return these statistics along with overall updated information about the deployed *Cloud Slice* back to the *Administrator*.

3.5 Cloud Slice Optimization Cycle

After the deployment, the virtual infrastructure is ready to support the *End-user's* application. During application runtime, the need for optimization of resource allocation may arise. For example, when an application experiences performance degradation, virtual machines can be migrated closer to one another to reduce network latency. Differently from the deployment of *Cloud Slices*, which happens only once for the initial allocation of resources, optimization is a continuous process that forms a cycle with two main phases: (*i*) *Events* need to be configured and monitored to indicate the conditions under which optimization is necessary (Figure 3.5) and (*ii*) whenever that condition is satisfied, an *Optimization Program* comes into play in order to reorganize resource allocation accordingly (Figure 3.6).

Figure 3.5: Sequence diagram for event configuration using the external monitoring infrastructure (first phase of the optimization cycle)

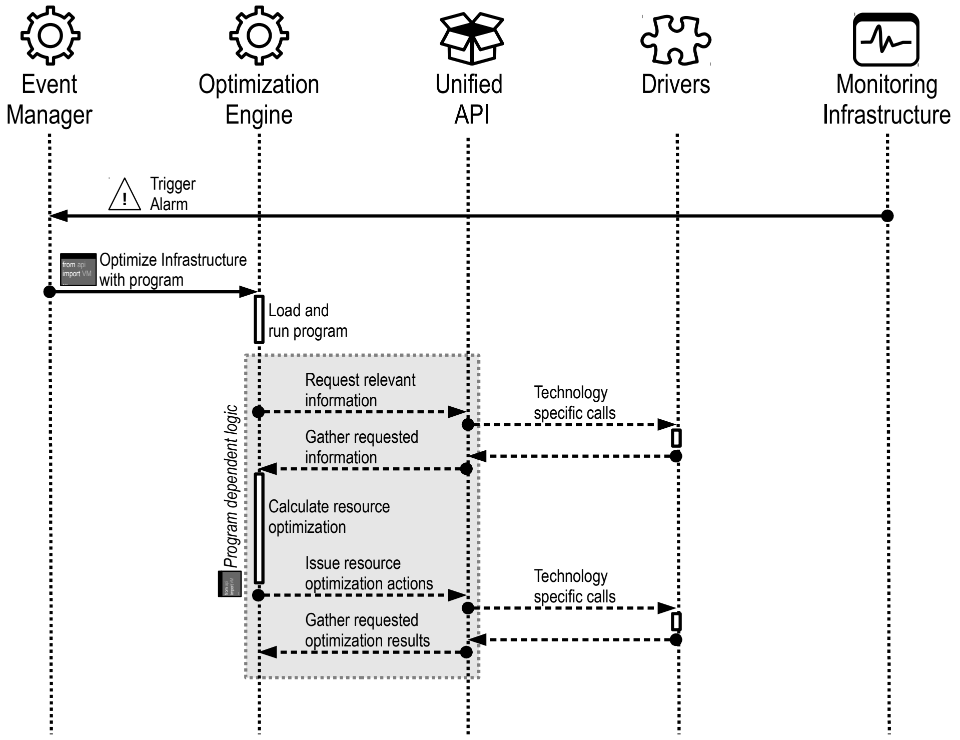


Source: by author (2015).

The first phase of the optimization cycle starts with the *Administrator* setting up an *Event* by interacting with the *Administrative GUI* as depicted in the flow diagram of Figure 3.5. A typical *Event* is composed of a condition (*e.g.*, a given metric has exceeded a certain threshold) and an associated *Optimization Program* (*e.g.*, rearrange virtual resources aiming to reduce the value of this metric). Conditions are expressed as one *Metric Program*, one relational operator (*e.g.*, greater than, equals to, less than) or comparison function (*e.g.*, contains, starts with, in/not in list), and one assigned value to be compared against. Whenever the condition holds (*i.e.*, evaluates to a true value), the *Optimization Program* associated with the *Event* needs to be triggered. This type of mechanism resembles Event-Condition-Action (ECA) systems commonly employed in Policy-based Network Management (PBMN) (TWIDLE et al., 2008). In addition, similarly to *Metric Programs* and *Optimization Programs*, *Events* can also belong to slice specific or global scope. Slice specific *Events*, are limited to use only *Metric Programs* and *Optimization Programs* that are also associated with the same *Cloud Slice* as the *Event* at hand.

The *Event Manager* component also needs to configure the appropriate settings within the *Monitoring Infrastructure* so that the condition can be periodically monitored and evaluated. As mentioned before, in Section 3.3 of this thesis, metrics in the proposed architecture are implemented as programs, which allows metric values to range from simple numerical measures gathered from the virtual or physical infrastructure (*e.g.*, the load average of a server), to very complex ones (*e.g.*, average server load per virtual machine ratio of the whole data center), including information from the running application (*e.g.*, load of a Web server running inside a virtual machine). Moreover, metrics are exposed as Web services so they can be collected directly from the *Monitoring Infrastructure*. When the *Monitoring Infrastructure* accesses this Web service the *Metric Program* is loaded and run similarly to other types of programs. The main difference is that a *Metric Program* will only be able to read information from the infrastructure using the *Unified API* and calculate its own value to return it at the end of execution. The computed value is reported by the *Metrics Engine* back to the *Monitoring Infrastructure* to generate statistics and evaluate the associated *Event* condition.

Figure 3.6: Sequence diagram for optimization of Cloud Slices using resource management programs (second phase of the optimization cycle)


Event Manager
Optimization Engine
Unified API
Drivers
Monitoring Infrastructure
Trigger Alarm
Optimize Infrastructure with program
Load and run program
Program dependent logic
Request relevant information
Technology specific calls
Gather requested information
Calculate resource optimization
Issue resource optimization actions
Technology specific calls
Gather requested optimization results

Source: by author (2015).

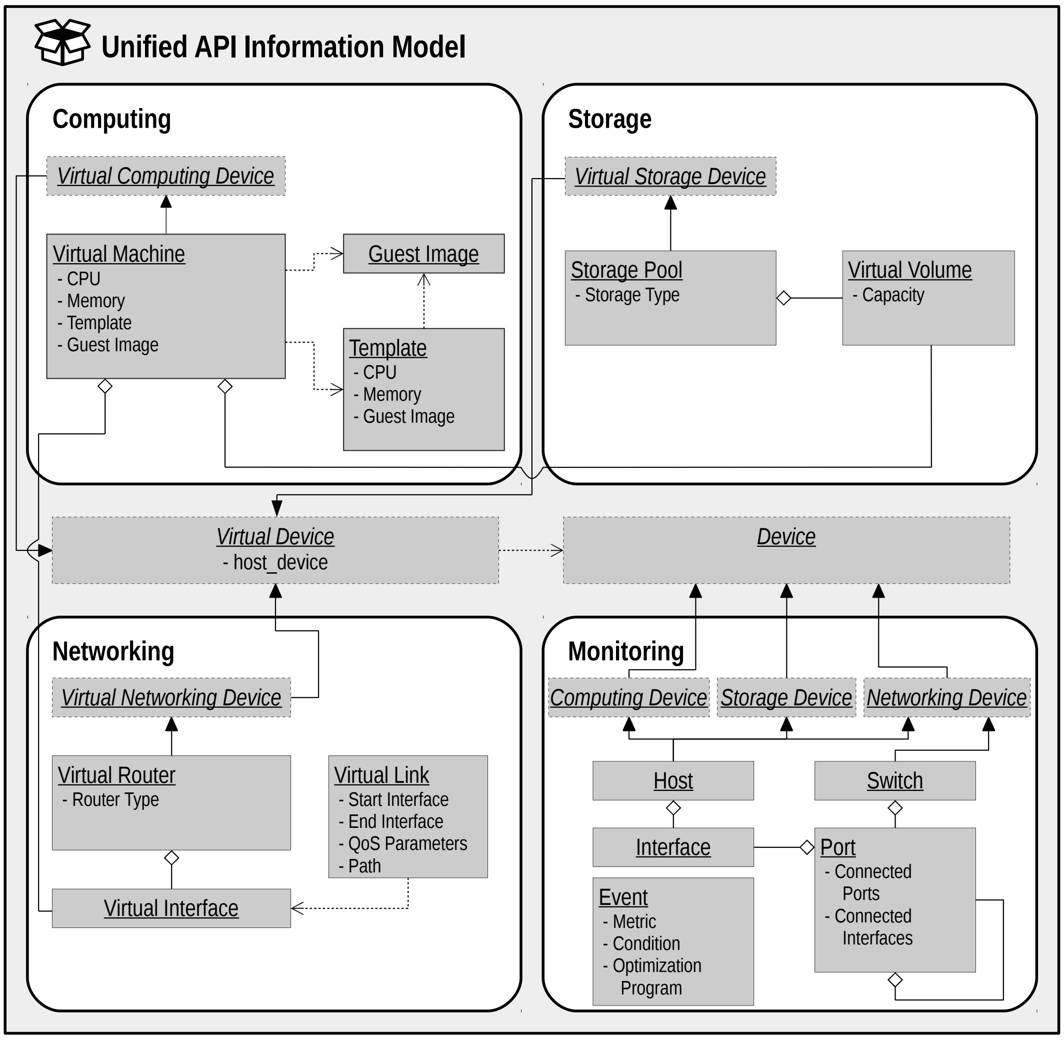
The second phase of the optimization cycle starts when a metric collected by the *Monitoring Infrastructure* reports a value that makes a condition hold. When that happens, the *Monitoring Infrastructure* automatically triggers an alarm informing that the condition for a particular *Event* has been satisfied. The *Event Manager* then starts the actual optimization of the infrastructure by loading and running the associated *Optimization Program*. These programs follow a similar execution flow as *Deployment Programs*, except that *Optimization Programs* can modify resources (*e.g.*, migrating virtual machines or rerouting virtual links), create new ones (scale up), or remove those already allocated (scale down).

Global scope *Optimization Programs* are not limited to changing only resources associated with any particular *Cloud Slice*. Of course, this situation can generate conflicting allocation decisions among different *Optimization Programs*. For instance, the *Administrator* can write a global program to move virtual machines to a small set of hosts and another one to optimize a specific *Cloud Slice* based on an application level metric that moves the same virtual machines apart again. To minimize the chance of conflicting or circular resource allocation decisions, three simple rules are followed in this thesis: (*i*) there is only one global scope *Optimization Program* active at any given moment, (*ii*) each *Cloud Slice* has only one active slice specific *Optimization Program* associated, and (*iii*) any *Cloud Slice* that is associated with an active slice specific *Optimization Program* will not be modified by a global scope *Optimization Program*. Finally, as opposed to what happens with the deployment of a *Cloud Slice*, no feedback after optimization is returned to the *Administrator* at the end of the process.

3.6 Unified Application Programming Interface (API)

All operations regarding handling of virtual resources performed during deployment and optimization of *Cloud Slices* make use of the proposed *Unified API*. This API aims to provide a simple interface with high-level abstractions for manipulating all types of resources that may compose a *Cloud Slice* at the same level of importance. Achieving this objective is fundamental to allow provisioning of virtual infrastructures with integrated support for different types of virtual resources and means to manage them, enabling the cloud platform to accommodate a wide range of applications. The proposed *Unified API* follows closely the concepts exposed in Section 2.1 and is organized into four main components: *Computing*, *Storage*, *Networking*, and *Monitoring*. Figure 3.7 displays in a class diagram the main abstractions and relationships included in each of these com-

Figure 3.7: Class diagram of the main abstractions and relationships provided by the *Unified API* organized into its four components



Source: by author (2015).

ponents, while Table 3.1 presents the main operations that can be performed over each abstraction.

The *Computing* component takes care of basic functionalities for handling three virtual resource abstractions: virtual machines, guest operating system images, and templates. A virtual machine is the main abstraction available through this component, which allows allocating primarily CPU cores and RAM memory from hosts of the data center. Any virtual machine needs to be hosted at a physical server of the infrastructure, whereas choosing an appropriate location to deploy these virtual resources is one of the main objectives of resource management programs. Virtual machines can also have virtual interfaces and attached virtual storage volumes (described next). From the API it is also possible to control the status (*e.g.*, starting, stopping, suspending) of the guest operating system of a virtual machine, among other operations detailed in Table 3.1.

Table 3.1: *Unified API* operations, parameters, and return values

Abstractions		Operations	Parameters	Return
Computing	Virtual Machines	**Create:** defines the internal representation of a virtual machine with its specified characteristics (*e.g.*, CPU, memory, guest image).	VXDL description (CPU, memory, disk, interface, etc.)	Reference to virtual machine created
		Remove: undefines a virtual machine previously created.	*--none--*	Success or failure
		Deploy: defines a virtual machine within the hypervisor of a node of the cloud infrastructure, including the transferring of the image file.	A destination node	Success or failure
		Undeploy: undefines a previously deployed virtual machine from a hypervisor, including the removal of the image file.	*--none--*	Success or failure
		Start: powers up a virtual machine, allocating the required resources, and starting the booting up process of its guest operating system.	*--none--*	Success or failure
		Stop: powers off a virtual machine, releasing the resources back to the hypervisor, and forcing the guest operating system down.	*--none--*	Success or failure
		Shutdown: signals the guest operating system to gracefully shutdown and then stops the virtual machine.	*--none--*	Success or failure
		Suspend: suspends the guest operating system and keeps resource allocated.	*--none--*	Success or failure
		Resume: resumes operation of a previously suspended virtual machine.	*--none--*	Success or failure
		Migrate: undeploys a virtual machine in one node and deploys it in another.	A destination node	Success or failure
	Guest Images	**Create:** defines a guest operating system image at the main repository of the platform, including the file transfer.	Path or URL	Reference to image created
		Remove: undefines a previously created image, removing the file from the repository.	*--none--*	Success or failure
	Templates	**Create:** defines a template of virtual machine, including the resource configurations (*i.e.*, CPU and memory) and an associated guest image.	CPU, memory, and image or VXDL description	Reference to template created
		Remove: undefines a template of virtual machine.	*--none--*	Success or failure
Storage	Virtual Volumes	**Create:** allocates chunks of storage on nodes.	VXDL description (capacity, destination pool, etc.)	Reference to virtual volume created
		Remove: deletes chunks of storage previously created.	*--none--*	Success or failure
		Attach to Virtual Machine: attaches a virtual volume to a given virtual machine.	Reference to a virtual machine	Success or failure
	Virtual Volume Pools	**Create:** defines a pool for storing virtual volumes (typically a local or remote/NFS directory).	Reference name, node, type	Reference to virtual pool created
		Remove: undefines a previously created virtual pool and all virtual volumes it contains.	*--none--*	Success or failure
Networking	Virtual Links	**Create:** defines the internal representation of a virtual link, which connects point-to-point two virtual interfaces of two virtual devices (*i.e.*, interfaces of virtual machines or virtual routers).	VXDL description (endpoints source and destination, and QoS parameters)	Reference to virtual link created
		Remove: undefines a previously created virtual link.	*--none--*	Success or failure
		Establish: establishes the virtual link within the network enabling traffic to flow between the connected devices.	*--none--*	Success or failure
		Disable: disables a previously created virtual link connection, stopping traffic flow between connected devices.	*--none--*	Success or failure
	Virtual Routers	**Create:** defines the internal representation of a virtual router, which has many virtual ports to interconnect many virtual interfaces of virtual devices.	VXDL description (type, ports, controller, etc.)	Reference to virtual router created
		Remove: undefines a previously created virtual router.	*--none--*	Success or failure
		Deploy: deploys the virtual router into a node of the infrastructure.	A destination node	Success or failure
		Undeploy: undeploys a previously deployed virtual router from the infrastructure.	*--none--*	Success or failure
Monitoring	Virtual Devices	**Monitor:** deploys the monitoring infrastructure required to monitor a given virtual device.	The virtual device to monitor	Success or failure
		Unmonitor: undeploys the monitoring infrastructure associated with a given virtual device.	The monitored virtual device	Success or failure
		Get Monitoring Information: fetches monitoring information within the monitoring system for a given virtual device.	The monitored virtual device	FlexACMS XML information
	Events	**Create:** defines the internal representation of an event, which may belong to a specific slice or operate in global scope.	Metric, operation, threshold, program, and cloud slice	Reference to event created
		Remove: undefines a previously created event.	*--none--*	Success or failure
		Deploy: deploys an event on the monitoring infrastructure to be triggered on demand.	*--none--*	Success or failure
		Undeploy: undeploys a previously deployed event from the monitoring infrastructure.	*--none--*	Success or failure
	Physical	**Discover Resources:** discover nodes and network topology available on the infrastructure. This collection also retrieves information about resource allocation on these physical elements.	*--none--*	List of nodes, links, switches, and resource allocations
		Get Monitoring Information: fetches monitoring information within the monitoring system for a given physical device (*e.g.*, node or switch).	The monitored physical device	FlexACMS XML information

Source: by author (2015).

Every virtual machine has a copy of one guest operating system image, which is another abstraction that can be handled through the *Unified API*. These images contain a pre-installed operating systems and can be used to facilitate the deployment of complex virtual appliances (*e.g.*, a virtual machine configured to act as a web server or a packet inspector). Templates are a third abstraction that are handy to size virtual appliances by combining a resource configuration (*e.g.*, amount of RAM and CPU) with a guest image and to create several virtual machines with the same characteristics.

The *Storage* component deals with two abstractions: virtual volumes and storage pools. Virtual volumes are capable of storing data up to a certain defined capacity and can be attached to virtual machines. Similarly to virtual machines, virtual volumes also need to be allocated from physical hosts of the infrastructure, except that they are not created from a hypervisor but allocated from a storage pool. Therefore, for performance reasons, positioning virtual volumes in relation to the virtual machines to which they will be associated is a fundamental decision resource management programs need to make. The required operation to associate a virtual volume with a virtual machine is available through the proposed API as detailed in Table 3.1. Storage pools are containers to store virtual volumes. This abstraction can be used to define different types of storage (*e.g.*, NFS share, local folder, SSD device) from which chunks of data (virtual volumes) can be drawn. In general, every physical host of the infrastructure that is capable of holding virtual volumes should contain at least one storage pool. Moreover, usually only virtual volumes will be part of *Cloud Slices* and there will be a number of available storage pools pre-defined by the *Administrator*.

The *Networking* component implements interfaces for creating two types of abstractions: virtual routers and virtual links. The virtual router abstraction is intended to mimic a network device, containing a set of virtual interfaces, and capable of connecting and forwarding traffic among other virtual devices. There are many possible types of virtual forwarding devices that can be implemented depending on the drivers available in the *Drivers* section of the architecture. Virtual routers based on software platforms (*e.g.*, Open vSwitch (Open vSwitch, 2012) or Click Modular Router (KOHLER et al., 2000)) need to be deployed at a physical host of the infrastructure using the deployment operation of the API. Virtual links are an abstraction used to establish a logical connection between two virtual endpoints. Without a virtual link, for example, two virtual machines are not allowed to communicate, even if they belong to the same *Cloud Slice*. Virtual links can also connect a virtual machine to a virtual router interface or two virtual routers directly. It is also possible to configure QoS parameters of virtual links, whereas some of these parameters are possible to configure within the infrastructure (*e.g.*, maximum and committed bandwidth rates) others depend on resource management programs to be assessed and satisfied (*e.g.*, average latency in communication).

The proposed conceptual architecture includes a fourth component as part of the *Unified API* to enable *Monitoring* to be considered as an “allocable” type of resource. This is quite unusual in most cloud platform designs; however, it is important to provide valuable information about resources, specially for deployment and optimization programs. Whenever a *Cloud Slice* is deployed, the corresponding monitoring infrastructure can also be configured with the appropriate calls to the API. Monitored information can be retrieved directly from each virtual device abstraction (*e.g.*, from virtual machines or virtual routers). Also, this component provides two abstractions to read information from the physical infrastructure: hosts and switches. Physical hosts may be able to support virtual computing, storage, or networking (*i.e.*, software routers/switches) devices. From hosts it is possible to retrieve information about their capabilities, hypervisors, and resource allocations. From switches and their connections it is possible to gather information about topology and network traffic. Moreover, this component also implements an event abstraction, which sets up alarms to be triggered from within a *Monitoring Infrastructure* back into the *Event Manager* component of the architecture (detailed in Section 3.5). The interactions between the proposed architecture and the associated *Monitoring Infrastructure* that can be performed with the abstractions described are presented in Chapter 4.

One last point that is worth mentioning is that other information models exist in the literature for representing virtualized infrastructures such as the ones included in DMTF's CIMI and OVF standards. CIMI is particularly complex for the intended usage of the proposed *Unified API* which strives for simplicity, while OVF is very much focused in computing resources. Although the information model presented in this section is not strictly aligned with a particular standard, it is heavily based on concepts earned from VXDL and Libvirt. In addition, the *Unified API* model is carefully designed to include just enough information to remain simple, while providing an adequate level of abstraction for managing the virtualized infrastructure objects in detail. Also, regarding the operations included in the proposed API, there are standards being discussed in the context of the cloud research community and standardization bodies (*e.g.*, OCCI) that include a similar set of remote-call type of resource management operations. However, such standards are intended to be used from the *End-user* (or the user-owned application) point of view, remotely sending resource requests from outside a given cloud system. Although the operations are similar, the *Unified API* is intended for a slightly different purpose, *i.e.*, to be used by the cloud *Administrator*, which requires higher levels of detail and advanced control mechanisms over both virtual and physical resources.

3.7 Technology Specific Drivers

All virtual device abstractions described in the previous section (*e.g.*, virtual machines, virtual routers, and virtual links) are implemented on the underlying infrastructure by a set of *Drivers* available at the bottom section of the architecture introduced throughout this chapter. Each *Driver* module is a technology specific piece of code that plays two main conceptual roles: (*i*) to abstract the complexity of technology specific configuration parameters and communication protocols from the *Unified API* and (*ii*) to make the cloud management platform more portable, *i.e.*, *Driver* modules can be replaced as technology evolves, as long as the provided set of functionalities remains the same.

As Figure 3.1 indicates, one *Driver* can implement a set of abstractions from different components of the architecture, as well as one abstraction may require many *Drivers* combined in order to be realized. For example, to establish virtual links, more than one *Driver* module may be necessary to connect two endpoints attached to different network technologies. Moreover, the *Administrator* is responsible for choosing and configuring beforehand the appropriate *Driver* modules that will be employed in each situation. This is important to avoid resource management programs having to choose what type technology should be used to materialize abstractions (*e.g.*, whether a virtual machine should be deployed using XEN or KVM). One last important point to emphasize is that, the set of *Driver* modules displayed in the conceptual architecture in Figure 3.1 are merely examples of technologies – which have been actually implemented in the platform detailed in Chapter 4 – that allow the establishment of the virtual device abstractions over a real infrastructure.

3.8 Summary

This chapter presented an overview of the conceptual architecture proposed in this thesis, which organizes the components that are fundamental to achieve more flexible and integrated resource management in IaaS cloud platforms. The main novelty of this research is not the architecture itself, but in the new functionality added by some of the components arranged within its sections. Some concepts, like separating resource management concerns into computing, storage, and networking, are commonly employed in almost every cloud platform architecture. However, three components of the architecture are mainly responsible for adding flexibility in resource management to the core of the platform are: *Deployment Engine*, *Optimization Engine*, and *Metrics Engine*. In these components, customizable *Resource Management Programs & Metrics* are installed, loaded, and executed. In addition, the bottommost sections of the architecture, *Unified API* and *Drivers* combined allow both integrated resource management through high-level abstractions of virtual devices and portability for the platform to cope with heterogeneous types of environments and technologies. Finally, the proposed architecture also interacts with a specialized external *Monitoring Infrastructure* in order to obtain advanced monitoring.

It is worth noting that no components are included in the proposed conceptual architecture specifically to manage physical resources. The underlying physical infrastructure forms an important part of the whole IaaS provisioning environment, whereby managing virtual resources encompasses understanding and handling information from both physical and virtual levels (*e.g.*, underlying topology, switching capabilities, disk access speed). Although management of physical resources is not explicitly considered, operations such as monitoring resource usage and allocating shares of computing, storage, and networking resources are present. In this thesis, these physical resources are assumed to be managed by any external system, although components to assist in bootstrapping, configuring, and optimizing physical hosts and switches may be included in future versions of the architecture. One last point worth of note is that the whole infrastructure is assumed to be controlled by the same administrative entity. Therefore, there is no information available from the managed infrastructure that cannot be used by the platform because of policy or security matters. Of course in a multi-provider or multi-datacenter scenario, which is not considered in this thesis, this assumption would not hold and interesting challenges could arise in making distributed management of resources and collaboration among platforms possible.

4 CLOUD MANAGEMENT PLATFORM IMPLEMENTATION

As a proof of concept, a cloud management platform – which is called Aurora Cloud Manager[1] – has been implemented following the design aspects of the conceptual architecture presented in Chapter 3. Most of the implementation has been performed using the Python programming language. Moreover, several third party tools, libraries, and systems have been brought together in order to materialize the proposed concepts. The decision not to rely on a complete platform, such as OpenStack or OpenNebula, was taken to avoid inheriting the internal complexity of such platforms. Lower level libraries and APIs, such as Libvirt and OpenFlow, provide sufficiently powerful abstractions for node virtualization and networking operations that the Aurora platform requires. This chapter presents, at a first moment, an overview of the interactions of the Aurora platform with other external systems. After that, the initial specification of a *Cloud Slice* based on VXDL is detailed. Following, the implementation of the core components to allow execution of *Resource Management Programs & Metrics*, the proposed API and its operations, and technology specific drivers are discussed.

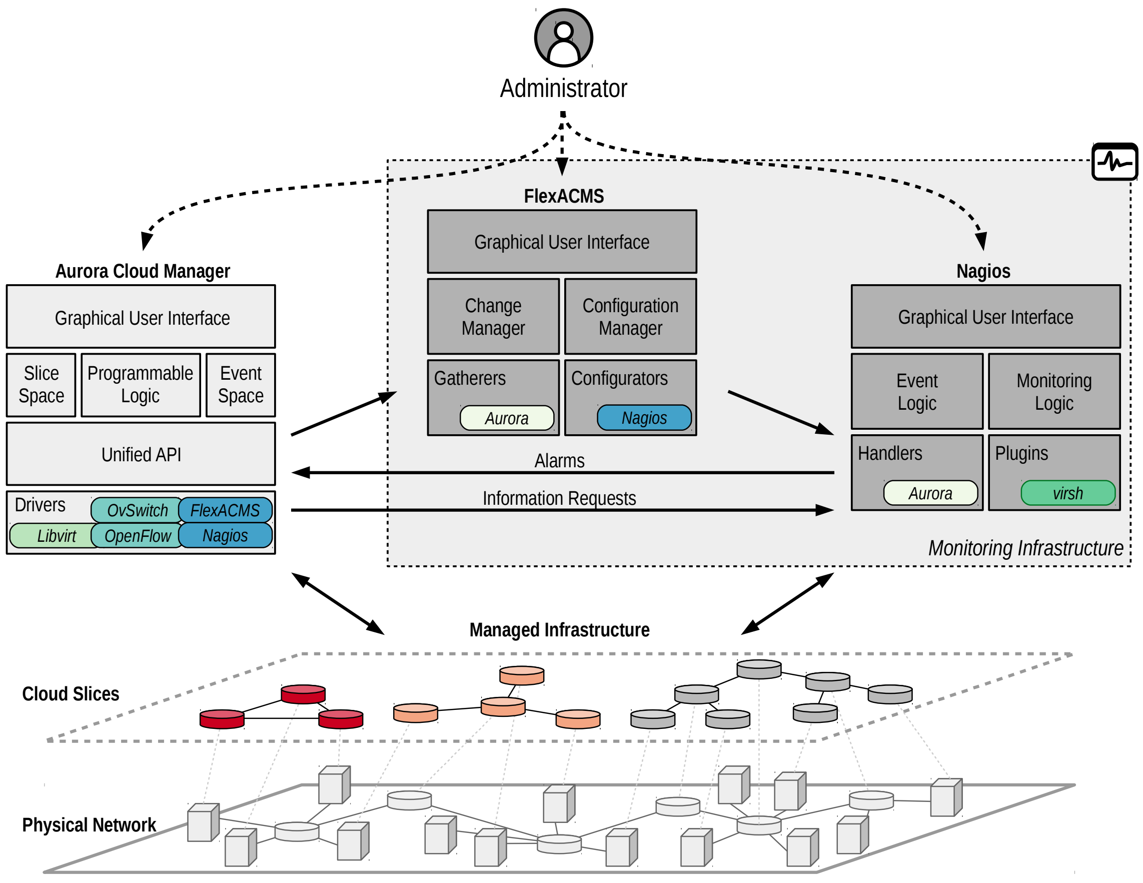
4.1 Aurora Platform Interactions Overview

Figure 4.1 shows an overview of the interactions between the Aurora platform and two other systems that compose the *Monitoring Infrastructure*. As previously indicated by the disposition of components in the conceptual architecture, monitoring functions are not implemented directly into the platform's core, since there are complete solutions for that. Instead, Aurora relies on a tool called Flexible Automated Cloud Monitoring Slices (FlexACMS) (CARVALHO et al., 2012) that builds *Monitoring Slices* automatically to reflect a *Cloud Slice* creation. This configures a scenario where management is taken as an allocable type of resource (as explained in Section 2.1), more specifically a Monitoring as a Service scenario. Therefore, a *Monitoring Slice* reflects the corresponding monitoring metrics and configurations in diverse monitoring systems that are necessary to properly monitor all resources of a *Cloud Slice*.

FlexACMS uses a specifically designed *gatherer* to collect an XML description from Aurora, containing information about *Cloud Slices*. The received information is used to detect changes that impact on *Monitoring Slices* (*e.g.*, a *Cloud Slice* creation),

[1] Source code available at: <https://github.com/ComputerNetworks-UFRGS/Aurora>

Figure 4.1: Interactions between the Aurora platform and the Monitoring Infrastructure



Source: by author (2015).

then FlexACMS instructs *configurators* to deploy or update the corresponding *Monitoring Slice*. *Monitoring Slices* are built based on predefined rules and are able to monitor any type of metric to be used by resource management programs, such as CPU consumption or network interface statistics. FlexACMS also provides several predefined templates for monitoring virtual elements, which can be synchronized with *Template* abstraction of the proposed API to inform how virtual machines should be monitored. For example, if a virtual machine is an HTTP or mail server, there will be specific related metrics that should be monitored in order to assure the correct operation of the specific service. This information can also be passed through the XML description for automatic configuration of the monitoring infrastructure accordingly.

In this implementation, FlexACMS builds the configuration of *Monitoring Slices* and their metrics based on Nagios (NAGIOS, 2013). Nagios was chosen to be the system that actually monitors both virtual and physical layers of the *Managed Infrastructure*. This monitoring system has been chosen mainly because (*i*) it is widely employed in large scale real infrastructures, (*ii*) it has all the built-in features needed to trigger events and configure polling intervals, and (*iii*) it is easily extensible via plug-ins. Nagios was also configured to monitor the underlying physical infrastructure. This configuration was performed through FlexACMS using the same XML specification, but with specific templates for physical nodes and switches. As shown in Figure 4.1, the *Administrator* needs to interact with all three systems to access and configure different platforms. So far, this configuration allows only the *Administrator* to access visual monitoring information (*e.g.*, charts, alerts, and tables) regarding managed physical and virtual resources. In future versions of the Aurora platform, relevant monitoring information should be made available through GUIs, in order to provide both *End-users* and *Administrators* with handy information associated with *Cloud Slices*.

4.2 VXDL-based Initial Specification

Before presenting details on the implementation of the core components of the Aurora platform, this section introduces the *End-user*'s input to the system. The *Initial Specification* of a virtual infrastructure is described in this platform according to an extension of the Virtual Infrastructure Description Language (VXDL) (VXDL Forum, 2011; KOSLOVSKI; SOUDAN; VICAT-BLANC, 2012). This language allows the detailed specification of many types of virtual elements, such as virtual machines (`vNode`), storage (`vStorage`), routers (`vRouter`), links (`vLink`), and access points (`vAccessPoint`). Based on these elements, it is possible to create a complete topology of virtual elements. Currently, Aurora does not include a graphical interface to design the virtual topology, which means the *End-user* (or the *Administrator* in his/her behalf) needs to prepare a VXDL file elsewhere and then submit it to the platform.

Figure 4.2 presents sample pieces of VXDL files that an *End-user* can use to specify a virtual topology. On the left side, the piece of XML shows how to define a virtual machine (`vNode`) with 1 CPU, 128MB of RAM, and 500MB of maximum storage capacity. Moreover, this piece of VXDL code also specifies the OpenWRT (Backfire release) image used to deploy the virtual machine and some properties of the virtual network interface to be created. The VXDL code on the right is used to define a virtual link (`vLink`) between two `vNodes` (source/destination). Link properties, such as upload/download bandwidth and latency, may also be specified with this language. In the implemented platform, this information can be used by resource management programs to compute link utilization in terms of allocated bandwidth, for example. Also virtual routers (`vRouters`), which are not included in the VXDL samples, are supported by the platform and are implemented as Open vSwitches. A previous investigation has exploited the dynamic creation of virtual routers as OpenFlow switches controlled from outside the platform (JESUS; WICKBOLDT; GRANVILLE, 2013).

Figure 4.2: Sample of initial specification with VXDL

```
<vNode id="Node0">
  <cpu>
    <cores>
      <simple>1</simple>
    </cores>
    <frequency>
      <simple>1</simple>
      <unit>GHz</unit>
    </frequency>
  </cpu>
  <memory>
    <simple>128</simple>
    <unit>MB</unit>
  </memory>
  <storage>
    <interval>
      <min>500</min>
    </interval>
    <unit>MB</unit>
  </storage>
  <image>OpenWrt-Backfire</image>
  <interface>
    <alias>net0</alias>
    <type>bridge</type>
  </interface>
</vNode>
```

```
<vLink id="Link0">
  <bandwidth>
    <forward>
      <interval>
        <min>10.0</min>
      </interval>
      <unit>Mbps</unit>
    </forward>
    <reverse>
      <interval>
        <min>5.0</min>
      </interval>
      <unit>Mbps</unit>
    </reverse>
  </bandwidth>
  <latency>
    <interval>
      <max>2.0</max>
    </interval>
    <unit>ms</unit>
  </latency>
  <source>
    <vNode>Node0</vNode>
    <interface>net0</interface>
  </source>
  <destination>
    <vNode>Node1</vNode>
    <interface>net0</interface>
  </destination>
</vLink>
```

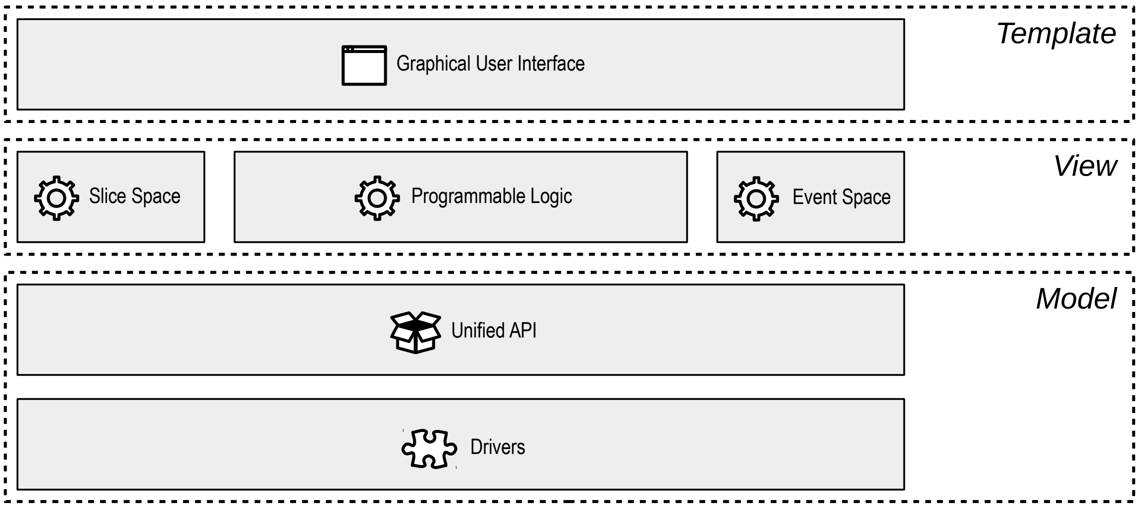
Source: by author (2015).

4.3 Core & Resource Management Programs

The core of the Aurora platform is written in Python and implemented as a Web-based application. A three-layered *Model-View-Template (MVT)* design model based on a framework called Django (Django, 2013) has been employed to organize the main components implemented. Figure 4.3 shows how every section of the conceptual architecture proposed in Chapter 3 fits in each layer of the design model chosen. *Templates* are used to create GUIs, most of the logic for resource management is implemented as *Views*, while the API abstractions and technology drivers are implemented as *Models*.

The *Template* layer holds the implementation of GUIs, where both *Administrators* and *End-users* are able to perform operations over virtual resources, such as checking virtual machine status, establishing virtual links, or requesting the creation of a *Cloud Slice* based on a VXDL description. Some samples of these interfaces are presented in

Figure 4.3: Conceptual architecture sections organized in the Model-View-Template design model



Source: by author (2015).

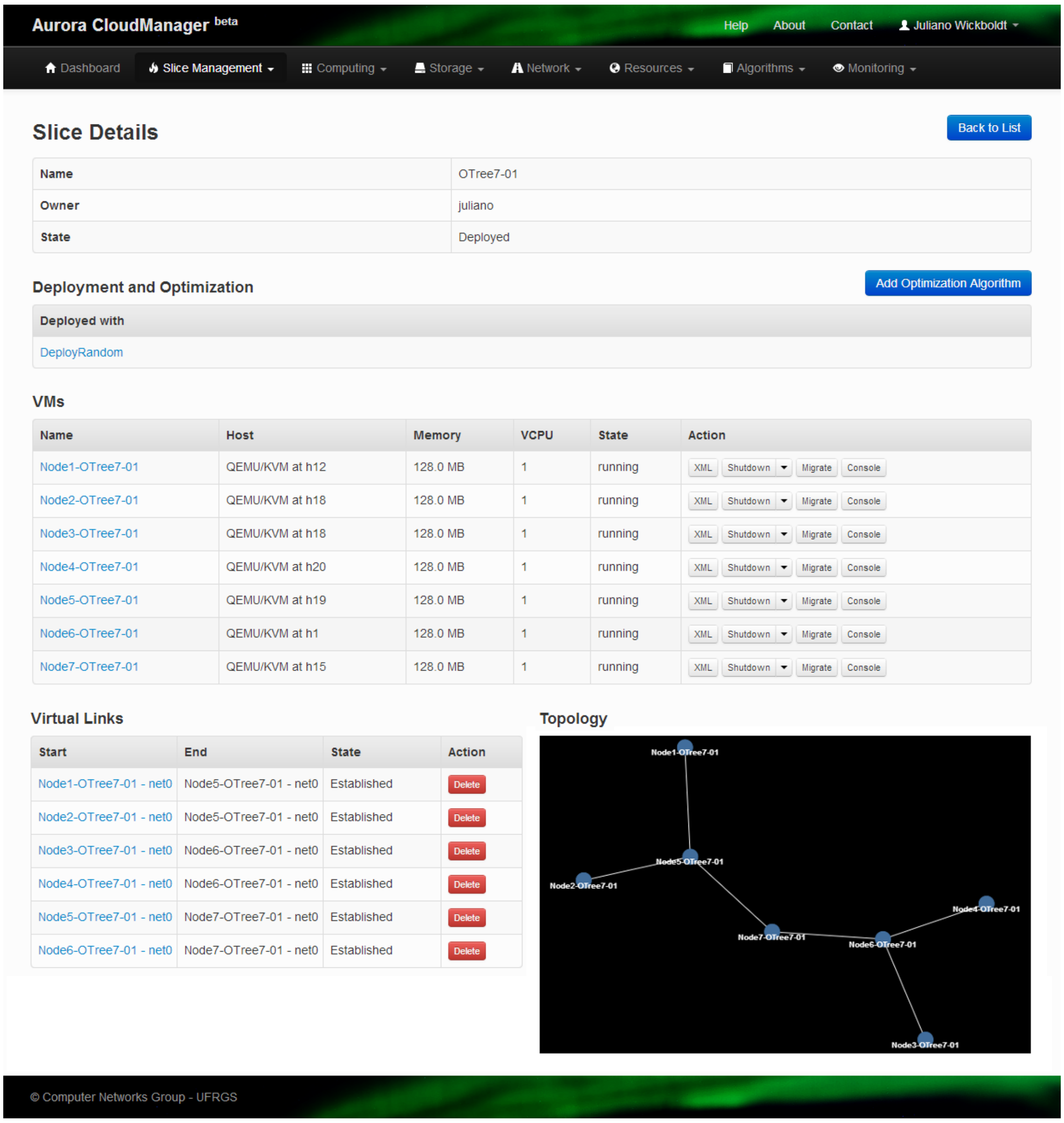
Figures 4.4 and 4.5. Figure 4.4 shows Aurora's slice management interface used by both *End-user* and *Administrator* to handle virtual resources and visualize the virtual topology of a *Cloud Slice*.

Figure 4.5 presents the interface for editing optimization programs. The edition of programs and metrics through the platform is currently performed in plain text. Future versions of the platform may be enhanced with IDE-like features, including integration with the API calls and debugging options, which can assist in writing resource management programs and metrics.

The *View* layer is where most of the logic of the platform is implemented, including basic functions for managing individual virtual and physical resources (*i.e.*, starting, stopping, migrating virtual machines, configuring access to hypervisors and network controllers). Also at this layer, the base framework for loading and running *Resource Management Programs & Metrics* is implemented. The implementation of *Deployment Engine*, *Optimization Engine*, and *Metrics Engine* components encompasses the selection of the appropriate programs for each specific situation (*i.e.*, deployment or optimization of *Cloud Slices*), as well as loading these programs and controlling their execution. Controlling the execution of a resource management program involves restraining scope of access to resources and handling exceptions that can happen during operations. Moreover, specifically for *Metric Programs* a JSON-based Web service access is implemented within the *Metrics Engine* component to allow remote calls to collect metric values.

Resource Management Programs & Metrics are implemented in Python and installed directly into the platform's core components. For the person who designs these

Figure 4.4: Screenshot of the cloud slice graphical user interface



Slice Details

Name	OTree7-01
Owner	juliano
State	Deployed

Deployment and Optimization

Deployed with
DeployRandom

VMs

Name	Host	Memory	VCPU	State	Action
Node1-OTree7-01	QEMU/KVM at h12	128.0 MB	1	running	XML Shutdown Migrate Console
Node2-OTree7-01	QEMU/KVM at h18	128.0 MB	1	running	XML Shutdown Migrate Console
Node3-OTree7-01	QEMU/KVM at h18	128.0 MB	1	running	XML Shutdown Migrate Console
Node4-OTree7-01	QEMU/KVM at h20	128.0 MB	1	running	XML Shutdown Migrate Console
Node5-OTree7-01	QEMU/KVM at h19	128.0 MB	1	running	XML Shutdown Migrate Console
Node6-OTree7-01	QEMU/KVM at h1	128.0 MB	1	running	XML Shutdown Migrate Console
Node7-OTree7-01	QEMU/KVM at h15	128.0 MB	1	running	XML Shutdown Migrate Console

Virtual Links

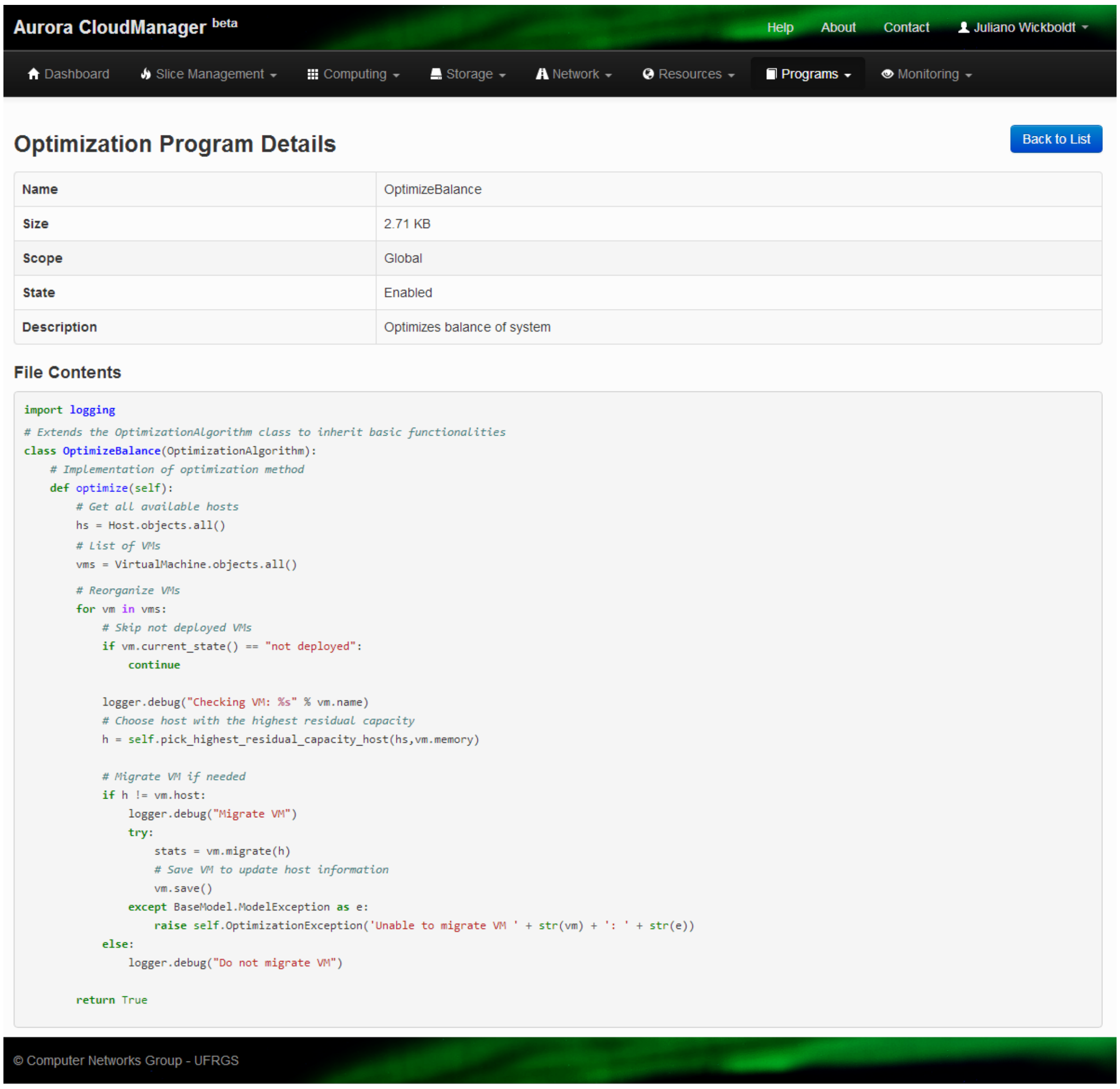
Start	End	State	Action
Node1-OTree7-01 - net0	Node5-OTree7-01 - net0	Established	Delete
Node2-OTree7-01 - net0	Node5-OTree7-01 - net0	Established	Delete
Node3-OTree7-01 - net0	Node6-OTree7-01 - net0	Established	Delete
Node4-OTree7-01 - net0	Node6-OTree7-01 - net0	Established	Delete
Node5-OTree7-01 - net0	Node7-OTree7-01 - net0	Established	Delete
Node6-OTree7-01 - net0	Node7-OTree7-01 - net0	Established	Delete

Topology

Source: by author (2015).

programs and metrics, the platform provides at the *Model* layer a high-level object-oriented API that includes all sorts of resource management operations. In fact, most platforms include remote-call types of APIs (*e.g.*, REST or SOAP) to request cloud resources. In general, these APIs differ greatly from the one proposed in this thesis because they are intended to be used by someone from outside the cloud platform domain (*e.g.*, an End-user's application requesting resources), so they usually hide many internal details of resources. The decision to use a programming language instead of remote calls is intended to provide easier access to both physical and virtual resources, their attributes, and operations to the *Administrator*. This allows, for example, the *Administrator* to try out individual commands of the API at runtime using an interactive auto-complete console interface of

Figure 4.5: Screenshot of the optimization program graphical user interface



Optimization Program Details

Name	OptimizeBalance
Size	2.71 KB
Scope	Global
State	Enabled
Description	Optimizes balance of system

File Contents

```
import logging
# Extends the OptimizationAlgorithm class to inherit basic functionalities
class OptimizeBalance(OptimizationAlgorithm):
    # Implementation of optimization method
    def optimize(self):
        # Get all available hosts
        hs = Host.objects.all()
        # List of VMs
        vms = VirtualMachine.objects.all()

        # Reorganize VMs
        for vm in vms:
            # Skip not deployed VMs
            if vm.current_state() == "not deployed":
                continue

            logger.debug("Checking VM: %s" % vm.name)
            # Choose host with the highest residual capacity
            h = self.pick_highest_residual_capacity_host(hs,vm.memory)

            # Migrate VM if needed
            if h != vm.host:
                logger.debug("Migrate VM")
                try:
                    stats = vm.migrate(h)
                    # Save VM to update host information
                    vm.save()
                except BaseModel.ModelException as e:
                    raise self.OptimizationException('Unable to migrate VM ' + str(vm) + ': ' + str(e))
            else:
                logger.debug("Do not migrate VM")

        return True
```

© Computer Networks Group - UFRGS

Source: by author (2015).

the platform. On the other hand, this decision limits the usability of the API to a specific programming language. To work that around, bindings to other popular programming languages can be added in future versions of the platform.

4.4 Implemented Technology Specific Drivers

To implement the *Drivers* section of the conceptual architecture, a set of pieces of code that rely on specific libraries and systems has been developed in order to "translate" high-level API calls into actual management actions on the underlying infrastructure. A lot of effort has been put to wrap technology specific parameters inside the driver implementation, abstracting this complexity from the *Unified API*. This is important to allow the *Administrator* to focus on developing his/her deployment and optimization programs and metrics only, rather than memorizing several configuration parameters. Ideally, configuration files should be used to tweak these drivers according to the setup of each environment.

For the *Networking* abstraction of virtual link, a driver for SDN based on the OpenFlow technology has been implemented. OpenFlow is suitable for the task, since the whole network can be controlled from a logically centralized element, called controller. Thus, to establish virtual links, the Aurora platform communicates with a Floodlight controller (Floodlight, 2014) pushing the appropriate OpenFlow forwarding rules. Therefore, in the current implementation scenario, the managed infrastructure is assumed to be a fully connected network with OpenFlow enabled switches. Moreover, virtual links are established considering layer 2 connectivity to have minimum interference on the choices for guest operating systems and communication protocols. In other words, the Aurora platform does not impose that deployed applications run necessarily on TCP/IP, like most platforms do.

For the virtual router abstraction, the technology used is Open vSwitch, with the restriction that all virtual machines connected to a virtual router need to be hosted at the same physical node. The deployed virtual router may have OpenFlow capabilities and the controller may be placed either inside or outside the Aurora platform (JESUS; WICKBOLDT; GRANVILLE, 2013). Another form of implementing virtual routers is by deploying virtual machines running router images (SANTOS et al., 2015); however, the provisioning time could be significantly higher. So far, no layer 3 routing is implemented within the virtual router abstraction.

For all *Computing* and *Storage* abstractions (except for guest images) the current development is based on Libvirt (Python binding) (Libvirt, 2012). Libvirt is a very popular library for virtualization management and is also used by other platforms, such as OpenStack. This library implements communication with several different hypervisors and all the abstractions needed by these two components of the proposed *Unified API* are sufficiently provided by Libvirt. The guest image abstraction is implemented and deployed based on simple file copy operations with *rsync*. Monitoring drivers were implemented based on FlexACMS and Nagios plug-ins, as explained in the beginning of this chapter.

4.5 Summary

This chapter presented a proof of concept cloud management platform called Aurora Cloud Manager. This platform has been developed following the conceptual aspects presented in Chapter 3 and combines several tools and libraries in order to manage virtual resources. Initially, an overview of the interactions of the proposed platform with the systems that compose the *Monitoring Infrastructure* was presented. Afterwards, the use of VXDL as a description language to trace the *Initial Specification* of a *Cloud Slice* was detailed. Then, the mapping of the conceptual architecture to the MVT design model and the implementation of core components to execute *Resource Management Programs & Metrics* was discussed. Finally, implementation of the proposed API abstractions and operations, as well as the technology specific drivers needed to materialize these abstractions were presented.

In order to obtain the results presented in the remainder of this thesis, the platform described in this chapter was employed to manage an emulated infrastructure created to reproduce slightly complex data center topologies. Further details on the setup of the experiments and the case study are presented in Chapter 5, focusing both on deployment and optimization of virtual infrastructures.

5 CASE STUDIES & EXPERIMENTS

To show the feasibility of the new concept of cloud management platform introduced in this thesis, this chapter presents two case studies conducted to cover different aspects of resource management in IaaS clouds. The first case study, presented in Section 5.1, intends to show the deployment of a network-intensive type of application based on concepts of Information-Centric Networking (ICN) and relying on the NetInf architecture (KUTSCHER et al., 2011). In addition, Section 5.2 presents the second case study focusing on optimization of resources according to different objectives and discusses the results obtained with three resource optimization algorithms implemented as programs within the proposed platform.

5.1 NetInf: Case Study & Experiments

For evaluation purposes this section presents a case study considering the deployment of an application based on the NetInf[1] architecture using the cloud platform proposed in this thesis. NetInf is a novel ICN architecture initially conceived in the FP7 project called *4WARD: Architecture and Design for the Future Internet* (4WARD, 2008) and then further developed during another FP7 project called *Scalable and Adaptive Internet Solutions (SAIL)* (SAIL, 2010). The NetInf architecture envisions routing information through a network based on the content itself, instead of relying on regular network addressing schemes. One of NetInf's assumptions is that, in the future, network devices will ship with native support for the NetInf architecture. This assumption is obviously not yet satisfied by current networks, so NetInf-based applications will rely on cloud platforms, such as the one proposed in this thesis, that can meet their unusual communication needs for large scale deployment.

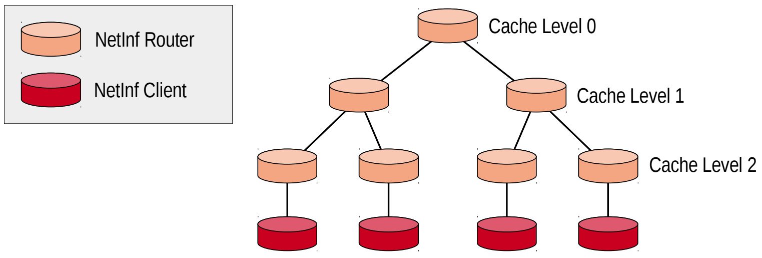
5.1.1 Scenario Description

The scenario considered in this case study for using the NetInf-based application is the following. A worldwide video producer (*e.g.*, TV channel) produces all kinds of video (*e.g.*, news, TV series, cartoons) in a central office. The producer company wants to

[1]<http://www.netinf.org/>

stream its videos on demand over the Internet to people that may come from all countries at any time. NetInf routers are able to receive video stream requests from clients, routing them through the network until these requests reach a router that has the demanded object. Once the object is found in the network, it is routed back to the client and cached by all NetInf routers in the path. With this caching system upcoming requests placed by clients for the same object may hit a cache along the path, avoiding the transmission again from the source and improving network utilization. The topology for this kind of content distribution network (interconnections of caches) should be preferably a multi-level tree, as shown in Figure 5.1. NetInf will rely on the topology to route content, as opposed to using router IP addresses.

Figure 5.1: NetInf application deployed topology



Source: by author (2015).

In this case study, a set of virtual machines running NetInf routers as well as NetInf clients are deployed over the cloud infrastructure. In a real world application, clients would be expected to come from outside the cloud, but to simplify the scenario clients are deployed within the same *Cloud Slice* as routers in this case study. All 7 NetInf routers and 4 NetInf clients are deployed from a cloud optimized version of Ubuntu Server 14.04 guest operating system images sizing near 213 MB, expansible up to 2 GB. Once deployed, the NetInf router in the root of the tree (*Cache Level 0*) already has published all the objects available for download. Clients will start placing requests for random objects nearly every minute as soon as they start up.

5.1.2 Deployment Program Implementation

For the deployment of the NetInf-based application, a simple program has been designed for the specific situation. The general concept of the implemented deployment program encompasses the calculation of resource allocation by analyzing pairs of virtual machines connected with a virtual link, instead of allocating one virtual resource at a time. Placing connected virtual machines close together in a data center is generally desirable to reduce communication latency and also to save network resources. The deployment program implemented is divided into three main phases: (*i*) resource discovery, (*ii*) reasoning, and (*iii*) resource allocation. The main structure of the program is presented in Algorithm 5.1, while most of the reasoning phase is presented in Algorithm 5.2.

Algorithm 5.1 Distance-aware deployment program

```
Require: $C \leftarrow$ cloud slice to be deployed
 1: $H \leftarrow$ set of physical hosts of the infrastructure
 2: $V \leftarrow$ set of virtual machines belonging to $C$
 3: $L \leftarrow$ set of virtual links belonging to $C$
 4: $host\_hops \leftarrow [\,]$
 5: for each $host \in H$ do
 6:     $host\_hops[host] \leftarrow distances(host, H)$
 7: end for
 8: $deployment\_pairs \leftarrow [\,]$
 9: for each $link \in L$ do
10:     $pair \leftarrow [null, null]$
11:     $pair[0] \leftarrow link.start.attached\_device$
12:     $pair[1] \leftarrow link.end.attached\_device$
13:     $deployment\_pairs.append(pair)$
14: end for
15: for each $p \in deployment\_pairs$ do
16:     if $p[0].host <> null$ and $p[1].host <> null$ then
17:         $continue$
18:     end if
19:     if $p[0].host == null$ and $p[1].host == null$ then
20:         $calculate\_allocation(null, p[0], H, host\_hops)$
21:         $calculate\_allocation(p[0], p[1], H, host\_hops)$
22:     else
23:         if $p[1].host == null$ then
24:             $calculate\_allocation(p[0], p[1], H, host\_hops)$
25:         else
26:             $calculate\_allocation(p[1], p[0], H, host\_hops)$
27:         end if
28:     end if
29: end for
30: for each $vm \in V$ do
31:     $vm.deploy()$
32:     $vm.start()$
33: end for
34: for each $link \in L$ do
35:     $link.establish()$
36: end for
```

As mentioned in Section 3.3, deployment programs are intended to calculate and allocate the resources for one particular *Cloud Slice*. Therefore, the input for this program is the specific *Cloud Slice* that is set to be deployed (requirement of Algorithm 5.1). In the first phase, the deployment program makes use of the Unified API to gather information about the physical resources available at the infrastructure (*e.g.*, set physical hosts) and also gathers information about the *Cloud Slice* to be deployed (Algorithm 5.1, lines 1 to 3). Moreover, in this phase, the deployment program needs to calculate the distances between all hosts of the infrastructure (Algorithm 5.1, lines 4 to 7). This procedure is abbreviated in the algorithm – enclosed in function $distances()$ – , but in summary it is performed with the aid of topology discovery methods available at the Physical abstraction of the Unified API. It is important to note that calculating routes between nodes on any type of graph can be a costly operation, thus it might not scale for large infrastructures. For example, with the best known implementation of Dijkstra's algorithm, the time to find the best path between nodes of a graph grows in the order of $O(|E| + |V|log|V|)$, where $|E|$ and $|V|$ are the number of edges and nodes in the graph respectively. Finally, the resource discovery phase ends by preparing a list of pairs of virtual machines based on the virtual links that belong to the *Cloud Slice* being deployed (Algorithm 5.1, lines 8 to 14).

In the second phase, the deployment program computes where to place virtual machines based on very specific heuristics (Algorithm 5.1, lines 15 to 29). As mentioned, instead of trying to allocate one virtual machine at a time, this program allocates pairs of virtual machines based on $deployment\_pairs$ list generated in the previous phase. The general idea is to select one virtual machine out of the pair to be placed first and work as a *pivot* for the operation, while the second virtual machine (called *free*) needs to be placed close to the pivot. When both virtual machines have not been allocated yet (Algorithm 5.1, lines 19 to 21) it does not matter which virtual machine of the pair is the pivot, so the first one will be allocated anywhere in the data center and will be the chosen pivot of the pair. When one virtual machine of the pair is already placed, it will be the pivot while the other is considered free (Algorithm 5.1, lines 22 to 28).

For example, consider a simple *Cloud Slice* with 3 virtual machines, VM1, VM2, and VM3, connected by two virtual links VM1↔VM2 and VM2↔VM3. This virtual infrastructure would be deployed in two pairs (VM1, VM2) and (VM2, VM3). When the first pair is being processed, both virtual machines are considered free to be placed anywhere in the data center, so the program will choose a host with low resource utilization (previous memory and CPU allocations) to deploy VM1 of the pair. Since there is a virtual link between VM1 and VM2, one may assume that these virtual machines will communicate often, therefore VM1 and VM2 should be placed close together in the net-

Algorithm 5.2 Calculate Allocation procedure

```
Require: $vm\_pivot \leftarrow$ virtual machine to be the pivot of the operation
Require: $vm\_free \leftarrow$ virtual machine to be allocated close to pivot
Require: $H \leftarrow$ set of physical hosts of the infrastructure
Require: $host\_hops \leftarrow$ calculated distances among hosts of the infrastructure
 1: $candidate\_hosts \leftarrow [\,]$
 2: for each $h \in H$ do
 3:     $mem\_total \leftarrow h.get\_memory()$
 4:     $mem\_allocation \leftarrow h.get\_memory\_allocation()$
 5:     $mem\_free \leftarrow mem\_total - mem\_allocation$
 6:     $mem\_percentage \leftarrow mem\_free/mem\_total$
 7:     $cpu\_total \leftarrow h.get\_cpu()$
 8:     $cpu\_allocation \leftarrow h.get\_cpu\_allocation()$
 9:     $cpu\_free \leftarrow cpu\_total - cpu\_allocation$
10:     $cpu\_percentage \leftarrow cpu\_free/cpu\_total$
11:     if $mem\_free < vm\_free.memory$ then
12:         $continue$
13:     end if
14:     if $cpu\_free < vm\_free.cpu$ then
15:         $continue$
16:     end if
17:     $hop\_count \leftarrow 1$
18:     if $vm\_pivot <> null$ then
19:         $hop\_count \leftarrow host\_hops[vm\_pivot.host][vm\_free.host]$
20:     end if
21:     $coefficient \leftarrow (cpu\_percentage * mem\_percentage)/hop\_count$
22:     $candidate \leftarrow [h, coefficient]$
23:     $candidate\_hosts.append(candidate)$
24: end for
25: $vm\_free.host \leftarrow get\_best\_coefficient(candidate\_hosts)$
```

work. Thus, for the placement of VM2, the program will prioritize physical hosts that have low resource utilization and are not too far away from the first one chose for VM1. For the next pair, VM2 would be already placed, so the program would only have to find a host for VM3 and the *Cloud Slice* would be ready for deployment.

The actual calculations to decide where each virtual machine will be placed are performed by a specific procedure presented in detail in Algorithm 5.2. This procedure receives two virtual machine objects (pivot and free), the set of physical hosts of the infrastructure, and the already calculated distances among hosts. Overall, for each host of the managed infrastructure (Algorithm 5.2 line 2) that has the resources required to deploy the free virtual machine (Algorithm 5.2 lines 11 to 16), a coefficient is calculated (Algorithm 5.2 line 21) representing how good of a candidate this host is to place this virtual machine. This coefficient takes into consideration the percentage of free resources

(CPU and memory) at the candidate host and also – in the case where a pivot has been specified (Algorithm 5.2 lines 18 to 20) – the distance to the host where the pivot virtual machine is. All candidate hosts along with their calculated coefficients are added to a list (Algorithm 5.2 lines 22 to 23), which will be used to select the best host at the end of this phase (Algorithm 5.2 line 25).

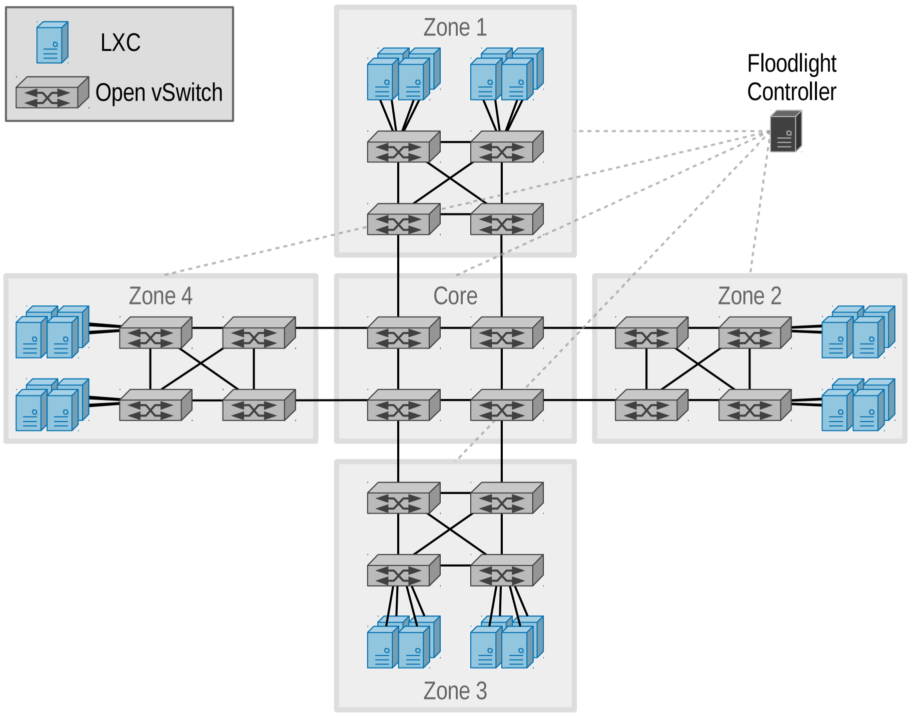
In the last phase, the program deploys all virtual machines on the hosts assigned in the previous phase. This deployment encompasses copying guest operating system images, defining virtual machines in hypervisors, and starting them (Algorithm 5.1 lines 30 to 33); these are operations all carried out by the Unified API. Finally, with all the virtual machines in place, the program establishes all the necessary virtual links (Algorithm 5.1 lines 34 to 36). It is important to emphasize that it is not the objective of this case study to claim that this is an optimal program for resource allocation. Nevertheless, it has shown to be efficient enough for the deployment of the specific application at hand and, more importantly, it was easily implemented in a few hours with the proposed platform.

In addition to this distance-aware program just described, another simple random deployment program was implemented to serve as comparison basis. This second program performs the same phases as the former, however the resource discovery phase does not need to calculate distances between hosts because they are not necessary. Moreover, the reasoning phase is also simplified in such a way that for every individual virtual machine of a *Cloud Slice* the program simply picks a random host with the resources required to store it. The actual algorithm for the random deployment program is trivial, thus it is not further discussed in this thesis.

5.1.3 Emulated Infrastructure & Virtual Topologies

Figure 5.2 shows the emulated topology created for experimentation with 20 Open vSwitches running the OpenFlow protocol and being coordinated by a Floodlight controller. Also, every one of the 32 hosts of the infrastructure is emulated by a Linux virtualization container based on LXC. The topology chosen for experimentation is organized into 4 zones and a core network. Each zone defined represents a “small data center” with a highly redundant local mesh network and two independent connections to different switches of the core network. The ring-structured core network contains 4 switches allowing communication to flow between zones and represents a backbone for interconnecting the “small data centers”. This emulated topology is interesting for this specific experimentation because of the amount of independent paths that can be used to establish virtual links, which gives the deployment program plenty of opportunities to optimize resource allocation.

Figure 5.2: Emulated physical topology based on LXC and Open vSwitches

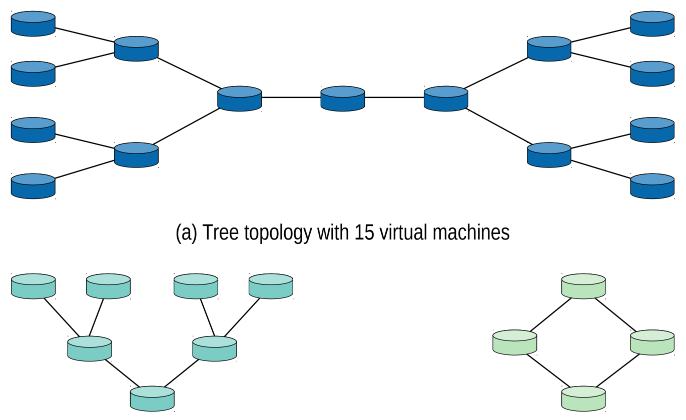


Source: by author (2015).

This setup is started as a Mininet (Mininet, 2013; LANTZ; HELLER; MCKEOWN, 2010) script on one Dell PowerEdge R815 server with 4 AMD Opteron Processor 6276 Eight-Core processors and 64GB of RAM memory. The Aurora platform runs also on this physical machine and all LXC hosts run one independent instance of Libvirt to receive commands from the platform. The Floodlight controller is provided with a dedicated connection to every switch of the infrastructure, although, for the sake of presentation, Figure 5.2 only displays one connection per zone. A separate management network was also set up (also not shown in Figure 5.2) with a direct connection to every host so that Libvirt traffic to manage virtual machines can flow without affecting production traffic.

To obtain the results presented in the next subsection, statistics have been gathered from the deployment of the NetInf-based application, which encompasses creating 11 virtual machines and 10 virtual links (topology of Figure 5.1). For comparison purposes, three additional topologies using the same guest operating system images of the NetInf routers have been deployed, as shown in Figure 5.3. A ring topology with 4 virtual machines and 4 virtual links (Figure 5.3-(c)) and two other tree topologies, like the one in Figure 5.1, but without the NetInf clients. One tree with 7 virtual machines and 6 links (Figure 5.3-(b)) and another with 15 virtual machines and 14 links (Figure 5.3-(a)). These different virtual topology sizes have been deployed to allow a first glimpse on the scalability of the deployment program and the platform implemented. Although these experiments target a specific application, creating virtual topologies can be beneficial for many kinds of applications in the context of content distribution systems or information-centric networking, for example (AHLGREN et al., 2012).

Figure 5.3: Virtual infrastructure topologies deployed



(a) Tree topology with 15 virtual machines

(b) Tree topology with 7 virtual machines

(c) Ring topology with 4 virtual machines

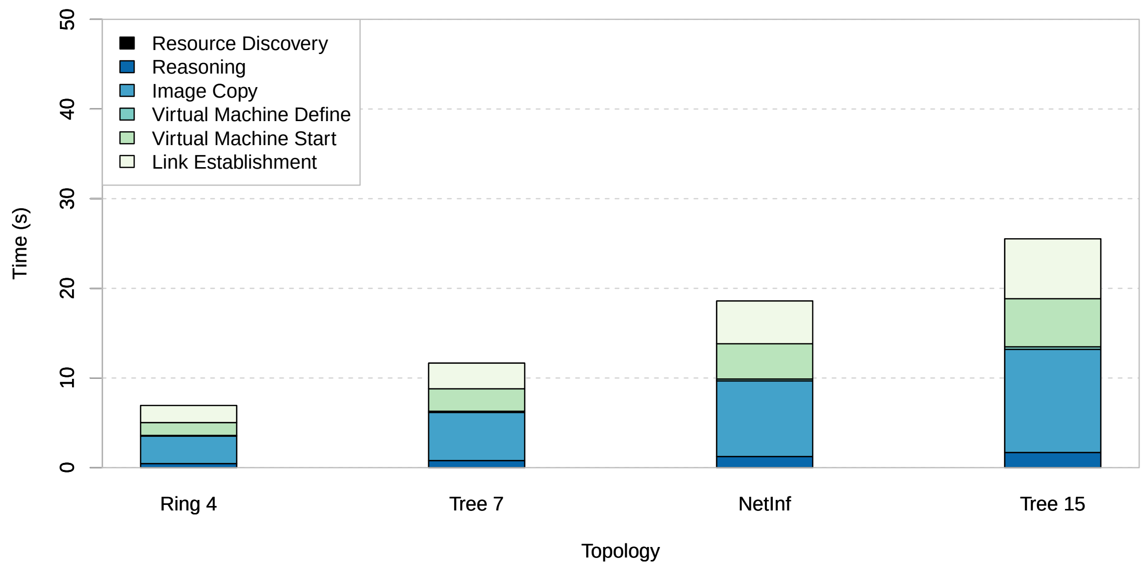
Source: by author (2015).

5.1.4 Deployment Statistics

For this experiment, both the distance-aware and random deployment programs described earlier have been monitored in each phase of their execution. A load of 5 *Cloud Slices* of each 4 different topologies described in Figures 5.1 and 5.3 (referred henceforth simply as NetInf, Ring 4, Tree 7, and Tree 15) have been deployed using each program. The specification of virtual machine memory requirements were also varied to generate a more heterogeneous memory allocation distribution. For Ring 4 and Tree 7 topologies each virtual machine memory was set to 512 MB and 256 MB respectively, whereas for the NetInf and Tree 15 topologies a smaller memory size of 128 MB was set. The total of 20 *Cloud Slices* deployed with each program sum up to a rough total of 35 GB of memory allocated, which accounts for around 55% of the emulated data center total memory capacity. In terms of guest images, there were 185 deployments with a total of roughly 36 GB of data transfered, while 170 virtual links with varied lengths were established by each program.

The results in terms of average deployment time achieved by each program for the scenario just described are presented in Figures 5.4 and 5.5. The stacked bars in these figures represent the average time spent in each phase (*i.e.*, resource discovery, reasoning, and resource allocation) of the deployment programs for each virtual topology. The resource allocation phase is further narrowed down into specific deployment events (*i.e.*, image copy, virtual machine defining and starting, and link establishment) to provide a deeper understanding of the share of time that each of these events plays into the overall resource allocation phase. For the sake of presentation, error bars are not shown in these stacked bar charts. However, each individual deployment phase and event was performed enough times so that the estimated error values were kept below 10% of each absolute measure, with a confidence interval of 95%.

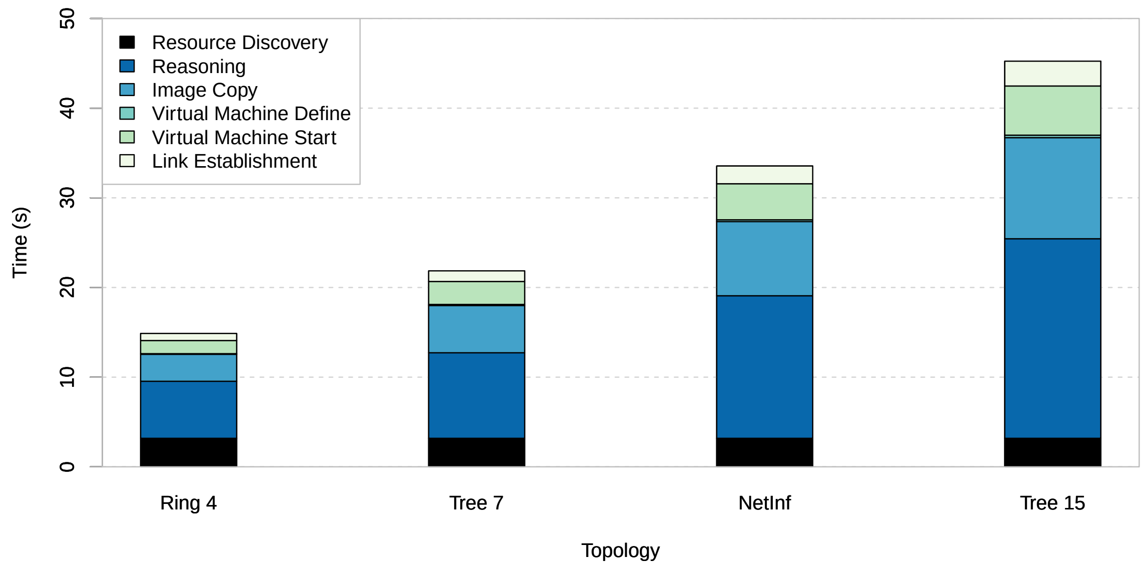
Figure 5.4: Average deployment time for the random program



Source: by author (2015).

In Figure 5.4, the results obtained with the random deployment program are presented. It is possible to see a nearly linear increase in the time to deploy a *Cloud Slice* as the number of virtual resources grows. The total deployment time of a *Cloud Slice* varies from around 7 seconds for the Ring 4 topology (4 virtual machines and 4 virtual links) up to 25 seconds for the Tree 15 (15 virtual machines and 14 virtual links). It is remarkable that most of the program execution time is actually spent on copying images and starting virtual machines. The resource discovery phase in this program is negligible since there is no much information that needs to be acquired before deploying virtual resources. The reasoning time is also insignificant for this program, since it only needs to find a random host with enough resource capacity to instantiate each virtual machine. Finally, the link establishment time also accounts for a large portion of the total time spent in deploying a virtual infrastructure. This time encompasses the communication between the Aurora platform and the Floodlight controller to discover the path for the virtual link, plus the time to install OpenFlow rules in each switch of the path.

Figure 5.5: Average deployment time for the distance-aware program



Source: by author (2015).

Figure 5.5 shows the results obtained while monitoring the deployment of the same load of 20 *Cloud Slices* using the distance-aware program. It is also possible to visualize that the time to deploy each virtual infrastructure grows near linearly with the size of it (*i.e.*, number of virtual resources). Virtual machine related operations (*e.g.*, image copy, starting, and defining) present the same results obtained with the random program, since these times are influenced mostly by the number of operations performed. However, the reasoning time in the distance-aware program increases more than tenfold in comparison with the random deployment. This behavior is expected, since this program needs to gather much more information to decide how to allocate every pair of virtual machines connected with virtual links (*i.e.*, a resource coefficient including memory, CPU, and distance requirements). Moreover, the resource discovery time now accounts for a more significant part of the total deployment time of any virtual infrastructure. During the resource discovery phase the algorithm calculates the distances between every host of the infrastructure and, since there is a fixed number of hosts, the amount of effort to perform this phase is always the same. Finally, the link establishment time decreases significantly with the distance-aware program. This behavior is due to the length of routes chosen by the algorithm which are much shorter in average than they were in the random deployment program, thus a smaller number of OpenFlow rules will have to be installed to deploy each virtual link. Just to provide an estimate, the average established virtual link length with the random deployment program is in the order of 6.7 hops, whereas with the distance-aware program this length is decreased down to 1.8.

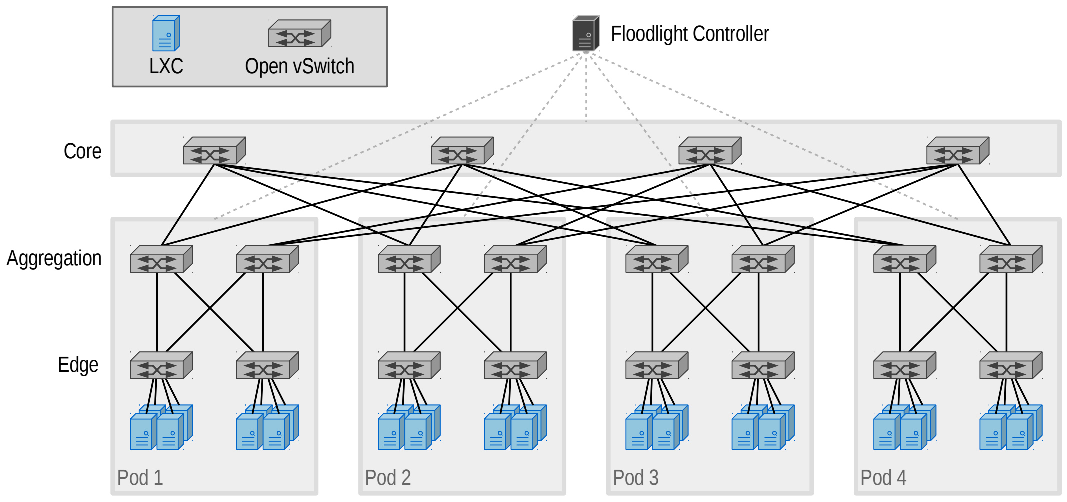
5.2 Optimization: Case Study & Experiments

The experiments presented in the previous section were targeted to demonstrate the use of deployment programs to achieve specific objectives while establishing a virtual infrastructure for the first time. On the other hand, the main purpose of the experiments described in this section is to show how an *Administrator* can benefit from the approach to cloud resource management introduced in this thesis by easily writing optimization programs. This second class of programs can be employed to reorganize already established resource allocations in order to achieve various distinct objectives.

5.2.1 Emulated Infrastructure & Virtual Topologies

The experiments presented in this section are also based on a topology emulated with Mininet. The topology created for this set of experiments is depicted in Figure 5.6 containing 20 Open vSwitches running OpenFlow protocol and being coordinated by a Floodlight controller. Also, this topology contains 32 hosts forming an emulated infrastructure based on Linux virtualization containers with LXC. The emulated topology chosen for these experiments is slightly different from the one employed in the previous section. This topology is a classical Fat-tree as described by Al-Fares *et al.* (2008). Fat-trees are commonly employed in data centers to provide increased throughput with link aggregation and high levels of redundancy. These topologies can be structured according to one input parameter – number of $pods$ –, which in this case study was set to 4 $pods$. Each pod includes two layers of switches, edge and aggregation, with $pods/2$ switches per layer. In addition, $(pods/2)^2$ core switches are included in the topology to interconnect $pods$. As for connections, every edge switch connects to $pods/2$ aggregation switches, while aggregation switches also connect to $pods/2$ core switches. According to the definition of a Fat-tree topology, $pods/2$ hosts are connected to every edge switch. However, this number of hosts was doubled in these experiments only to increase the proportion of computing resources in relation to the amount of network resources and to match the same infrastructure size used in the previous case study.

Figure 5.6: Emulated physical topology based on LXC and Open vSwitches



Source: by author (2015).

The hardware description of the server where experiments are conducted is the same presented in Section 5.1. The Aurora platform is also installed on this machine and communicates with the LXC hosts through a separate management network (not shown in Figure 5.6). The monitoring infrastructure with FlexACMS and Nagios runs on another physical machine and the communication between the systems goes over HTTPS connection on an ordinary LAN. Finally, the virtual topologies employed in this experiment where the same displayed in Figures 5.1 and 5.3 in the previous section (*i.e.*, Ring 4, Tree 7, NetInf, Tree 15).

5.2.2 Optimization Programs Implemented

Three optimization programs have been defined for this case study: *OptimizeBalance*, *OptimizeGroup*, and *OptimizeHops*. First, it is important to point out that the programs introduced in this section are not completely novel. Instead, they are inspired in algorithms found in the literature. *OptimizeBalance*, for example, aims to spread virtual machines of a *Cloud Slice* over the managed infrastructure, which is the same load balancing objective of the algorithm proposed by Chowdhury *et al.* (2012) for virtual network embedding. *OptimizeGroup*, in turn, uses migration to concentrate virtual machines on a small set of physical hosts, which was envisioned by the VROOM architecture (WANG et al., 2008), in the context of virtual routers. Similar to the work proposed by Meng *et al.*

(2010), the *OptimizeHops* program aims to reduce the communication cost by reducing the distance between virtual machines.

Algorithm 5.3 describes the behavior of the *OptimizeBalance* program. For each virtual machine deployed over managed infrastructure, the program selects the physical host with the highest residual capacity in terms of memory (line 4) and migrates the virtual machine to it (line 6). By selecting the host with the highest residual capacity, *OptimizeBalance* maps a virtual machine to the least loaded host. In an infrastructure of homogeneous physical hosts (*i.e.*, same total memory capacity), this program ends up creating a nearly even memory allocation across the whole infrastructure. To minimize the number of migrations performed during the optimization procedure, the $get\_highest\_residual\_capacity\_host()$ function makes sure never to select a $candidate$ host that will have the its residual capacity reduced below $vm.host$'s residual capacity after the migration.

Algorithm 5.3 OptimizeBalance program

```
1: $H \leftarrow$ set of physical hosts
2: $V \leftarrow$ set of virtual machines
3: for each $vm \in V$ do
4:     $candidate \leftarrow get\_highest\_residual\_capacity\_host(H, vm)$
5:     if $candidate$ != $vm.host$ then
6:         $migrate(candidate, vm)$
7:     end if
8: end for
```

The second program, called *OptimizeGroup* (Algorithm 5.4), goes into the opposite direction by mapping virtual machines into the smallest possible subset of physical hosts. This is done by migrating virtual machines to the set of hosts with the lowest residual capacity in terms of memory. If physical hosts have the same initial capacity, this program chooses the same host to receive a number virtual machines until the selected host runs out of capacity; in which case, a new host is selected. The *OptimizeGroup* program also never selects a $candidate$ host that will have the its residual capacity increased above $vm.host$'s residual capacity after the migration to minimize the number of migration operations. If associated with an energy management mechanism that can turn hosts on/off on demand, the *OptimizeGroup* program can be used for energy-saving purposes, since a smaller number of hosts will actually receive many virtual machines while others completely become idle.

The third program, called *OptimizeHops* (Algorithm 5.5), reduces the number of hops between linked virtual machines. This is achieved by analyzing pairs of virtual

Algorithm 5.4 OptimizeGroup program

```
1: $H \leftarrow$ set of physical hosts
2: $V \leftarrow$ set of virtual machines
3: **for each** $vm \in V$ **do**
4:     $candidate \leftarrow get\_lowest\_residual\_capacity\_host(H, vm)$
5:     **if** $candidate$ != $vm.host$ **then**
6:         $migrate(candidate, vm)$
7:     **end if**
8: **end for**
```

machines connected by a virtual link (line 4), fixing one of them as a *pivot*, and moving the other (referred to as *free*) onto a host located at a shorter distance in the physical network (similar method used in Algorithm 5.1). To decide which virtual machine is the *pivot* and which one is *free* to move, the total number of virtual links connecting these virtual machines is considered (lines 5 to 10). This decision is based on the assumption that migrating a virtual machine that has a smaller number of connections seems to cause less disruption to the service running on the *Cloud Slice*. Afterward, the program will find a list of hosts with capacity to store *free* (line 12) and record the sum of distances to all neighbors of *free* at its original location (line 13). Then, for each candidate host, the program will find the best host to store *free* considering the sum of distances to its neighbors at the new candidate location (lines 14 to 21). The idea behind this operation is trying not to migrate *free* to a location closer to *pivot* and farther from most of its other neighbors. Finally, if the best distance to all neighbors of *free* found in the previous step improves over its current location (line 22), then *free* will be migrated to *best_candidate* (line 23).

The main benefit of optimization programs to the cloud *Administrator* is flexibility and adaptability to different situations. The first two implemented programs are mainly focused in optimizing the infrastructure for load balancing and energy consumption, which are conflicting objectives and choosing one of them can depend on many factors, such as budget constraints and internal administrative policies. The third program can be considered an optimization of both infrastructure and application dimensions. From the infrastructure perspective, the benefit of *OptimizeHops* is minimizing the overall traffic in network. For the application, reducing the physical distance of virtual links will most likely also reduce delay in communication between connected virtual machines. Besides adding flexibility, programs are quite simple to write using the proposed Unified API. Just to provide a rough idea, *OptimizeBalance* and *OptimizeGroup* programs

Algorithm 5.5 OptimizeHops program

```
1: $H \leftarrow$ set of physical hosts
2: $L \leftarrow$ set of virtual links
3: $P \leftarrow$ set of pairs $(vm\_from, vm\_to) \in L$
4: **for each** $p \in P$ **do**
5:     **if** $link\_count(p.vm\_from) > link\_count(p.vm\_to)$ **then**
6:         $pivot \leftarrow p.vm\_from$
7:         $free \leftarrow p.vm\_to$
8:     **else**
9:         $pivot \leftarrow p.vm\_to$
10:        $free \leftarrow p.vm\_from$
11:    **end if**
12:    $candidates \leftarrow get\_list\_of\_hosts\_with\_capacity(H, free)$
13:    $original\_distance \leftarrow distance\_neighbors(free, free.host)$
14:    $best\_distance \leftarrow original\_distance$
15:    **for each** $candidate \in candidates$ **do**
16:        $new\_distance \leftarrow distance\_neighbors(free, candidate)$
17:        **if** $new\_distance < best\_distance$ **then**
18:            $best\_distance \leftarrow new\_distance$
19:            $best\_candidate \leftarrow candidate$
20:        **end if**
21:    **end for**
22:    **if** $best\_distance < original\_distance$ **then**
23:        $migrate(best\_candidate, free)$
24:    **end if**
25: **end for**
```

are less than 40 lines of code long. The *OptimizeHops* program, which includes all pair processing and distance calculations, is still only 125 lines long.

5.2.3 Events & Optimization Scenario

Next, the scenarios designed to illustrate the benefits of optimization programs are presented. The optimization programs used in this case study are activated by a set of triggering events. Whenever a triggering event occurs, the active optimization program is executed. It is important to notice that only one optimization program is active in the Aurora platform at a time. The *Administrator* has to manually load another program when the optimization objective changes. The triggering events currently supported by the platform and used in this case study are listed in Table 5.1.

Initially, 20 *Cloud Slices* (5 of each virtual topology described in the previous section, *i.e.*, Ring 4, Tree 7, NetInf, Tree 15) are randomly deployed over the emulated

Table 5.1: Triggering events

Event name	Description
Slice creation	A new slice is created and can affect the already deployed ones
Slice termination	The life of the cloud slice has ended and it is no longer active
Slice modification	The slice is modified by adding or removing virtual machines
Capacity adjustment	The slice owner can modify the capacity (*e.g.*, increasing the memory of a virtual machine) of his/her slices
Administration	The platform *Administrator* can manually trigger the optimization program to perform administrative tasks (*e.g.*, maintenance)

Source: by author (2015).

infrastructure of Figure 5.6 with no optimization whatsoever. This reaches a load of 55% of the total data center capacity in terms of memory consumption, which leaves plenty of room for *OptimizeBalance* and *OptimizeGroup* programs to rearrange memory allocation. Regarding virtual link allocations, a Fat-tree topology allows for a vast number of possible routes anywhere in the network with distinct lengths (number of hops). Considering that each physical host adds an additional hop (hosts connect virtual machines to an extra local virtual switch) two virtual machines positioned in the same host are actually 1 hop away from each other. Virtual machines positioned in two different hosts, but in the same edge switch are 3 hops away; whereas a pair of virtual machines connected at different edge switches of the same pod are 5 hops away. Finally, two virtual machines positioned at hosts in different pods are 7 hops away, which is the farthest distance possible for this topology (considering only optimal routes). Statistically, it is very likely that the random deployment program will establish very long links, since the probability of two virtual machines connected by a virtual link falling by chance into the same host, for example, is very low. This will provide the *OptimizeHops* program with the many opportunities to optimize link allocations by migrating virtual machines to different locations of the data center.

After this initial state of random deployment, each of the optimization programs described in this case study was set active in the Aurora platform and was responsible for reorganizing the managed infrastructure according to their specific objectives, reacting to the following list of changes:

1. Four of the previously allocated *Cloud Slices* expire (one of each type) and release resources. In this case, other active slices can be reconfigured to benefit from the changes in the infrastructure (*Slice termination event*);

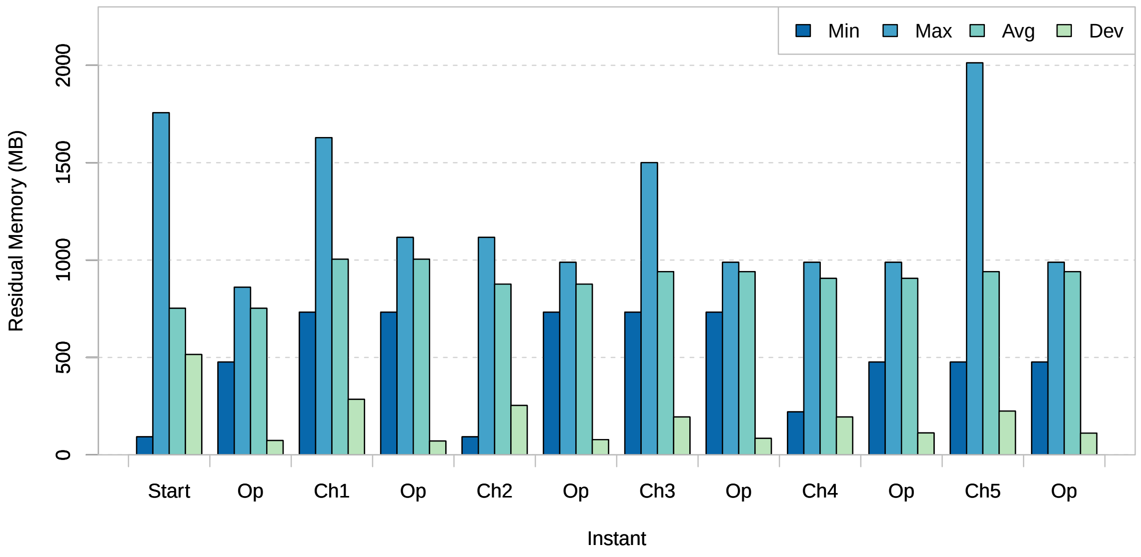
2. Four new virtual machines are instantiated and connected to each of two NetInf *Cloud Slices*. Here, optimization is required to prevent that the placement of the new virtual machines conflicts with the active policy (*Slice modification event*);
3. Two virtual machines are removed from each of two Ring 4 *Cloud Slices*, which makes room for reconfigurations similarly to item 1 (*Slice modification event*);
4. The *Administrator* performs maintenance in the infrastructure, for example, to install software updates in a host. Therefore, virtual machines need to be temporarily migrated to a different location (*Administration event*);
5. The *Administrator* replaces equipment to increase the capacity of the infrastructure, which may require optimizations to allow deployed *Cloud Slices* to take advantage of the new resources made available (*Administration event*).

Finally, it is worth mentioning that the events generated in this experiment are artificially triggered in a controlled environment, so that they happen always in the same order and affecting the same set of resources. Of course, in a real environment events will happen randomly and at arbitrary rates. For example, in a large setup a *Cloud Slice* could expire every minute or so, which could create a situation where resources are in constant state of optimization. Therefore, it is up to the *Administrator* to understand the consequences of optimization procedures (*e.g.*, the impact of migrating virtual machines too often) and configure the handling of events in the managed infrastructure accordingly.

5.2.4 Optimization Statistics

Changes and optimizations of the managed infrastructure have been monitored during experimentation for all three optimization programs previously explained. The charts presented in Figures 5.7 and 5.8 show statistics about the residual memory capacity of the infrastructure respectively for *OptimizeBalance* and *OptimizeGroup* programs. The residual memory capacity represents how much memory is not allocated for virtual machines, *i.e.*, memory that is available to be allocated by new virtual machines or migrations. The maximum memory of a host defined in this experiment is 2 GB; therefore, the *Max* value never surpasses this amount. The *Instant* axis represents the time span of the experiment, where changes and optimizations happen. The *Start* instant is the initial configuration of the infrastructure (20 *Cloud Slices* randomly deployed). Every *Op* instant represents the residual memory capacity right after an optimization occurred, while *Ch** are instants that succeed a change, according to the list of changes previously explained.

Figure 5.7: Residual memory capacity statistics using OptimizeBalance program

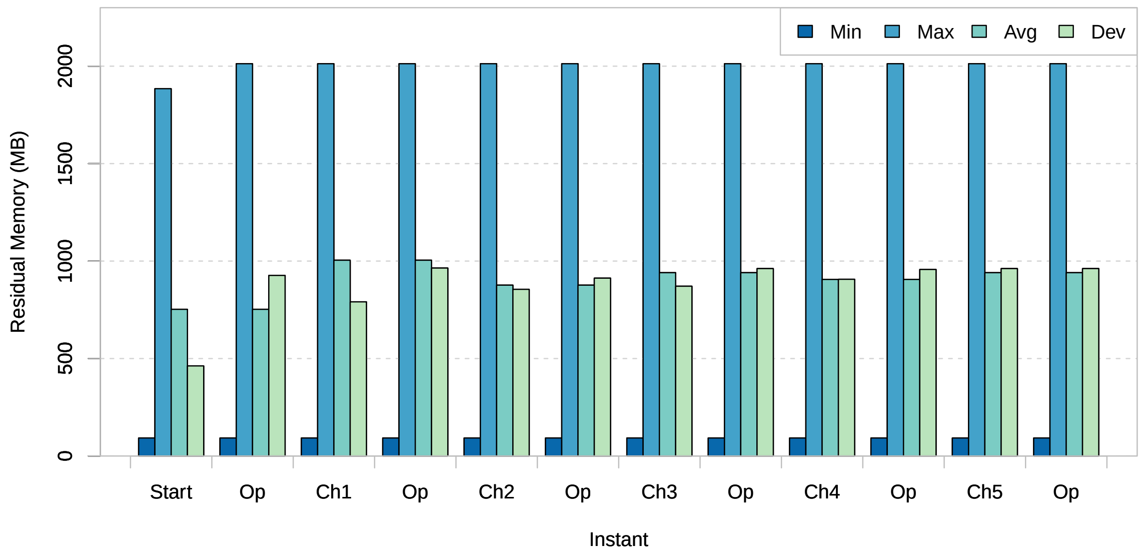


Source: by author (2015).

In Figure 5.7, it is possible to note how the *OptimizeBalance* program was able to influence the behavior of memory allocations to keep the infrastructure well balanced. In the *Start* instant of experimentation, there is quite some difference between *Max*, *Min*, and *Avg* residual memory capacity, which means that there are hosts heavily loaded while others have only one or two virtual machines instantiated. Also, *Dev* (standard deviation) is high at this instant, which indicates high variation between residual memory capacity of hosts. At the first *Op* instant, the *OptimizeBalance* program was already able to distribute the load among hosts, dropping *Dev* drastically and leading *Max*, *Min*, and *Avg* values much closer to one another. After every change, some degree of unbalancing is introduced in memory allocations, which is reflected in the disturbance of *Max*, *Min*, and *Avg* values and the increase of *Dev*. For example, in *Ch5* an *Administration event* happens in which the *Administrator* replaces a host of the managed infrastructure that was set to maintenance in *Ch4*, which increases *Max* residual memory allocation to its peek during the whole experiment. Afterward, the subsequent optimization conducts the infrastructure back to an optimized state.

Figure 5.8 shows the same measures as Figure 5.7, but employing the *Optimize-Group* program to manage resource allocation. Considering this program, for the infrastructure to be considered optimized, *Min* and *Max* values need to be as far apart as possible, while *Dev* needs to be high. This means that there are hosts with a lot of residual memory (high *Max*) and others with their capacities almost fulfilled (low *Min*). That is a

Figure 5.8: Residual memory capacity statistics using OptimizeGroup program



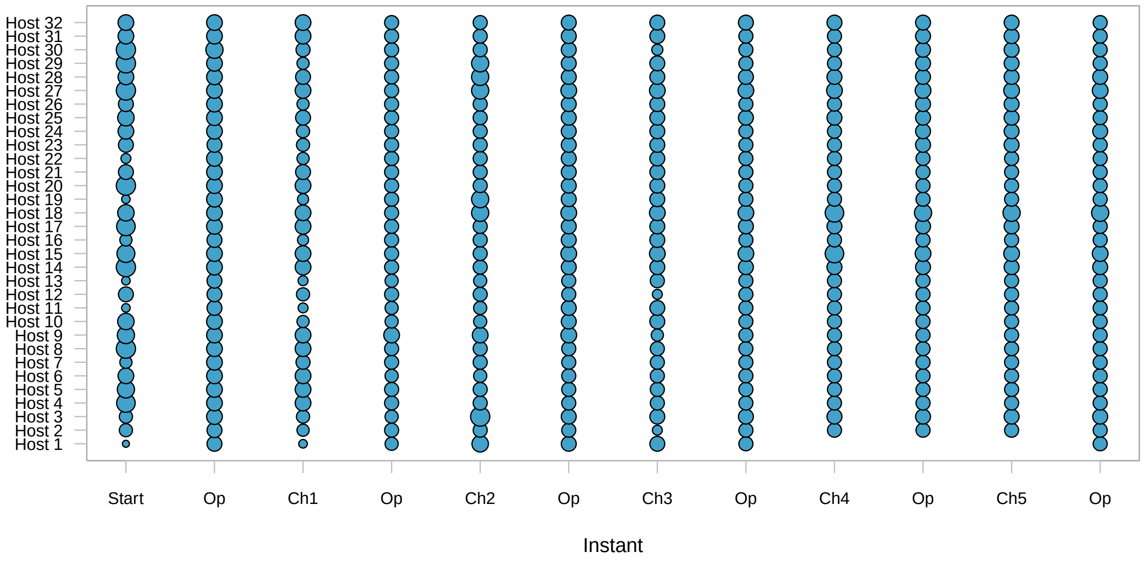
Source: by author (2015).

behavior that can be observed right after the first optimization. Having many hosts with their total capacity as residual/free memory means they can be considered idle and could be turned off to save energy. However, this program stumbles upon a fragmentation problem, for example, in this case study, it was never able to reduce the *Min* residual memory below 94 MB because none of the instantiated virtual machines was small enough to fit in this much memory. Another important fact to note in Figures 5.7 and 5.8 is that the *Avg* residual memory at any given instant of the results for both *OptimizeBalance* and *OptimizeGroup* programs is always exactly the same. This happens because the experiment was designed in such a way that the total amount of memory allocated is the same at every particular instant to provide a fair comparison standpoint between both programs.

The difference between *OptimizeBalance* and *OptimizeGroup* programs is more clearly visible in Figures 5.9 and 5.10. These charts show memory allocations per host throughout the experiment. The x axis presents *Host IDs* for every one of the 32 hosts of the managed infrastructure The y axis (*Instant*) – just like in Figures 5.7 and 5.8 – represents the time span of experiments. The area of each bubble in these charts represents the total memory allocated in every host for virtual machines at a given moment.

In Figure 5.9 it is easy to visualize that at the beginning (*Start* instant) memory allocations seem unbalanced. Optimizations (*Op* instants) tend to consistently bring all bubbles to similar sizes, whereas changes (*Ch** instants) reintroduce disturbance in allocations. It is interesting to note that in *Ch4* the *Administrator* removes *Host 1* from the infrastructure to perform some maintenance tasks, which becomes unavailable for allocation (bubble disappears). Because of this change, the *Administrator* manually migrated

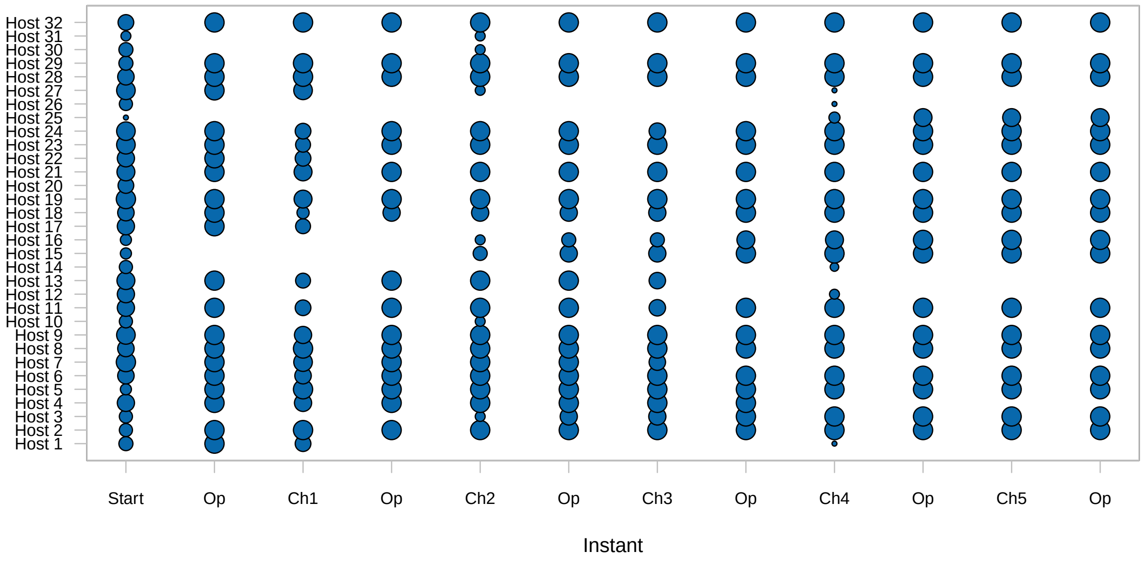
Figure 5.9: Memory allocations per host using OptimizeBalance program



Source: by author (2015).

virtual machines from *Host 1* to hosts *Host 15* and *Host 18*. The following optimization redistributed the load from *Host 15* and *Host 18* within other 6 hosts. After that, the next change (*Ch5*) reintroduces *Host 1* and, on the optimization next to this change, *Host 1* receives a load of virtual machines again.

Figure 5.10: Memory allocations per host using OptimizeGroup program

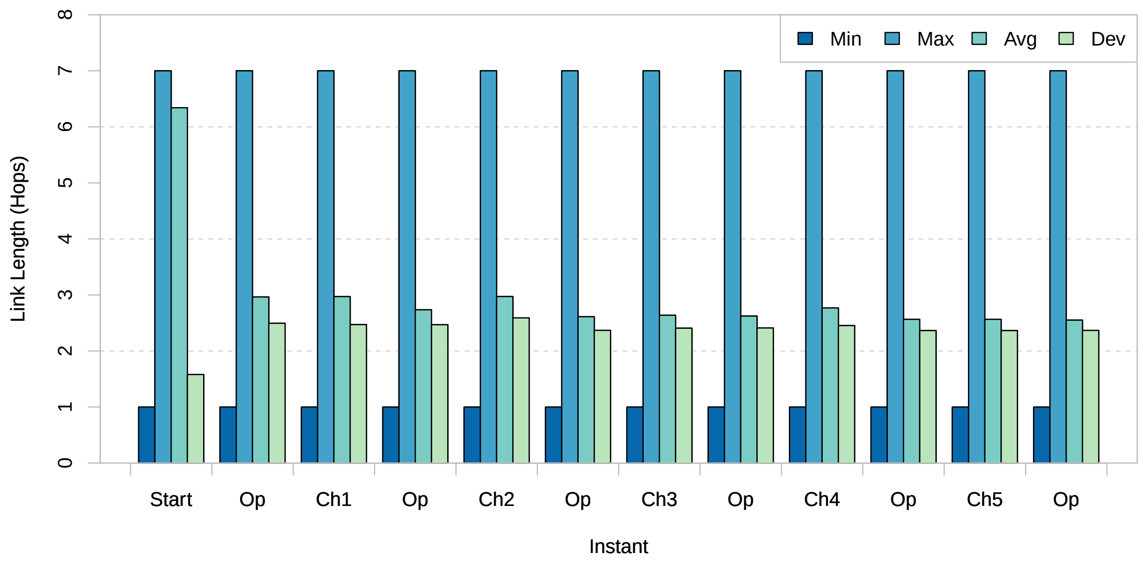


Source: by author (2015).

In Figure 5.10, it is possible to see that the behavior of memory allocations guided by the *OptimizeGroup* program is completely opposite to *OptimizeBalance*. After the first optimization, for example, all the load is distributed among only 21 hosts, making all others idle. This is an interesting result because, although 55% of the total data center memory capacity is allocated while 45% is idle, only around 35% of the managed infrastructure or 11 hosts have their full amount of residual memory and can be considered actually idle to be turned off. This phenomenon happens because of the memory fragmentation problem mentioned earlier. In the 21 hosts heavily loaded there is actually an aggregated total of nearly 2 GB of memory that cannot be allocated because of the size of virtual machines.

Figures 5.11, 5.12, and 5.13 show the results achieved with the *OptimizeHops* program. The bar chart of Figure 5.11 is very similar to the ones presented in Figures 5.7 and 5.8. However, instead of residual memory, the length of virtual links (*i.e.*, number of hops) connecting pairs of virtual machines is displayed on the *Link Length* axis. As explained earlier, in this experiment every switch is considered as one network hop, plus the extra virtual switch included inside each host. Therefore, hop counts for virtual links in this experiment will always be in the range of 1, 3, 5, or 7.

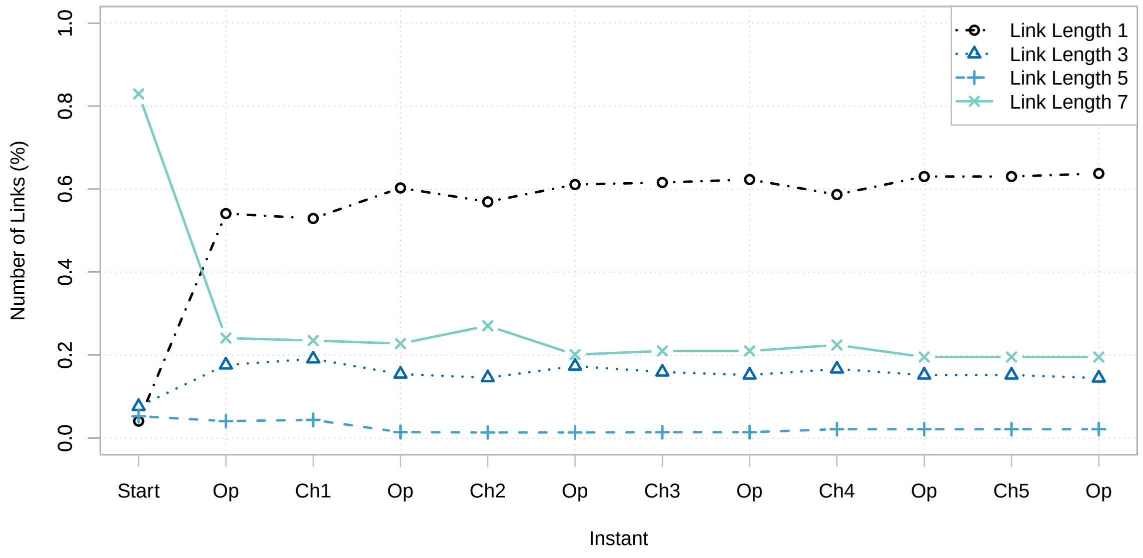
Figure 5.11: Virtual link length statistics using OptimizeHops program



Source: by author (2015).

One may note from Figure 5.11 that at the beginning of the experiment (*Start* instant), because of the random deployment, the longest virtual links connect virtual machines 7 hops apart from each other, while the *Avg* hop count is as high as 6.35 hops. This behavior is expected for random deployment in a Fat-tree topology since the majority of links are expected to have the maximum length for probabilistic reasons, as explained before. After the first optimization, the *OptimizeHops* program was able to reduce more than twofold the *Avg* hop count down to 2.9. However, this program was unable to reduce the *Max* link length below 7 throughout the whole experiment, which means there is always at least one virtual link that connects virtual machines across two separate pods. The 4 virtual topologies chosen (*i.e.*, Ring 4, Tree 7, NetInf, and Tree 15) are very intricate and the program finds difficulty in moving around virtual machines with a large number of connections. It is important to emphasize that the *OptimizeHops* program is not optimal, *i.e.*, it does not consider every possible allocation for all links to find the best configuration. Instead, it tries to optimize link-by-link moving connected virtual machines closer on the physical topology. For most cases the result was satisfactory, since after the first round of optimization the *Avg* hop count remains always below 3, reaching 2.5 by the end of the experiment.

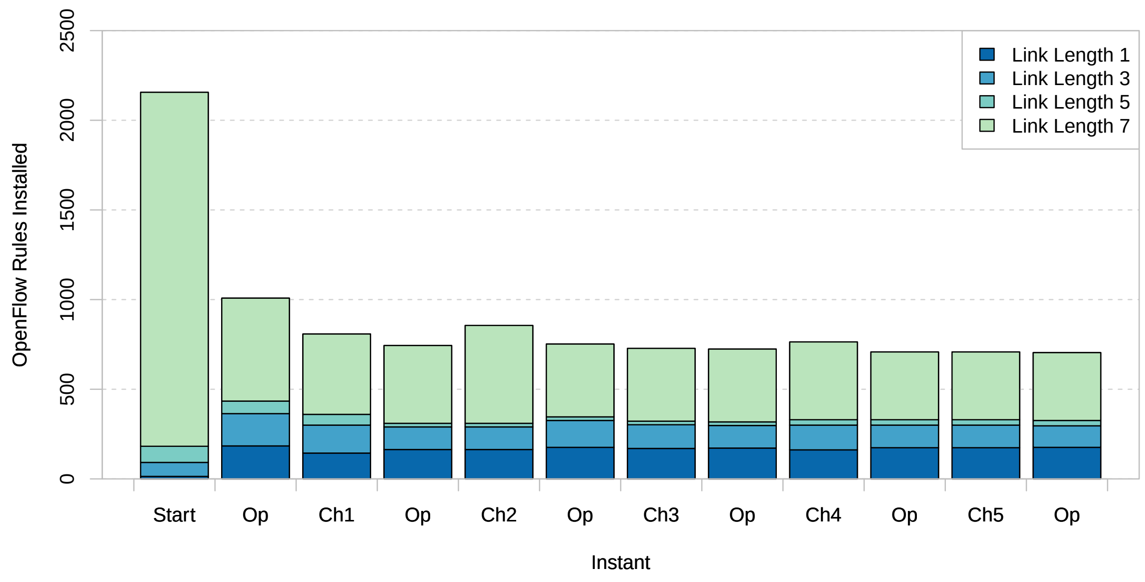
Figure 5.12: Percentage number of virtual links of each possible length



Source: by author (2015).

The high value of *Dev* in Figure 5.11, even after optimizations, also indicates that there is a significant amount of long links (possibly 7 hops-long) and another large set of short links (possibly 1 hop-long). Therefore, Figure 5.12 shows in the y axis the number of virtual links with each length as a percentage of the total number of virtual links. A percentage is used in this chart because the absolute total amount of virtual links increases or decreases depending on the change (*Ch**) performed in the infrastructure, thus the absolute number of virtual links of each length may also vary arbitrarily at instants succeeding a change. As expected, at the *Start* instant, it is possible to see that more than 80% of all virtual links are of length 7. Then, as inferred by the high *Dev* value in the previous chart, after the first optimization instant, 7 hop-long links still account for 20% of the total; the percentage of 1 hop-long links increased significantly to almost 55% though. Intermediate length virtual links, *i.e.*, lengths 3 and 5, are less “attractive” to the optimization program because of the reduced number or possible routes to reallocate a virtual machine within the same edge switch or pod, and still the length of these routes are higher than the prioritized 1 hop-long ones. Although less numerous, virtual links of length 5 seem to decrease very slightly from the beginning of the experiment towards the end, while 3 hop-long links increase at the first optimization instant and remain stable until the last optimization instant. It would be nice if this optimization program could be improved to explore more of the 3 and 5 hop-long routes to minimize even more the number of virtual links of length 7.

Figure 5.13: Resource consumption in terms of OpenFlow rules installed in switches



Source: by author (2015).

An interesting dimension of optimizing link lengths is the potential improvement in network resource consumption. Thus, Figure 5.13 shows the total number of OpenFlow rules installed in all switches of the infrastructure during the experiment with the *OptimizeHops* program. For every virtual link, a number of OpenFlow rules need to be installed in order to instruct switches on how to forward traffic from one virtual machine to another. Every virtual link will install 2 OpenFlow rules (one for ongoing traffic and another for returning) in each switch of its path, thereby a virtual link of length equal to 7 hops will install 14 OpenFlow rules, for instance. At the beginning of the experiment (*Start*) an approximated total number of 2150 rules are installed in the network, being the vast majority due to virtual links of length 7. Since after the first optimization more than half of the virtual links are established with length 1 (as seen in Figure 5.12), the *OptimizeHops* program could reduce the total number of installed OpenFlow rules to nearly 1000. Also, although 1 hop-long links are more numerous throughout the rest of the experiment, they contribute with a very small portion of the total resource consumption in terms of OpenFlow rules installed given their short lengths. This last chart helps in clarifying the actual impact an optimization program can have on the overall network resource consumption. One needs to consider that not only the amount of OpenFlow rules each switch can handle is limited and needs to be carefully managed, but also longer links force traffic to be forwarded through a larger number of devices wasting other resources, such as bandwidth and energy.

5.3 Summary

This chapter presented two case studies to exemplify how resource management in IaaS clouds can be made flexible through the use of deployment and optimization programs. The first case study, presented in Section 5.1, focused on demonstrating how a deployment program can be used to rationally allocate the resources required to establish virtual infrastructures considering a specific ICN application based on the NetInf architecture. The results obtained by monitoring the deployment of 20 *Cloud Slices* of varied sizes and topologies showed that, despite the increase in resource discovery and reasoning times, of the distance-aware deployment program was able to reduce the length of virtual links and consequently the time spent in establishing them in the network. The second case study, presented in Section 5.2, on the other hand, was designed to show the potential benefits of employing optimization programs to reorganize resource allocation of already deployed *Cloud Slices*. Again, 20 *Cloud Slices* of varied sizes and topologies were deployed using a random deployment program, later to be optimized by one of three optimization programs, namely: *OptimizeBalance*, *OptimizeGroup*, and *OptimizeHops*. Afterward, a series of events simulating everyday resource management tasks where triggered, changing the state of resource allocations and forcing each optimization program to reorganize virtual infrastructures according to their specific objectives. The results achieved showed how an *Administrator* can – with a few dozen lines of code – design an optimization program that is able to drive resource management towards entirely different objectives.

6 CONCLUSION

This thesis has discussed the state-of-the-art of providing IaaS over clouds and the limitations found in this context, such as the black-box-like centralized control design of cloud management platforms and the lack of integrated resource management to handle all sorts of resources (*i.e.*, computing, storage, and networking) at the same level of importance. A new concept of cloud management platform was introduced where resource management is made flexible by the addition of programmability at the core of the platform and the introduction of a simplified object-oriented API. In addition, with the proposed API, resource allocation and optimization programs can be written to organize how resource management is performed in a more integrated fashion (all resource handling functions at one place).

Given the limitations exposed in the context of resource management in IaaS cloud environments and the proposal to rely on programmability to make resource management in this context more integrated and flexible, extensive research, development, and experimentation has been conducted aiming to verify the following hypothesis.

Hypothesis: **a cloud platform incorporating the concepts of programmable and integrated resource management can enable the flexibility required to support modern applications and to adapt to environment specific needs**

The investigations conducted and the results presented in this thesis set a clear path towards supporting the proposed hypothesis. The first of two case studies presented has demonstrated how deployment programs may be written to establish complex and particularly network-intensive applications on top of a cloud infrastructure. This case study presented the deployment of a complete ICN application based on the NetInf architecture, with different virtual topologies in under a minute of total deployment time. Besides instantiating a set of virtual machines, like most cloud platforms would support today, the Aurora platform created the virtual topology for the application based on the configuration of virtual links leveraging SDN and OpenFlow. The use of VXDL as the input specification language enabled the deployment of an application in different topologies and with specific resource requirements. The benefits of the distance-aware deployment program implemented could be evidenced by the reduction in the average length of virtual links and a consequent reduction in total link establishment time.

The second case study presented in this thesis aimed to demonstrate that optimization programs written by Administrators can really influence how resource management is performed, optimizing the infrastructure according to the desired objectives. To show the benefits of the proposed platform in terms of flexibility and adaptability to different situations, three optimization programs based on well-known algorithms from the literature have been developed. The OptimizeBalance program was able to keep maximum and minimum residual memory capacities really close to the average and the standard deviation as low as possible. On the other hand, the OptimizeGroup program gathered all the load in 21 hosts of the infrastructure most of the time leaving 35% of the resources idle. The last program, OptimizeHops, brought connected virtual machines close to each other considering the network topology of the underlying infrastructure. The average virtual link length was reduced more than twofold, although the maximum length was never reduced to a value lower than 7 hops. Moreover, the benefits in network resource consumption, in terms of the total amount of OpenFlow rules installed in the network, were also evidenced by the experiments with the OptimizeHops program. In summary, a single platform – incorporating the proposed concepts of programmability and integrated resource management – was able to accommodate a network-intensive application with unusual requirements and to adapt to environment specific objectives.

Based on the work presented in this thesis, it is possible to identify evidences to answer the three research questions (RQ) associated with the hypothesis that have been posed to guide this study. The answers to each question are following detailed.

RQ I. *How to enable clouds to deal with network as a first order manageable resource?*

Answer: Historically, cloud platforms have implemented diverse network functionality and configuration options, most of the time aiming to achieve basic isolation and connectivity needs for virtual machines. There is no single set of abstractions widely used for virtualization of network elements in clouds, like there is for computing and storage (*e.g.*, virtual machines, storage volumes, and operating system images).

In this thesis, concepts of network virtualization have been employed to allow *Cloud Slices* to form complex virtual topologies including two fundamental types of virtual network elements, *i.e.*, virtual routers and virtual links. Abstractions of these elements are made available through the proposed *Unified API*, which provides the required operations to handle virtual network elements at the same level of importance of other types of virtual resources. Furthermore, the platform implemented included drivers to materialize these conceptual elements using modern enabling technologies, such as SDN with OpenFlow and Open vSwitches.

RQ II. *How to enable the detailed specification of applications needs?*

Answer: Specifying and specially achieving particular application needs in terms of resource allocation is a major concern of *End-users* of cloud systems. Recent studies aimed to allow high-level specification of application goals (SUN et al., 2012; WUHIB; STADLER; LINDGREN, 2012; ESTEVES et al., 2013) and proposed complex methods to translate them to actual resource allocation needs. Therefore, such translation was left out of the scope of this thesis. In addition, several attempts have been made to create standard models and interfaces for representing virtual elements and to define the operations to manage them as discussed in Chapter 2.

In this thesis, the initial specification of application needs in terms of virtual resources are expressed through VXDL. This language was chosen because it allows the specification of complete virtual network topologies including all sorts of resources (*e.g.*, virtual routers, virtual links, and virtual machines) in a simple way. The platform developed was able to interpret this language and, with a deployment program, materialize the necessary resource allocations indicated in the initial specification. Moreover, elasticity needs can also be expressed and achieved by resource optimization programs specifically designed to enhance the performance of the deployed application.

RQ III. *How can a cloud platform enable flexible integrated resource management for a wide range of applications and environments?*

Answer: Aligning resource management strategies to the requirements of each specific environment or application is a primary concern for Administrators of cloud systems. Normally, cloud management platforms ship with one or a set of resource allocation strategies hard-coded into their core. Throughout the life-cycle of cloud applications few opportunities are presented for Administrators to influence how resource management is performed.

In this thesis, resource management in cloud environments is made flexible through programmability. The programming framework included in the proposed conceptual architecture and implemented in the Aurora platform allowed easy creation of programs for both deployment and optimization of

Cloud Slices. In addition, the introduced Unified API enabled network virtualization to meet server virtualization granting Administrators with an integrated set of abstractions for all sorts of virtual resources, even including high-level management tasks, such as monitoring performance indicators of physical and virtual elements. The effectiveness of the programming framework and the Unified API have been shown in the two case studies presented, where both application and environment specific needs have been considered while managing resources.

Based on the studies conducted it is possible to identify a few open issues in the investigated topic, which can be subject of future work. For example, the selection of deployment programs could be performed dynamically considering specific types of applications or the current configuration of the cloud environment. The proposed programming framework could also be enhanced by the inclusion of IDE-like features to assist writing and debugging code of deployment, optimization, and metric programs. To facilitate fault handling it would be interesting to have transaction control at the API level, thus resource management programs could include operations such as reservation, commit, and rollback during virtual resource allocation. Furthermore, future investigations can help better evaluating the proposed approach. Optimization programs could be used to improve the performance of running applications, possibly based on metrics personalized by the *End-user* and collected from inside the application. Also, further evaluations of scalability and measurements of the impact of optimizations (specially migrations) on the performance of running applications can be performed. Finally, the proposed management platform has been designed considering a single cloud provider scenario, where all the resources available are under control of the platform and its management programs. As a possible extension work, a multi-provider or multi-datacenter scenario could be explored, where possibly new challenges will arise in the distributed management of resources and collaboration among multiple platforms.

REFERENCES

4WARD. *Architecture and Design for the Future Internet*. 2008. Accessed: September 2014. Available from Internet: <http://www.sail-project.eu/>.

AHLGREN, B. et al. A survey of information-centric networking. **IEEE Communications Magazine**, Piscataway, NJ, USA, vol. 50, no. 7, p. 26–36, 2012. ISSN 0163-6804. Available from Internet: <http://dx.doi.org/10.1109/MCOM.2012.6231276>.

AL-FARES, M.; LOUKISSAS, A.; VAHDAT, A. A scalable, commodity data center network architecture. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, vol. 38, no. 4, p. 63–74, Aug 2008. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1402946.1402967>.

ALI-ELDIN, A.; TORDSSON, J.; ELMROTH, E. An adaptive hybrid elasticity controller for cloud infrastructures. In IEEE NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM (NOMS), 13., 2012, Maui, HI, USA. **Proceedings...** Maui, HI, USA: IEEE Computer Society, 2012. p. 204–212. ISSN 1542-1201. Available from Internet: <http://dx.doi.org/10.1109/NOMS.2012.6211900>.

Amazon. **Amazon Elastic Compute Cloud (Amazon EC2)**. 2013. Accessed: May 2013. Available from Internet: <http://aws.amazon.com/ec2/>.

Azure. **Microsoft Azure**. Microsoft Corporation, 2014. Accessed: May 2014. Available from Internet: <http://azure.microsoft.com/>.

BARI, M. et al. Data Center Network Virtualization: A Survey. **IEEE Communications Surveys Tutorials**, Piscataway, NJ, USA, vol. 15, no. 2, p. 909–928, Second Quarter 2013. ISSN 1553-877X. Available from Internet: <http://dx.doi.org/10.1109/SURV.2012.090512.00043>.

BAUN, C. et al. **Cloud Computing: Web-Based Dynamic IT Services**. 1. ed. New York, NY, USA: Springer Publishing Company, Incorporated, 2011. 100 p. ISBN 978-3-642-20917-8. Available from Internet: <http://www.springer.com/us/book/9783642209161>.

BELOGLAZOV, A.; BUYYA, R. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In IEEE/ACM INTERNATIONAL CONFERENCE ON CLUSTER, CLOUD AND GRID COMPUTING (CCGRID), 10., 2010, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2010. p. 826–831. ISBN 978-0-7695-4039-9. Available from Internet: <http://dx.doi.org/10.1109/CCGRID.2010.46>.

BENSON, T. et al. CloudNaaS: a cloud networking platform for enterprise applications. In ACM SYMPOSIUM ON CLOUD COMPUTING (SOCC), 2., 2011, Cascais, Portugal. **Proceedings...** Cascais, Portugal: ACM, 2011. p. 8:1–8:13. ISBN 978-1-4503-0976-9. Available from Internet: <http://doi.acm.org/10.1145/2038916.2038924>.

BUYYA, R. et al. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. **Future Generation Computer Systems**, vol. 25, no. 6, p. 599–616, 2009. ISSN 0167-739X. Available from Internet: <http://dx.doi.org/10.1016/j.future.2008.12.001>.

CARVALHO, M. Barbosa de et al. A cloud monitoring framework for self-configured monitoring slices based on multiple tools. In INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT (CNSM), 9., 2012, Zürich, Switzerland. **Proceedings...** Zürich, Switzerland: IEEE Communications Society, 2012. p. 180–184. Available from Internet: <http://dx.doi.org/10.1109/CNSM.2013.6727833>.

CHOWDHURY, M.; RAHMAN, M.; BOUTABA, R. ViNEYard - Virtual Network Embedding Algorithms With Coordinated Node and Link Mapping. **IEEE/ACM Transactions on Networking**, Piscataway, NJ, USA, vol. 20, no. 1, p. 206 –219, Feb 2012. Available from Internet: <http://dx.doi.org/10.1109/TNET.2011.2159308>.

CHOWDHURY, N.; BOUTABA, R. Network Virtualization: State of the Art and Research Challenges. **IEEE Communications Magazine**, Piscataway, NJ, USA, vol. 47, no. 7, p. 20–26, July 2009. Available from Internet: <http://dx.doi.org/10.1109/MCOM.2009.5183468>.

CloudStack. **Apache CloudStack: Open Source Cloud Computing**. Apache Software Foundation, 2012. Accessed: December 2013. Available from Internet: <http://cloudstack.apache.org>.

DeltaCloud. **Apache DeltaCloud an API that abstracts the differences between clouds**. Apache Software Foundation, 2011. Accessed: December 2013. Available from Internet: <http://deltacloud.apache.org/>.

Django. **Django 1.6**. Django Software Foundation, 2013. Accessed: January 2014. Available from Internet: <http://www.djangoproject.com/>.

DMTF. **Cloud Infrastructure Management Interface (CIMI) – Version 1.0.0**. Distributed Management Task Force, 2013. Accessed: May 2013. Available from Internet: <http://dmtf.org/standards/cloud>.

DMTF. **Open Virtualization Format (OVF) Specification – Version 2.0.1**. Distributed Management Task Force, 2013. Accessed: December 2013. Available from Internet: <http://dmtf.org/standards/ovf>.

EDMONDS, A. et al. Toward an Open Cloud Standard. **IEEE Internet Computing**, Piscataway, NJ, USA, vol. 16, no. 4, p. 15–25, 2012. ISSN 1089-7801. Available from Internet: <http://dx.doi.org/10.1109/MIC.2012.65>.

ESTEVES, R.; GRANVILLE, L.; BOUTABA, R. On the management of virtual networks. **IEEE Communications Magazine**, Piscataway, NJ, USA, vol. 51, no. 7, p. 80–88, 2013. ISSN 0163-6804. Available from Internet: <http://dx.doi.org/10.1109/MCOM.2013.6553682>.

ESTEVES, R. et al. Paradigm-Based Adaptive Provisioning in Virtualized Data Centers. In IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT (IM), 13., 2013, Ghent, Belgium. **Proceedings...** Ghent, Belgium: IEEE Communications Society, 2013. p. 169–176. ISBN 978-1-4673-5229-1. Available from Internet: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6572983&isnumber=6572961>.

Eucalyptus. **The Open Source Cloud Platform**. Eucalyptus Systems, Inc., 2009. Accessed: September 2014. Available from Internet: <http://open.eucalyptus.com/>.

Floodlight. **Floodlight an Open SDN Controller**. Big Switch Networks, 2014. Accessed: March 2014. Available from Internet: <https://www.projectfloodlight.org/floodlight/>.

FOSTER, I. et al. Cloud Computing and Grid Computing 360-Degree Compared. In GRID COMPUTING ENVIRONMENTS WORKSHOP (GCE), 1., 2008, Austin, TX, USA. **Proceedings...** Austin, TX, USA: IEEE, 2008. p. 1–10. ISBN 978-1-4244-2860-1. Available from Internet: <http://dx.doi.org/10.1109/GCE.2008.4738445>.

FRINCU, M.; CRACIUN, C. Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments. In IEEE INTERNATIONAL CONFERENCE ON UTILITY AND CLOUD COMPUTING (UCC), 4., 2011, Victoria, Australia. **Proceedings...** Victoria, Australia: IEEE, 2011. p. 267–274. ISBN 978-1-4577-2116-8. Available from Internet: <http://dx.doi.org/10.1109/UCC.2011.43>.

GENI. **Global Environment for Network Innovations**. 2011. Accessed: January 2014. Available from Internet: <http://www.geni.net/>.

GUO, L.; GUO, Y.; TIAN, X. IC Cloud: A Design Space for Composable Cloud Computing. In IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING (CLOUD), 3., 2010, Miami, FL, USA. **Proceedings...** Miami, FL, USA: IEEE, 2010. p. 394–401. ISBN 978-1-4244-8207-8. Available from Internet: <http://dx.doi.org/10.1109/CLOUD.2010.18>.

HASAN, M. et al. Integrated and autonomic cloud resource scaling. In IEEE/IFIP WORKSHOP ON CLOUD MANAGEMENT (CLOUDMAN), 3., 2012, Maui, HI, USA. **Proceedings...** Maui, HI, USA: IEEE Communications Society, 2012. p. 1327–1334. ISSN 1542-1201. Available from Internet: <http://dx.doi.org/10.1109/NOMS.2012.6212070>.

HE, S. et al. Elastic Application Container: A Lightweight Approach for Cloud Resource Provisioning. In IEEE INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION NETWORKING AND APPLICATIONS (AINA), 26., 2012, Fukuoka, Japan. **Proceedings...** Fukuoka, Japan: IEEE, 2012. p. 15–22. ISSN 1550-445X. Available from Internet: <http://dx.doi.org/10.1109/AINA.2012.74>.

HÜTTERMANN, M. **DevOps for developers**. New York, NY, USA: Springer Publishing Company, Incorporated, 2012. vol. 1. 196 p. ISBN 978-1-4302-4569-8. Available from Internet: <http://www.apress.com/9781430245698>.

HUU, T. T. et al. Joint Elastic Cloud and Virtual Network Framework for Application Performance-cost Optimization. **Journal of Grid Computing**, Springer Netherlands, vol. 9, no. 1, p. 27–47, 2011. ISSN 1570-7873. Available from Internet: <http://dx.doi.org/10.1007/s10723-010-9168-6>.

ISLAM, S. et al. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. **Future Generation Computer Systems**, Amsterdam, Netherlands, vol. 28, no. 1, p. 155–162, January 2012. ISSN 0167-739X. Available from Internet: <http://dx.doi.org/10.1016/j.future.2011.05.027>.

JENNINGS, B.; STADLER, R. Resource Management in Clouds: Survey and Research Challenges. **Journal of Network and Systems Management**, Springer US, p. 1–53, 2014. ISSN 1064-7570. Available from Internet: <http://dx.doi.org/10.1007/s10922-014-9307-7>.

JESUS, W. P. de; WICKBOLDT, J. A.; GRANVILLE, L. Z. ProViNet - An Open Platform for Programmable Virtual Network Management. In IEEE COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), 37., 2013, Kyoto, Japan. **Proceedings...** Kyoto, Japan: IEEE, 2013. p. 329–338. ISBN 978-0-7695-4987-3. Available from Internet: <http://dx.doi.org/10.1109/COMPSAC.2013.58>.

KHAN, A. et al. Network Virtualization: a Hypervisor for the Internet? **IEEE Communications Magazine**, vol. 50, no. 1, p. 136–143, January 2012. ISSN 0163-6804. Available from Internet: <http://dx.doi.org/10.1109/MCOM.2012.6122544>.

KOHLER, E. et al. The Click Modular Router. **ACM Transactions on Computer Systems (TOCS)**, ACM, New York, NY, USA, vol. 18, no. 3, p. 263–297, August 2000. ISSN 0734-2071. Available from Internet: <http://doi.acm.org/10.1145/354871.354874>.

KÖPSEL, A.; WOESNER, H. OFELIA – Pan-European Test Facility for OpenFlow Experimentation. In ABRAMOWICZ, W. et al. (Ed.). **Towards a Service-Based Internet**. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, vol. 6994). p. 311–312. ISBN 978-3-642-24754-5. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-24755-2_30>.

KOSLOVSKI, G.; SOUDAN, S.; VICAT-BLANC, P. **Modeling Cloud Computing and Cloud Networking with VXDL**. Miyazaki, Japan: World Telecommunications Congress 2012 - Cloud Computing in the Telecom Environment Bridging the Gap Workshop (Presentation), 2012.

KOSLOVSKI, G. P.; PRIMET, P. V.-B.; CHARAO, A. S. VXDL: Virtual Resources and Interconnection Networks Description Language. In **Networks for Grid Applications**. Springer Berlin Heidelberg, 2009. vol. 2, p. 138–154. ISBN 978-3-642-02080-3. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-02080-3_15>.

KUTSCHER, D. et al. **D-B:1 The Network of Information: Architecture and Applications**. Online, 2011. Accessed: February 2012. Available from Internet: <http://www.sail-project.eu/wp-content/uploads/2011/08/SAIL_DB1_v1_0_final-Public.pdf>.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In ACM SIGCOMM WORKSHOP ON HOT TOPICS IN NETWORKS (HOTNETS), 9., 2010, Monterey, CA, USA. **Proceedings...** New York, NY, USA: ACM, 2010. p. 19:1–19:6. ISBN 978-1-4503-0409-2. Available from Internet: <http://doi.acm.org/10.1145/1868447.1868466>.

LENK, A. et al. What's Inside the Cloud? An Architectural Map of the Cloud Landscape. In ICSE WORKSHOP ON SOFTWARE ENGINEERING CHALLENGES OF CLOUD COMPUTING, 1., 2009, Washington, DC, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2009. p. 23–31. ISBN 978-1-4244-3713-9. Available from Internet: <http://dx.doi.org/10.1109/CLOUD.2009.5071529>.

LI, J. et al. CloudOpt: Multi-goal optimization of application deployments across a cloud. In INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT (CNSM), 7., 2011, Paris, France. **Proceedings...** New York, NY, USA: IEEE Communications Society, 2011. p. 1–9. ISBN 978-1-4577-1588-4. Available from Internet: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6103947&isnumber=6103944>.

Libcloud. **Apache Libcloud a unified interface to the cloud**. Apache Software Foundation, 2012. Accessed: December 2013. Available from Internet: <http://libcloud.apache.org/>.

Libvirt. **Libvirt: The Virtualization API – Version 0.9.9**. 2012. Accessed: February 2012. Available from Internet: <http://www.libvirt.org>.

LXC. **Linux Containers**. 2012. Accessed: September 2012. Available from Internet: <http://lxc.sourceforge.net/>.

MANVI, S. S.; SHYAM, G. K. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. **Journal of Network and Computer Applications**, vol. 41, no. 0, p. 424–440, 2014. ISSN 1084-8045. Available from Internet: <http://dx.doi.org/10.1016/j.jnca.2013.10.004>.

MCKEOWN, N. et al. OpenFlow: enabling innovation in campus networks. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, vol. 38, p. 69–74, March 2008. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1355734.1355746>.

MEDHIOUB, H.; MSEKNI, B.; ZEGHLACHE, D. OCNI - Open Cloud Networking Interface. In INTERNATIONAL CONFERENCE ON COMPUTER COMMUNICATIONS AND NETWORKS (ICCCN), 22., 2013, Nassau, Bahamas. **Proceedings...** New York, NY, USA: IEEE Communications Society, 2013. p. 1–8. ISBN 978-1-4673-5774-6. Available from Internet: <http://dx.doi.org/10.1109/ICCCN.2013.6614161>.

MENG, X.; PAPPAS, V.; ZHANG, L. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In IEEE INFOCOM, 23., 2010, San Diego, CA, USA. **Proceedings...** IEEE, 2010. p. 1–9. ISSN 0743-166X. Available from Internet: <http://dx.doi.org/10.1109/INFCOM.2010.5461930>.

Mininet. **An Instant Virtual Network on your Laptop (or other PC) – Version 2.1.0**. 2013. Accessed: October 2014. Available from Internet: <http://mininet.org/>.

MONTES, J. et al. GMonE: A complete approach to cloud monitoring. **Future Generation Computer Systems**, vol. 29, no. 8, p. 2026–2040, 2013. ISSN 0167-739X. Available from Internet: <http://dx.doi.org/10.1016/j.future.2013.02.011>.

MORENO-VOZMEDIANO, R.; MONTERO, R.; LLORENTE, I. IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. **IEEE Computer**, Piscataway, NJ, USA, vol. 45, no. 12, p. 65–72, 2012. ISSN 0018-9162. Available from Internet: <http://dx.doi.org/10.1109/MC.2012.76>.

MORENO-VOZMEDIANO, R.; MONTERO, R.; LLORENTE, I. Key Challenges in Cloud Computing: Enabling the Future Internet of Services. **IEEE Internet Computing**, Piscataway, NJ, USA, vol. 17, no. 4, p. 18–25, 2013. ISSN 1089-7801. Available from Internet: <http://dx.doi.org/10.1109/MIC.2012.69>.

NAGIOS. **Nagios: The Industry Standard In IT Infrastructure Monitoring – Version 3.5.0**. Nagios Enterprises, LLC, 2013. Accessed: September 2014. Available from Internet: <http://www.nagios.org/>.

NURMI, D. et al. The Eucalyptus Open-Source Cloud-Computing System. In IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID), 9., 2009, Shanghai, China. **Proceedings...** New York, NY, USA: IEEE, 2009. p. 124–131. Available from Internet: <http://dx.doi.org/10.1109/CCGRID.2009.93>.

OCF. **OFELIA Control Framework (OCF)**. 2011. Available from Internet: <http://fp7-ofelia.github.io/ocf/>.

OGF. **Open Cloud Computing Interface (OCCI)**. Open Grid Forum, 2012. Accessed: September 2012. Available from Internet: <http://occi-wg.org/>.

Open vSwitch. **An Open Virtual Switch**. 2012. Accessed: September 2012. Available from Internet: <http://openvswitch.org/>.

OpenNebula. **The Open Source Solution for Data Center Virtualization**. 2008. Accessed: Sep. 2014. Available from Internet: <http://www.opennebula.org/>.

ORTIZ, S. The Problem with Cloud-Computing Standardization. **IEEE Computer**, Piscataway, NJ, USA, vol. 44, no. 7, p. 13–16, 2011. ISSN 0018-9162. Available from Internet: <http://dx.doi.org/10.1109/MC.2011.220>.

PATHAN, M.; BUYYA, R.; VAKALI, A. Content delivery networks: State of the art, insights, and imperatives. In BUYYA, R.; PATHAN, M.; VAKALI, A. (Ed.). **Content Delivery Networks**. Springer Berlin Heidelberg, 2008, (Lecture Notes Electrical Engineering, vol. 9). p. 3–32. ISBN 978-3-540-77886-8. Available from Internet: <http://dx.doi.org/10.1007/978-3-540-77887-5_1>.

PENG, C. et al. VDN: Virtual machine image distribution network for cloud data centers. In IEEE INFOCOM, 25., 2012, Orlando, FL, USA. **Proceedings...** New York, NY, USA: IEEE, 2012. p. 181–189. ISSN 0743-166X. Available from Internet: <http://dx.doi.org/10.1109/INFCOM.2012.6195556>.

ProtoGENI. **A prototype implementation and deployment of GENI**. 2012. Accessed: September 2014. Available from Internet: <http://www.protogeni.net/>.

Rackspace. **OpenStack Cloud Software**. Rackspace Cloud Computing, 2010. Accessed: December 2013. Available from Internet: <http://openstack.org/>.

RAO, J. et al. Self-adaptive Provisioning of Virtualized Resources in Cloud Computing. In ACM JOINT INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS (SIGMETRICS), 34., 2011, San Jose, CA, USA. **Proceedings...** New York, NY, USA: ACM, 2011. p. 129–130. ISBN 978-1-4503-0814-4. Available from Internet: <http://doi.acm.org/10.1145/1993744.1993790>.

RAO, J. et al. DynaQoS: Model-free self-tuning fuzzy control of virtualized resources for QoS provisioning. In IEEE INTERNATIONAL WORKSHOP ON QUALITY OF SERVICE (IWQOS), 19., 2011, San Jose, CA, USA. **Proceedings...** New York, NY, USA: IEEE, 2011. p. 1–9. ISSN 1548-615X. Available from Internet: <http://dx.doi.org/10.1109/IWQOS.2011.5931341>.

SAIL. **Scalable and Adaptive Internet Solutions**. 2010. Accessed: September 2014. Available from Internet: <http://www.sail-project.eu/>.

SANTOS, R. L. dos et al. App2net: A Platform to Transfer and Configure Applications on Programmable Virtual Networks. In IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (ISCC), 20., 2015, Golden Bay, Larnaca, Cyprus. **Proceedings...** Golden Bay, Larnaca, Cyprus: IEEE Computer Society, 2015. p. 335–342. Available from Internet: <http://www.ieee-iscc.org/programme.html>.

SEMPOLINSKI, P.; THAIN, D. A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. In IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING TECHNOLOGY AND SCIENCE (CLOUDCOM), 2., 2010, Indianapolis, IN, USA. **Proceedings...** New York, NY, USA: IEEE, 2010. p. 417–426. ISBN 978-1-4244-9405-7. Available from Internet: <http://dx.doi.org/10.1109/CloudCom.2010.42>.

SNIA. **Cloud Data Management Interface (CDMI) – Version 1.0.2**. Storage Networking Industry Association, 2012. Accessed: December 2013. Available from Internet: <http://www.snia.org/cdmi>.

SNIA. **S3 and CDMI: A CDMI Guide for S3 Programmers – Version 1.0**. Storage Networking Industry Association, 2013. Accessed: December 2013. Available from Internet: <http://www.snia.org/cdmi>.

SOTOMAYOR, B. et al. Virtual Infrastructure Management in Private and Hybrid Clouds. **IEEE Internet Computing**, Piscataway, NJ, USA, vol. 13, no. 5, p. 14–22, September-October 2009. ISSN 1089-7801. Available from Internet: <http://dx.doi.org/10.1109/MIC.2009.119>.

STRØMMEN-BAKHTIAR, A.; RAZAVI, A. R. Cloud Computing Business Models. In **Cloud Computing for Enterprise Architectures**. Springer London, 2011. p. 43–60. ISBN 978-1-4471-2236-4. Available from Internet: <http://dx.doi.org/10.1007/978-1-4471-2236-4_3>.

SUN, Y. et al. Mapping Application Requirements to Cloud Resources. In **Euro-Par 2011: Parallel Processing Workshops**. Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, vol. 7155). p. 104–112. ISBN 978-3-642-29736-6. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-29737-3_12>.

TWIDLE, K. et al. Ponder2 - A Policy Environment for Autonomous Pervasive Systems. In IEEE WORKSHOP ON POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS (POLICY), 7., 2008, Palisades, NY, USA. **Proceedings...** New York, NY, USA: IEEE, 2008. p. 245–246. ISBN 978-0-7695-3133-5. Available from Internet: <http://dx.doi.org/10.1109/POLICY.2008.10>.

VAQUERO, L. M. et al. A break in the clouds: towards a cloud definition. **SIGCOMM Computer Communications Review**, ACM, New York, NY, USA, vol. 39, no. 1, p. 50–55, dez. 2008. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1496091.1496100>.

VORAS, I.; ORLIC, M.; MIHALJEVIC, B. The effects of maturation of the Cloud computing market on Open source cloud computing infrastructure projects. In INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY INTERFACES (ITI), 35., 2013, Cavtat, Croatia. **Proceedings...** New York, NY, USA: IEEE, 2013. p. 101–106. ISSN 1334-2762. Available from Internet: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6649005&isnumber=6648975>.

VXDL Forum. **Virtual Infrastructure Description Language (VXDL) - Version 2.0**. 2011. Accessed: February 2012. Available from Internet: <http://www.vxdlforum.org/>.

WANG, Y. et al. Virtual Routers on the Move: Live Router Migration As a Network-management Primitive. **SIGCOMM Computer Communications Review**, ACM, New York, NY, USA, vol. 38, no. 4, p. 231–242, August 2008. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1402946.1402985>.

WUHIB, F.; STADLER, R.; LINDGREN, H. Dynamic resource allocation with management objectives - Implementation for an OpenStack cloud. In INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT (CNSM): MINI-CONFERENCE, 8., 2012, Las Vegas, NV, USA. **Proceedings...** New York, NY, USA: IEEE, 2012. p. 309–315. ISBN 978-1-4673-3134-0. Available from Internet: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6380035&isnumber=6379984>.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. **Journal of Internet Services and Applications**, Springer-Verlag, vol. 1, no. 1, p. 7–18, 2010. ISSN 1867-4828. Available from Internet: <http://dx.doi.org/10.1007/s13174-010-0007-6>.

ZHU, Q.; AGRAWAL, G. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. **IEEE Transactions on Services Computing**, vol. 5, no. 4, p. 497–511, Fourth 2012. ISSN 1939-1374. Available from Internet: <http://dx.doi.org/10.1109/TSC.2011.61>.

APPENDIX A PUBLISHED PAPERS

1. Juliano Araujo Wickboldt, Wanderson Paim de Jesus, Pedro Heleno Isolani, Cristiano Bonato Both, Juergen Rochol, and Lisandro Zambenedetti Granville. Software-Defined Networking: Management Requirements and Challenges. IEEE Communications Magazine: Network and service management series, Volume 53, Issue 1, January (2015), pp. 278-285, ISSN 0163-6804.
2. Lisandro Zambenedetti Granville, Rafael Pereira Esteves, Juliano Araujo Wickboldt. Virtualization in the Cloud. Cloud Services, Networking and Management: Book Chapter of Wiley-IEEE Press. Eds. Nelson Fonseca and Raouf Boutaba. ISBN: 978-1-118-84594-3, 400 pages, April 2015.
3. Juliano Araujo Wickboldt, Rafael Pereira Esteves, Márcio Barbosa de Carvalho, Lisandro Zambenedetti Granville. Resource Management in IaaS Cloud Platforms made Flexible through Programmability. Elsevier Computer Networks, Special Issue on Communications and Networking in the Cloud, Volume 68 (2014), pp. 54-70, ISSN 1389-1286.
4. Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Fabian Schneider, Dominique Dudkowski, and Marcus Brunner. Rethinking Cloud Platforms: Network-aware Flexible Resource Allocation in IaaS Clouds. Proceedings of the mini-conference of 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), 27-31 May 2013, Ghent, Belgium, ISBN: 978-1-4673-5229-1, pp. 450-456.
5. Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Fabian Schneider, Dominique Dudkowski, and Marcus Brunner. A New Approach to the Design of Flexible Cloud Management Platforms. Short-paper in proceedings of 8th International Conference on Network and Service Management (CNSM 2012), 22-26 October 2012, Las Vegas, USA, ISBN: 978-3-901882-48-7, pp. 155-158.
6. Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Fabian Schneider, Dominique Dudkowski, and Marcus Brunner. HyFS Manager: A Hybrid Flash Slice Manager. Demo Presented at IEEE Latin America Conference on Cloud Computing and Communications (LatinCloud 2012): Cloud Tools Lab, 26-27 November 2012, Porto Alegre, Brazil.
7. Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Dominique Dudkowski, and Marcus Brunner. HyFS Manager: A Hybrid Flash Slice Manager. Demo

Presented at 13th IEEE/IFIP Network Operations and Management Symposium (NOMS 2012): Student Demo Contest, 16-20 April 2012, Maui, Hawaii.

APPENDIX B OTHER COLLABORATIONS

1. Anderson Santos da Silva, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Alberto Schaeffer-Filho. ATLANTIC: A Framework for Anomaly Traffic Detection, Classification, and Mitigation in SDN. Proceedings of the 15th IEEE/IFIP Network Operations and Management Symposium (NOMS 2016). 25-29 April, 2016, Istanbul, Turkey. (to appear)
2. Anderson Santos da Silva, Juliano Araujo Wickboldt, Alberto Schaeffer-Filho, Angelos K. Marnerides, Andreas Mauthe. Tool Support for the Evaluation of Anomaly Traffic Classification for Network Resilience. Proceedings of the 20th IEEE Symposium on Computers and Communications (ISCC 2015). 06-09 July, 2015, Golden Bay, Larnaca, Cyprus. (to appear)
3. Cristian Cleder Machado, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Alberto Schaeffer-Filho. An EC-Based Formalism for Policy Refinement in Software-Defined Networking. Proceedings of the 20th IEEE Symposium on Computers and Communications (ISCC 2015). 06-09 July, 2015, Golden Bay, Larnaca, Cyprus. (to appear)
4. Ricardo Luis dos Santos, Oscar Mauricio Caicedo Rendony, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville. App2net: A Platform to Transfer and Configure Applications on Programmable Virtual Networks. Proceedings of the 20th IEEE Symposium on Computers and Communications (ISCC 2015). 06-09 July, 2015, Golden Bay, Larnaca, Cyprus. (to appear)
5. Pedro Heleno Isolani, Juliano Araujo Wickboldt, Cristiano Bonato Both, Juergen Rochol, and Lisandro Zambenedetti Granville. Interactive Monitoring, Visualization, and Configuration of OpenFlow-based SDN. Proceedings of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015). 11-15 May 2015, Ottawa, Canada, ISBN: 978-3-901882-76-0, pp. 207-215.
6. Cristian Cleder Machado, Juliano Araujo Wickboldt, Alberto Schaeffer-Filho, and Lisandro Zambenedetti Granville. Policy Authoring for Software-Defined Networking Management. Proceedings of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015). 11-15 May 2015, Ottawa, Canada, ISBN: 978-3-901882-76-0, pp. 216-224.
7. Eduardo Germano da Silva, Luis Augusto Dias Knob, Juliano Araujo Wickboldt, Luciano Paschoal Gaspary, Lisandro Zambenedetti Granville, and Alberto Egon

Schaeffer-Filho. Capitalizing on SDN-Based SCADA Systems: An Anti-Eavesdropping Case-Study. Proceedings of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015). 11-15 May 2015, Ottawa, Canada, ISBN: 978-3-901882-76-0, pp. 165-173.

8. Pedro Heleno Isolani, Juliano Araujo Wickboldt, Cristiano Bonato Both, Juergen Rochol, and Lisandro Zambenedetti Granville. SDN Interactive Manager: An OpenFlow-Based SDN Manager. Demo Presented at the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015). 11-15 May 2015, Ottawa, Canada, ISBN: 978-3-901882-76-0, pp. 1157-1158. Best Demonstration Award.
9. Runxin Wang, Rafael Pereira Esteves, Lei Shi, Juliano Araujo Wickboldt, Brendan Jennings, and Lisandro Zambenedetti Granville. Network-aware Placement of Virtual Machine Ensembles using Effective Bandwidth Estimation. Proceedings of 10th IEEE/IFIP International Conference on Network and Service Management (CNSM 2014), November 17-21, 2014, Rio de Janeiro, Brazil. pp. 100-108.
10. Cristian Cleder Machado, Lisandro Zambenedetti Granville, Alberto Schaeffer-Filho, Juliano Araujo Wickboldt. Towards SLA Policy Refinement for QoS Management in Software-Defined Networking. Proceedings of 28th IEEE International Conference on Advanced Information Networking and Applications (AINA 2014), May 13-16, 2014, Victoria, Canada. ISSN: 1550-445X, pp. 397-404.
11. Pedro Heleno Isolani, Juliano Araujo Wickboldt, Cristiano Bonato Both, Juergen Rochol, Lisandro Zambenedetti Granville. Uma Análise Quantitativa do Tráfego de Controle em Redes OpenFlow. Proceedings of XIX Workshop de Gerência e Operação de Redes e Serviços (WGRS 2014) together with XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2014), May 5-6, 2014, Florianópolis, SC, Brazil, ISSN: 2177-496X, pp. 75-88. (in Portuguese)
12. Wanderson Paim de Jesus, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville. ProViNet – An Open Platform for Programmable Virtual Network Management. Proceedings of 37th IEEE Computer Software and Applications Conference (COMPSAC 2013), 20-26 July 2013, Kyoto, Japan, ISBN: 978-0-7695-4987-3 pp. 329-338.
13. Ricardo Luis dos Santos, Juliano Araujo Wickboldt, Roben Castagna Lunardi, Bruno Lopes Dalmazo, Lisandro Zambenedetti Granville, Luciano Paschoal Gaspary. Identifying the Root Cause of Failures in IT Changes: Novel Strategies and Trade-offs.

Proceedings of 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), 27-31 May 2013, Ghent, Belgium, ISBN: 978-1-4673-5229-1, pp. 118-125.

14. José Jair Cardoso de Santanna, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville. A BPM-Based Solution for Inter-domain Circuit Management. Proceedings of 13th IEEE/IFIP Network Operations and Management Symposium (NOMS 2012), 16-20 April 2012, Maui, Hawaii, ISBN: 978-1-4673-0268-5, pp. 385-392.
15. Ricardo Luis dos Santos, Juliano Araujo Wickboldt, Roben Castagna Lunardi, Bruno Lopes Dalmazo, Lisandro Zambenedetti Granville, Luciano Paschoal Gaspary, Claudio Bartolini, and Marianne Hickey. A Solution for Identifying the Root Cause of Problems in IT Change Management. Proceedings of Mini-conference of 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), 23-27 May 2011, Dublin, Ireland, ISBN: 978-1-4244-9220-6, pp. 592-599.
16. Bruno Lopes Dalmazo, Weverton Luis da Costa Cordeiro, Abraham Lincoln Rabelo de Sousa, Juliano Araujo Wickboldt, Roben Castagna Lunardi, Ricardo Luis dos Santos, Luciano Paschoal Gaspary, Lisandro Zambenedetti Granville, Claudio Bartolini, and Marianne Hickey. Leveraging IT Project Lifecycle Data to Predict Support Costs. Proceedings of 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), 23-27 May 2011, Dublin, Ireland, ISBN: 978-1-4244-9220-6, pp. 249-256.