Computer Networks 68 (2014) 54–70

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Resource management in IaaS cloud platforms made flexible through programmability

Juliano Araujo Wickboldt *, Rafael Pereira Esteves, Márcio Barbosa de Carvalho, Lisandro Zambenedetti Granville

Institute of Informatics, Federal University of Rio Grande do Sul, Av. Bento Gonçalves, 9500 Porto Alegre, RS, Brazil

ARTICLE INFO

Article history:
Received 7 June 2013
Received in revised form 5 November 2013
Accepted 18 February 2014
Available online 27 February 2014

Keywords:
IaaS cloud
Resource management
Optimization
Programmability
Aurora
Platform

ABSTRACT

Infrastructure as a Service (IaaS) clouds are becoming a customary way to deploy modern Internet applications. Many cloud management platforms are available for one who wants to build a private or public IaaS cloud (*e.g.*, OpenStack, Eucalyptus, OpenNebula). A common design aspect of current platforms regards their black-box-like controlling nature, where cloud administrators have few opportunities to influence how resources are actually managed (*e.g.*, virtual machine placement or virtual link path selection). We envision that administrators could benefit from customizations in resource management strategies to achieve environment specific objectives or to enable application oriented resource allocation. In this article, we introduce a new concept of cloud management platform where resource management is made flexible by the addition of programmability to the core of the platform, with a simplified object-oriented API. We present a proof of concept prototype and an evaluation of three resource management programs on an emulated network using Linux virtualization containers and Open vSwitch running the OpenFlow protocol. Results show the feasibility of our approach and how optimization programs were able to achieve different objectives defined by the administrator.

1. Introduction

Providing Infrastructure as a Service (IaaS) through clouds has drawn much attention from key information technology (IT) players (*e.g.*, Amazon, 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*. The reasons for choosing private instead of public clouds vary, but it commonly encompasses situations 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 and private clouds. Some platforms have been designed to comply with the specific purpose of a cloud provider, such as Amazon EC2 for Amazon's cloud. On the other hand,

* Corresponding author. Tel.: +55 (51) 9861 1336; fax: +55 (51) 3316 7308.

E-mail addresses: jwickboldt@inf.ufrgs.br (J.A. Wickboldt), rpesteves@inf.ufrgs.br (R.P. Esteves), mbcarvalho@inf.ufrgs.br (M.B. de Carvalho), granville@inf.ufrgs.br (L.Z. Granville).

http://dx.doi.org/10.1016/j.comnet.2014.02.018
1389-1286/

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.*, OpenStack [1], Eucalyptus [2], OpenNebula [3]). For example, HP's public cloud services are built with 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. Computing management is tightly related to handling virtual machines to allocate CPU and memory. Storage management, in its turn, enables allocation of persistent – possibly distributed – data volumes over a data center. Lastly, networking management encompasses enabling communication between virtual resources. 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 on one or another concern.

In general, the life-cycle of an application in most of the aforementioned cloud platforms includes four phases:

1. The specification phase, when 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, and an IP address range.
2. The provisioning phase, when the cloud platform allocates the requested resources from the data center.
3. The runtime phase, when 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, when resources are 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 controlling design, which resembles cluster task schedulers. In such a design, from slice specification to 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). We envision that IaaS clouds could benefit from customization in resource allocation strategies under two different perspectives: (i) to achieve environment specific objectives, such as optimizing energy consumption or reducing expensive link utilization, and (ii) to enable application oriented resource management, *e.g.*, placing connected virtual machines close together in the data center to reduce communication delay. Because of the lack of flexibility to accommodate such customization, current platforms fail to support many modern Internet applications – specially highly distributed and network intensive ones – with strict requirements for elasticity or Quality of Service (QoS) [4].

In this article, we introduce a new concept of cloud management platform where resource allocation and optimization are made flexible. We add programmability to the core of the platform with a simplified object oriented API, in such a way that administrators can easily describe and run personalized programs for both application deployment and optimization. In addition, administrators can also use the API to customize metrics and configure events to trigger optimization whenever necessary. The proposed API offers high-level operations to handle all sorts of resources (*i.e.*, computing, storage, and networking), all at the same level of importance. Further than that, administrators can use the API to collect information from the monitoring system in order to use all this information when deploying or optimizing applications.

A prototype has been implemented to show the feasibility of our approach. This prototype supports a wide range of virtualization technologies through Libvirt [5], advanced networking through software-defined networks with OpenFlow [6], and integration with a configurable cloud monitoring framework [7]. We evaluate the developed prototype on an emulated environment, using Linux virtualization containers with LXC [8], Open vSwitches [9] running OpenFlow, and Mininet [10]. Results show a promising path towards enabling flexible resource management in cloud platforms, from deployment to optimization of cloud slices. We show that by using our programmable platform an administrator is able to write, in a just few lines of code, optimization programs that achieve completely distinct objectives.

The remainder of this article is organized as follows. In Section 2, we describe some of the current efforts from both academia and industry to provide IaaS over clouds, focusing mainly on resource management aspects. In Section 3, we present the key concepts that drive our research organized as modules of a conceptual architecture. Following, we detail the prototype developed as a proof of concept in Section 4. Experiments conducted and results achieved using our prototype are presented in Section 5. At last, in Section 6, we conclude this article with final remarks and future work perspectives.

2. Related work

In this section, we examine some of the main proposals towards providing IaaS through cloud platforms. We analyze many different aspects of each proposal, including: (i) to which extent resource management concerns (computing, storage, and networking) have evolved; (ii) what is the approach to "translate" an initial application specification into actual resource allocation; and (iii) how cloud administrators can influence resource allocation and optimization processes.

Cloud computing is a term coined around 2007. Already in 2008, Vaquero et al. [11] were on the pursuit of a consolidated definition of the term by analyzing a large number of previously given definitions. In that study, the 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 cluster and grid environments inside the high-performance computing community. Networking was then only

considered as a means to interconnect virtual resources. Benson et al. [12] also highlighted some of the limitations of cloud platforms in supporting robust networking functions. Recently, Moreno-Vozmediano et al. [4] 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 more complex elasticity rules, based on different types of metrics, is not well supported, including both infrastructure-level and QoS metrics.

Quickly, many cloud management platforms were developed inside 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: Nimbus [13], originated from Virtual Workspaces [14]; OpenNebula [15], from the FP7 project RESERVOIR [16]; Eucalyptus [17], as an outcome of the MAYHEM project; and OpenStack [1], by Rackspace Hosting and NASA. By analyzing the main features of these platforms, it is possible to conclude that they focus mainly on virtualization of computing and storage resources, while networking support is still too basic [18]. The majority of these platforms offers network configuration via DHCP servers to configure IP addresses for virtual machines. Eucalyptus also provides a mode where it is possible to isolate traffic between sets of virtual machines through the use VLAN tags. Only recently, OpenStack started a parallel project called Quantum [19] to further develop its networking capabilities. This project aims to develop the concept of Connectivity-as-a-Service (CaaS) in the platform, by adding transparent VLAN creation, explicit definition of network interfaces of virtual machines, and plug-ins to expose API extensions for more complex functionality support, such as network QoS.

There are 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) [20], from the FP7 Ofelia project [21], allows researchers to reserve resources and run experiments over OpenFlow networks [6]. Another example is ProtoGENI [22], from the GENI project [23], which allows the creation of a virtual infrastructure of interconnected virtual resources, aggregating resources available from many partner federations. However, these platforms do not strictly follow the cloud computing model. The notion of 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 Network as a Service (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.

Although we focus in this article mainly on providing IaaS, it is important to emphasize 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 high level specification of application constraints [24–26] and complex methods of translation to resource allocation. In our solution, we are not as much concerned about allowing the application to be specified in very high level means. However, we do include ways for the user to specify a detailed virtual infrastructure where the application can be deployed – further explained in Section 4.

To turn cloud platforms more customizable, Guo et al. [27] 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 to the same VM 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. [28]. 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 [29]. We argue 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 [30]. We thus advocate for easily programmable optimization strategies and metrics inside the core of cloud platforms.

One last point to emphasize in the current picture of cloud platforms is that many platforms already support some default interoperability interfaces, such as the Open Cloud Computing Interface (OCCI) [31], Cloud Infrastructure Management Interface (CIMI) [32], and the *de facto* standard Amazon EC2 [33]. The use of such interfaces enables other applications to remotely send requests to the platform in order to allocate virtual resources. Nevertheless, that only eliminates the need to access the user interface to request resources. Still, if an administrator needs to really change resource allocation strategies on the core of a cloud platform, it would be necessary to dig into the source code – which is usually accessible in open source platforms, but not really easy to change.

Given the state-of-the-art, we conclude that many solutions exist for specific problems in cloud computing platforms and 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 allowing complex configuration and optimization of them all. Although 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, more flexibility needs to be added to the core of platforms to allow administrators to customize resource management for their specific environment or applications' needs. In the remainder of this paper, we introduce concepts, implementation, and evaluation of our cloud platform solution.

3. Key concepts & architecture

In this section, we present some of the main concepts that drive our research and introduce the conceptual architecture that organizes the components of our solution. To make concepts more didactic, we organize the explanations along the typical process of request, establishment, and maintenance of virtual infrastructures over IaaS clouds.

The two actors involved in the process are the *End-user* and the *Administrator*, as depicted in the top part of Fig. 1. The *End-user* (sometimes also referred to as tenant) is the person or organization interested in "buying" virtual resources for deploying a service or application on top of a cloud based infrastructure. On the other side of the this business relationship, there is the cloud provider, which is represented in our architecture by the figure of an *Administrator*. There are many possible business models for the interaction between *End-users* and cloud providers [34], but in this work we try to keep this relationship as simple as possible. We focus on the provisioning of virtual infrastructures as a service, where it is up to the *End-user* to describe his/her needs and to the cloud provider to allocate and maintain the requested virtual resources accordingly.

The process starts with a request for a set of resources, forming a virtual infrastructure described via a *Initial Specification* document. In our architecture, this request is typically specified and submitted by the *End-user*, although it is also possible that the *Administrator* acts on behalf of the *End-user* to place the request. The *Initial Specification* document should reflect, as much as possible, requirements of the service or application that will run upon the virtual infrastructure it describes. Of course, not all applications have obvious requirements that can be easily mapped to a virtual network setup. Therefore, an intuitive interface to help the specification of such requirements is desirable, but it is also outside the scope of this research. A few standards are under way that can be used for describing virtual infrastructures, such as Virtual Infrastructure Description Language (VXDL) [35], Open Cloud Computing Interface (OCCI) [31], Cloud Infrastructure Management Interface (CIMI) [32], and Open Virtualization Format (OVF) [36].

Subsequently, the *Initial Specification* is parsed and interpreted by the *Cloud Slice Manager* component of the architecture, turning the virtual infrastructure specification into an internal representation, which in our research we call a *Cloud Slice*. The concept of slice is well known in both computer and network virtualization environments, but among cloud related proposals this concept varies in interpretation. For the sake of our platform, we state that a *Cloud Slice* is an aggregation of all different kinds of resources that compose the virtual infrastructure over which an application is deployed (*i.e.*, computing, storage, and networking). A *Cloud Slice* is dynamic in 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.

After the creation of a *Cloud Slice*, the components in the 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*. These three components operate based on the *Resource Management Programs* & *Metrics* written by the *Administrator*. We envision that the platform should be shipped already with a basic set of general purpose programs and metrics. For further customization, its up to the *Administrator* to change these programs and metrics or to write new ones in order to adapt resource management to fit environment or application specific needs. More detailed examples of how these programs and metrics are written are presented in Section 5.

For the deployment of a *Cloud Slice*, the *Deployment Engine* component selects a program to perform the initial resource allocation. So far, we assume that the *Administrator* can either configure the *Deployment Engine* to use a default deployment program or manually select one for each *Cloud Slice*. In future versions of the architecture, we may consider adding another component to perform the deployment program selection logic accordingly. After deployment, the application is ready to run over the virtual infrastructure. During application runtime, the need for optimization of resource allocation may arise. For example,

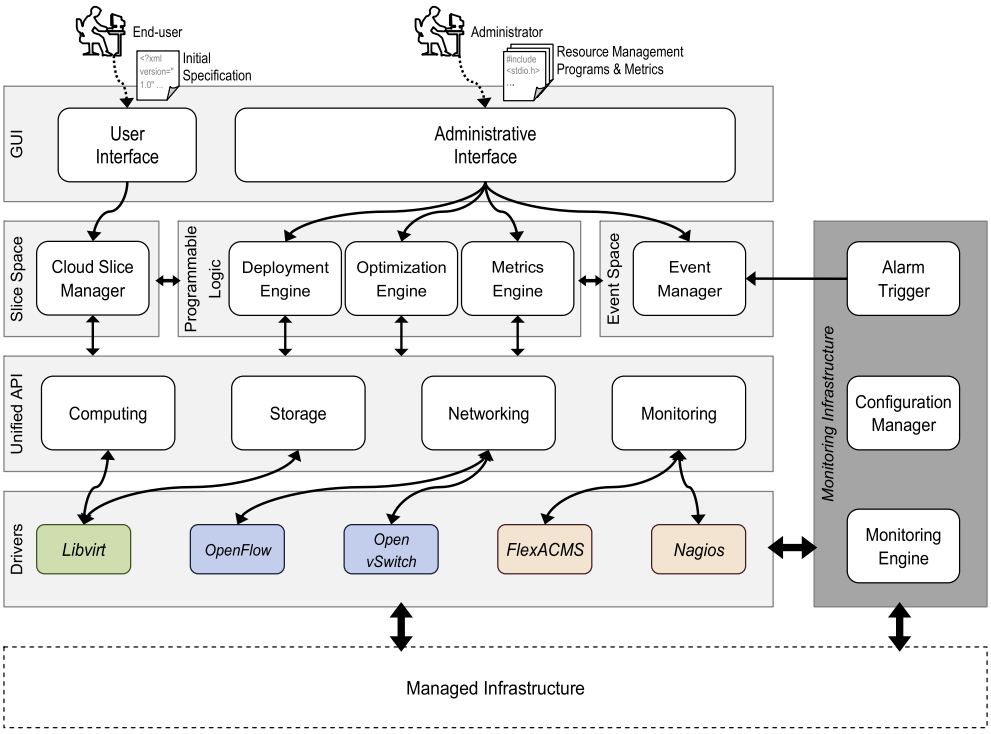

Fig. 1. Conceptual architecture of a cloud management platform.

virtual machines can be migrated to reduce network latency or energy consumption. For this purpose, the *Optimization Engine* employs an optimization program that is executed based on conditions predefined by the *Administrator*. Conditions are expressed in the form of events handled by the *Event Manager* component. 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).

The *Metrics Engine* component also operates based on metric programs. So, basically, any type of metric can be written to be used as condition for an event, including metrics that read information directly from the running application. Metric values may range from simple numerical values associated with the virtual or physical infrastructures (*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). It is important to notice that events, metrics, and optimization programs can all be created and configured to affect a specific *Cloud Slice*, or to be slice independent. For instance, the *Administrator* can write a program to optimize energy consumption affecting all deployed *Cloud Slices*, and another program to optimize a specific *Cloud Slice* based on an application level metric. In this stage of our research, we assume that only one optimization program will affect a *Cloud Slice* at a time, so to avoid conflicting or circular allocation decisions.

All operations regarding handling of virtual resources performed during deployment and optimization of *Cloud Slices* make use of our *Unified API*. This API aims to provide a simple interface for manipulating all types of resources that may compose a *Cloud Slice* at the same level of importance. The *Computing* component takes care of basic functionalities for virtual machines (*i.e.*, creating, staring, stopping, migrating) and guest images (*i.e.*, operating system installation). The *Storage* component deals with allocation of virtual volume and pools abstraction. The *Networking* component implements interfaces for creation of two types of abstractions: virtual links and virtual routers.

We also add in our conceptual architecture a fourth component as part of the *Unified API* to enable *Monitoring* to be considered as an "allocable" type of resource. This is 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 is also configured. Moreover, this component 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. The interactions of our proposed platform and the associated *Monitoring Infrastructure* are further clarified in Section 4.

Virtual device abstractions are implemented on the underlying infrastructure by a set of *Drivers* available at the bottom section of our architecture. These *Drivers* are technology specific pieces of code, which play two main conceptual roles: (i) they abstract complexity of technology specific configuration parameters and communication protocols from the *Unified API* implementation, and

(ii) they make the cloud management platform more portable, *i.e.*, *Drivers* can be replaced as technology evolves, as long as the provided set of functionalities remains the same. It is important to emphasize that, the set of *Drivers* displayed in our conceptual architecture serves only as an example of technologies – which we actually implemented in our prototype – that allow the establishment of the virtual device abstractions over a real infrastructure.

4. Prototype implementation

As a proof of concept, we have implemented a prototype cloud management platform – which we called Aurora Cloud Manager – following the design aspects of our conceptual architecture. Most of the implementation has been performed using the Python programming language. We have brought together several third party tools, libraries, and systems in order to materialize the proposed concepts. We have chosen not to rely on a complete platform, such as OpenStack or OpenNebula, because we did not want to inherit their internal complexity. Lower level libraries, such as Libvirt and OpenFlow, provide abstractions sufficient for node virtualization and networking operations that our platform requires. In this section, we present, in a first moment, an overview of the interactions of Aurora with other external systems. After that, we detail the implementation of some of our platform's core components.

Fig. 2 presents a high level overview of the interactions between the Aurora platform and two other systems that compose the external *Monitoring Infrastructure*. As previously indicated by the disposition of components in our conceptual architecture, we chose not to implement monitoring functions directly into the platform's core, since there are complete solutions for that. Instead, we decided to rely on a tool called Flexible Automated Cloud Monitoring Slices (FlexACMS) [7] that builds *Monitoring Slices* automatically to reflect a *Cloud Slice* creation. A *Monitoring Slice* reflects the corresponding monitoring metrics and configuration in diverse monitoring systems that are necessary to monitor appropriately all resources of a *Cloud Slice*.

FlexACMS uses a *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), 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 optimization programs, such as CPU consumption or network interface statistics. FlexACMS also provides several predefined templates for monitoring virtual elements. Therefore, it is possible, in the XML description, to tag these elements with keywords informing what should be monitored by default in every one of them. For example, if a virtual machine is an HTTP or mail server, or if a virtual router is an OpenFlow switch, there will be specific related metrics that should be monitored

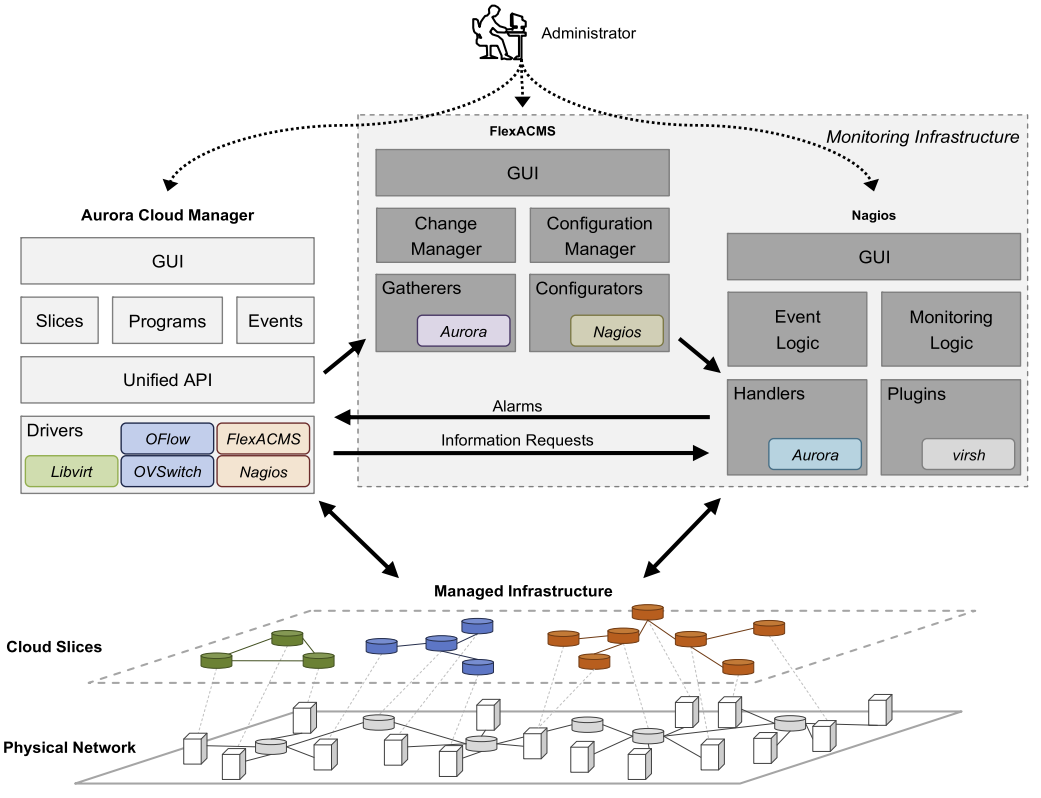

Fig. 2. Interactions between the Aurora platform and the monitoring infrastructure.

in order to assure the correct operation of the specific systems. This information can also be passed through the XML description for automatic configuration of the monitoring infrastructure accordingly.

In our prototype, FlexACMS builds the configuration of *Monitoring Slices* and their metrics based on Nagios [37]. Nagios was chosen to be the system that actually monitors both virtual and physical layers of the *Managed Infrastructure*. We chose this monitoring system mainly because (i) it is widely employed in large scale real infrastructures, and (ii) it has all the built-in features we needed, (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 tags for physical nodes and switches. As shown in Fig. 2, the *Administrator* needs to interact with all three systems to access and configure different platforms. In future versions of our solution, we will provide relevant monitoring information through the Aurora platform GUI, in order to provide both *Administrators* and *End-users* with handy information associated with *Cloud Slices*.

In regards to the implementation of the core components of the Aurora platform, we initially describe the *End-user*'s input to the system. The *Initial Specification* of a virtual infrastructure is described in our prototype according to an extension of the Virtual Infrastructure Description Language (VXDL) [35,38]. 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. Our prototype currently does not include a designer for the virtual topology, which means the user needs to prepare a VXDL file elsewhere and then submit it to the Aurora platform.

In Fig. 3, we present 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, 128 MB of RAM, and 500 MB of maximum storage capacity. Moreover, we specify 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 our prototype, this information can be used by programs to compute link utilization in terms of allocated bandwidth, for example. Although we do not include any `vRouters` in the VXDL samples, they are supported by the prototype and are implemented as Open vSwitches. In a previous work we have exploited the dynamic creation of virtual routers as OpenFlow switches controlled from outside the platform [39].

The core of our platform is written in Python and implemented as a Web based application. We have employed a three-layered Model-View-Template (MVT) framework called Django [40]. The Template layer implements the presentation of the graphical user interface. Through this interface, 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 this interface are presented in Figs. 4 and 5. In Fig. 4, we show the Aurora's slice management interface used by both *End-user* and *Administrator* to handle virtual resources and visualize the virtual topology of a

```
<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>
```

Fig. 3. Sample of initial specification with VXDL.

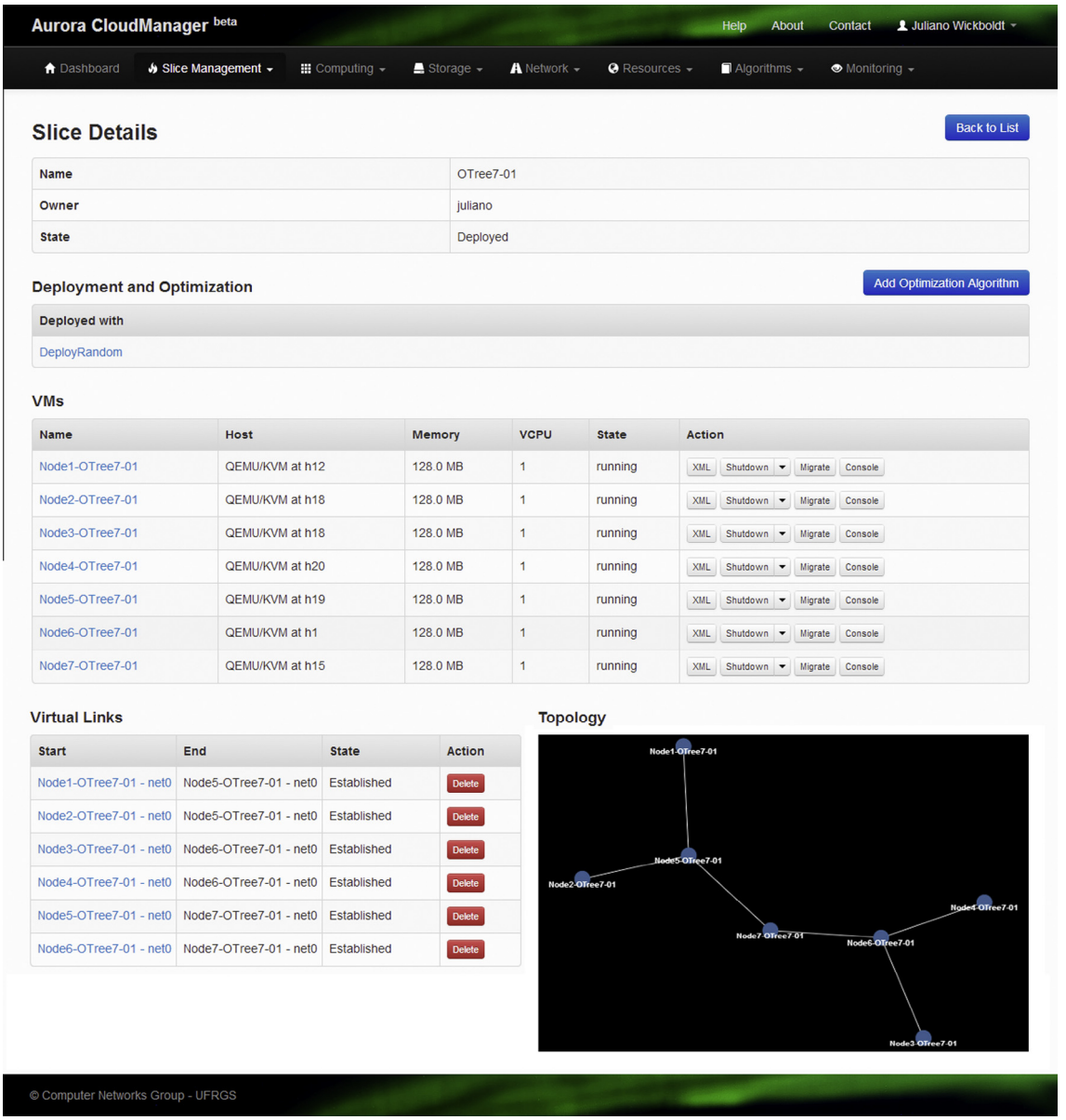

Fig. 4. Screenshot of the slice management graphical user interface.

Cloud Slice. Fig. 5 presents the interface for editing optimization programs. The edition of programs and metrics through the platform is currently performed in plain text. In the future, we intend to add an IDE-like feature, including integration with the API calls and debugging options.

The View layer is where the logic of the platform is implemented, including basic functions for managing virtual and physical resources (*i.e.*, starting/stopping/migrating virtual machines, configuring access to hypervisors and network controllers). Also at the View layer, the base framework for *Resource Management Programs* & *Metrics* is implemented. When a *Cloud Slice* needs to be deployed or optimized, one of the programs available for each purpose performs the actual resource allocation actions. Deployment programs will act when a new *Cloud Slice* is set to be deployed. The selection of programs to be used can be by default settings of the platform, or manually selected by the specific needs of the slice. Optimization programs and metrics can be both designed to operate on a specific *Cloud Slice* or under global scope. Global optimization programs and metrics will only affect slices that do not have specific programs associated, so to avoid circular or conflicting optimization decisions.

Resource Management Programs & *Metrics* are implemented as Python code directly into the platform's core. For the person who designs these programs and metrics, the platform provides 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,

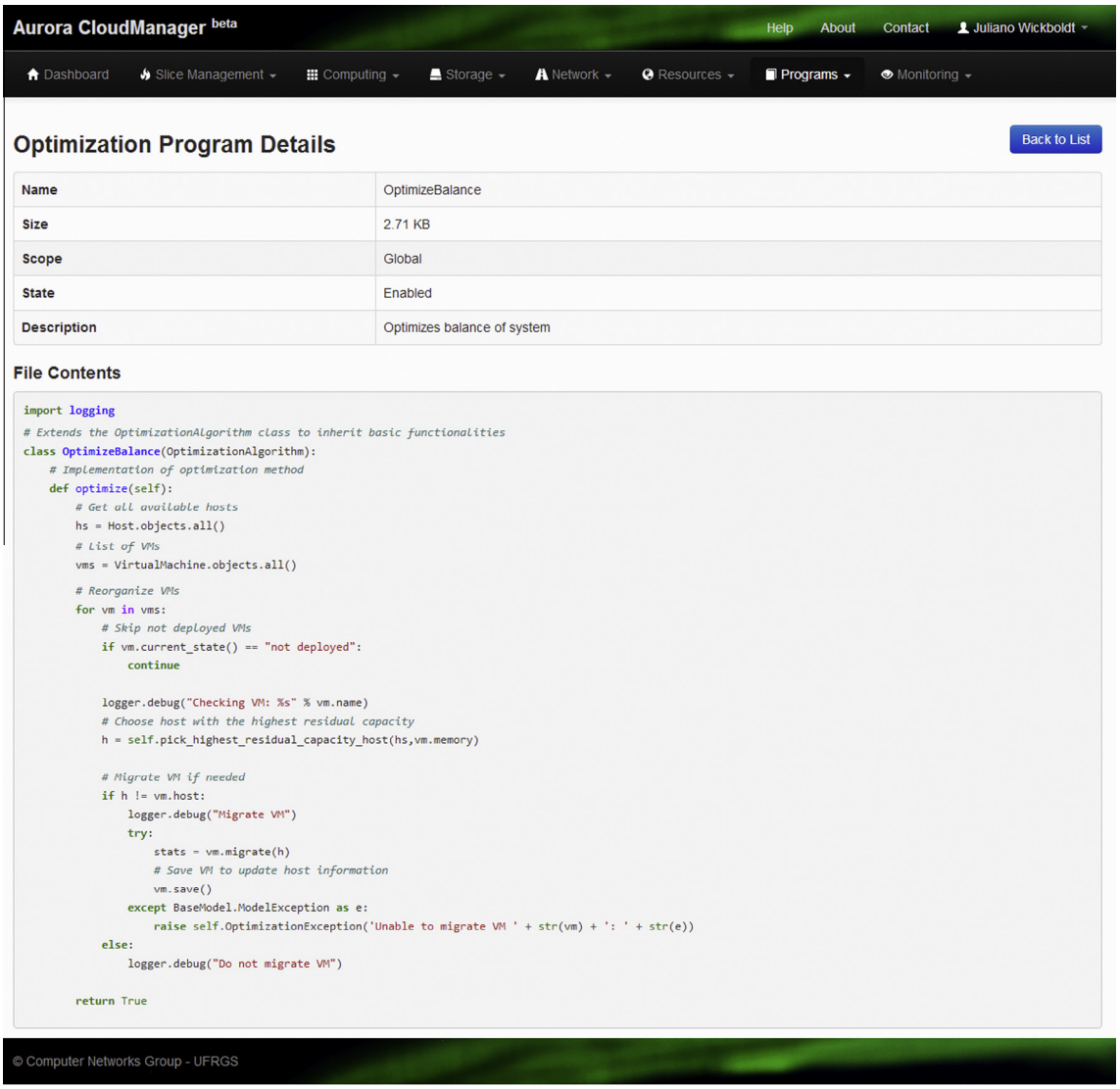

Fig. 5. Screenshot of the optimization program management graphical user interface.

these APIs differ greatly from ours because they are intended to be used by someone from outside the cloud, so they usually hide many internal details of virtual resources. We chose to use a programming language instead of remote calls 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 the platform.

The operations provided by our API are organized into four components according to the architecture presented in Section 3: *Computing*, *Storage*, *Networking*, and *Monitoring*. Similar operations are available through some other remote call APIs, although the focus of usage is different, as explained earlier. Most commonly virtual machine and storage management operations are implemented, although networking and specially monitoring operations are rarely considered. In our API, we try to maintain all operations and abstractions at the same level of importance, to provide a complete framework for *Administrators* to write their resource management programs. The main operations available through our proposed API are following described:

- Computing (Virtual Machines)
  - –*Create*/*Remove:* defines/undefines the internal representation of a virtual machine with its specified characteristics (*e.g.*, CPU, memory, guest image).
  - –*Deploy*/*Undeploy:* defines/undefines a virtual machine on within the hypervisor of a node of the cloud infrastructure, including the transfer/removal of the image file.
  - –*Start*/*Stop*/*Suspend*/*Resume:* basic operations to handle the state of the guest operating system.
  - –*Migrate:* undefines a virtual machine in one node and defines it on another. The destination node needs be specified.

- Computing (Images)
  - –*Create*/*Remove:* defines/undefines a guest operating system image at the main repository of the platform, including the transfer/removal of the file.
- Storage (Virtual Volumes)
  - –*Create*/*Remove:* allocates/deletes chunks of storage on nodes.
  - –*Attach Volume to Virtual Machine:* attaches a volume file to a given virtual machine.
- Storage (Virtual Volume Pools)
  - –*Create*/*Remove:* defines/undefines a pool for storing virtual volumes (typically a local or remote/NFS directory).
  - –*Add Volume:* adds a virtual volume to a volume pool.
- Networking (Virtual Links)
  - –*Create*/*Remove:* defines/undefines the internal representation of a virtual link, which connects point-to-point two virtual interfaces of two virtual devices (*i.e.*, virtual machines or virtual routers).
  - –*Establish*/*Disable:* establishes/disables the virtual link within the network enabling/disabling traffic to flow between the connected devices.
- Networking (Virtual Routers)
  - –*Create*/*Remove:* defines/undefines the internal representation of a virtual router, which has many virtual ports to interconnect many virtual interfaces of virtual devices.
  - –*Deploy*/*Undeploy:* deploys/undeploys the virtual router in a node of the infrastructure.
- Monitoring (Virtual Devices)
  - –*Monitor*/*Unmonitor:* deploys/undeploys the monitoring infrastructure required to monitor a given virtual device.
  - –*Get Monitoring Information:* fetches monitoring information within the monitoring system for a given virtual device.
- Monitoring (Events)
  - –*Create*/*Remove:* defines/undefines the internal representation of an event, which may belong to a specific slice or operate in global scope.
  - –*Deploy*/*Undeploy:* deploys/undeploys the event on the monitoring infrastructure to be triggered on demand.
- Monitoring (Physical)
  - –*Discover Resources:* this is actually a collection of operations to discover nodes and network topology available on the infrastructure. This collection also retrieves information about resource allocation on these physical elements.
  - –*Get Monitoring Information:* fetches monitoring information within the monitoring system for a given physical device (*e.g.*, node or switch).

To implement the *Drivers* section of the conceptual architecture, we have developed a set of pieces of code that relies on technology specific libraries and systems 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 allocation 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 set-up of each environment.

For the *Networking* abstraction of virtual link, we have implemented a driver for software-defined networking based on the OpenFlow technology. OpenFlow is pretty suitable for the task, since all the network can be controlled from a logically centralized element, called controller. Thus, to establish virtual links, the Aurora platform communicates with a POX controller [41] pushing the appropriate OpenFlow forwarding rules. Therefore, in our current implementation scenario, we assume a managed infrastructure fully connected 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, we do not want to impose that deployed applications run necessarily on TCP/IP, like most platforms do.

For virtual router abstraction, we use Open vSwitch technology, 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 [39]. Another form of implementing virtual routers is by deploying virtual machines running router images; 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) our current development is based on Libvirt (Python binding) [5]. 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 our *Unified API* are sufficiently provided by Libvirt. The image abstraction is implemented based on simple file copy operations on a centralized network storage, accessible by all nodes of the infrastructure. Monitoring drivers where implemented based on FlexACMS and Nagios plug-ins, as explained in the beginning of this section.

Finally, in order to obtain the results presented in the remainder of this paper, we have reproduced an emulated infrastructure to be able to create slightly complex datacenter topology. Further details on the setup of the experiments and the case study are presented in Section 5.

5. Experiments

To evaluate our proposal, we have conducted a series of experiments over an emulated infrastructure. The main purpose of the experiments described in this section is to show the feasibility of our approach and how an administrator can benefit from our proposed platform to easily write programs to achieve various distinct objectives. We perform our experiments based on a Mininet [10]

topology, which is the most popular tool for realistic emulation of OpenFlow networks today. We have created the topology depicted in Fig. 6 with 10 Open vSwtiches running OpenFlow protocol and being coordinated by a POX controller. All 20 hosts of this infrastructure are based on Linux virtualization containers with LXC.This setup is started as a Mininet script on one physical machine where our experiments were conducted. The hardware specification of this machine is: Dell PowerEdge R815 server with 4 AMD Opteron Processor 6276 Eight-Core processors and 64 GB of RAM memory. The Aurora platform is also installed on this machine and all LXC hosts run one independent instance of Libvirt to receive commands from our platform. The monitoring infrastructure with FlexACMS and Nagios run on another physical machine and the communication between the systems goes over an HTTPS connection on an ordinary LAN.

We have defined 3 different virtual topologies to be deployed in the form of *Cloud Slices*, as shown in Fig. 7. Topologies (*a*) and (*b*) are organized as binary trees, containing respectively 7 and 15 virtual machines, while topology (*c*) is a ring with 4 virtual machines. The deployed virtual machines boot OpenWRT router images [42]. Therefore, the traffic generated between virtual machines connected with virtual links flows through the OpenFlow rules created by our platform. Traffic that goes between two virtual machines of the same *Cloud Slice* that are not directly connected is forwarded by each OpenWRT in the path. Although in these experiments we do not target any specific application, creating virtual topologies can be useful for many kinds of applications. For example, it is common to find content distribution systems or information-centric networking applications that rely on a specific topology for efficient content routing [43]. Moreover, we did not consider topologies with virtual routers in these experiments because, at this stage of implementation, migrating these elements is still not supported.

We have initially defined three optimization programs in our case study. The first program, called *OptimizeBalance* (Algorithm 1), spreads virtual machines of a *Cloud Slice* over the managed infrastructure. For each virtual machine of the slice, the program picks the physical node with the highest residual capacity in terms of memory (line 4) and migrates the virtual machine to it (line 6). By selecting the server having the highest residual capacity, *OptimizeBalance*P maps a virtual machine to the least used physical nodes. If the physical nodes are homogeneous (*i.e.*, nodes have the same configuration and total capacity), the algorithm selects a different node for each VM of a request.

Algorithm 1. OptimizeBalance program

```
        pseudocode
1: H ← set of physical hosts
2: V ← set of virtual machines
3: for each vm ∈ V do
4: candidate ← 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 2), goes into the opposite direction by mapping virtual machines into the smallest subset of physical nodes. This is done by migrating virtual machines to the nodes with the lowest residual capacity in terms of memory. Again, if physical nodes have the same initial capacity, the algo-

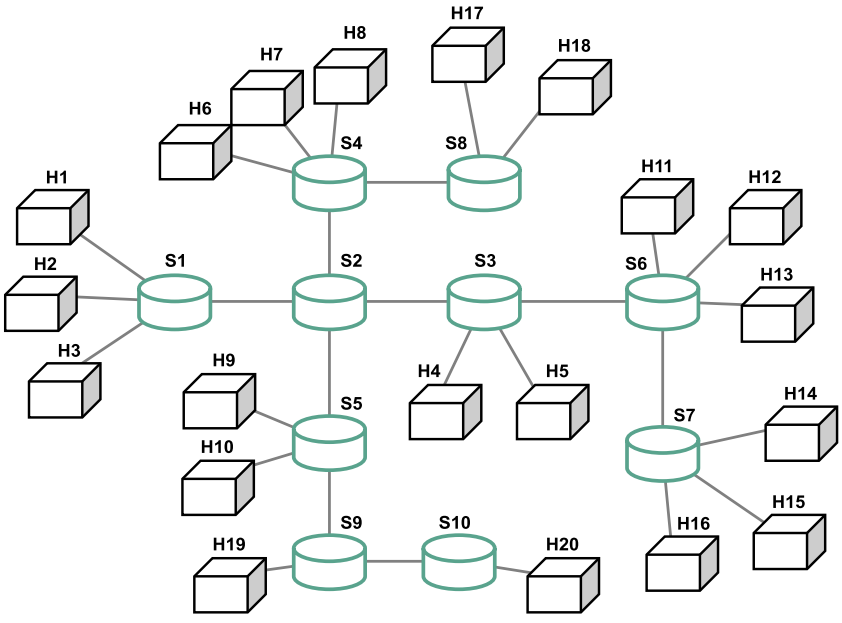

Fig. 6. Emulated physical topology based on LXC and Open vSwitches.

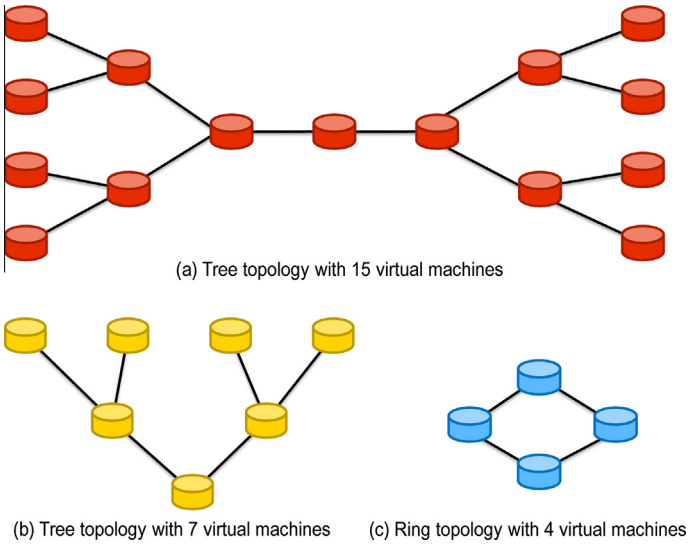

Fig. 7. Virtual infrastructure topologies deployed.

rithm chooses the same node to host all VMs of a request until the selected nodes runs out of capacity; in which case, a new node is selected. *OptimizeGroup* can be used for energy-saving purposes, for example, since a smaller number of nodes will actually host many virtual machines, while others become idle.

Algorithm 2. OptimizeGroup program

```
        pseudocode
1: H ← set of physical hosts
2: V ← set of virtual machines
3: for each vm ∈ V do
4: candidate ← get_lowest_residual_capacity_host
   (H, vm)
5:     if candidate != vm.host then
6:         Migrate (candidate, vm)
7:     end if
8: end for
```

The third program, called *OptimizeHops* (Algorithm 3), reduces the number of hops between linked virtual machines of a *Cloud Slice*. This is achieved by analyzing pairs of virtual machines connected by a virtual link (line 4), fixing one of them as a *pivot*, and moving the other (referred to as *free*) onto the node located at the shortest distance in the physical network with the required resource available. To decide which virtual machine is the *pivot* and which one is *free* to move, we check their total number of virtual links (line 5). This program will always try to select as *free* the virtual machine connected to the smallest number of virtual links, as an attempt to minimize migrations. Finally, if the *candidate* node is located at a closer distance to the *pivot*, then the current node where *free* is (line 13) will be migrated to *candidate*. In this program, we assume that virtual machines connected by virtual links will communicate often, therefore placing them close together in the network is a way to minimize the traffic in the physical links. For the initial deployment of slices, we have used a program which simply randomly deploys each virtual machine at a node with enough available resources.

Algorithm 3. OptimizeHops program

```
        pseudocode
1: H ← set of physical hosts
2: L ← set of virtual links
3: P ← set of pairs (vm_from, vm_to) ∈ L
4: for each p ∈ P do
5:   if link_count(p.vm_from) > link_count(p.vm_to)
   then
6:     pivot ← p.vm_from
7:     free ← p.vm_to
8:   else
9:     pivot ← p.vm_to
10:    free ← p.vm_from
11:  end if
12:  candidate ← get_closest_host_with_capacity
     (H, pivot, free)
13:  if distance(pivot.host, candidate) < p.length then
14:    Migrate (candidate, free)
15:  end if
16: end for
```

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 the choice for one can depend on many factors, such as budget constraints, overall objective, 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 our proposed API. Just to provide a rough idea, OptimizeBalance and OptimizeGroup programs are less than 40 lines of code. The OptimizeHops program, which includes all pair processing and distance calculations, is still only 125 lines long.

It is important to point out that the algorithms used here are not completely novel. Instead, they are inspired in other proposals found in the literature. OptimizeBalance aims to achieve the same load balancing objective of the algorithm proposed by Chowdhury et al. for virtual network embedding [44]. OptimizeGroup uses migration to concentrate virtual machines on a small set of physical nodes, which was envisioned by the VROOM architecture [45], in the context of virtual routers. Similar to the work proposed by Meng et al. [46], OptimizeHops aims to reduce the communication cost between virtual machines by reducing the distance between them.

The optimization programs we have used 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 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 are listed in Table 1.

Following, we define the scenarios designed to illustrate the benefits of our optimization programs. Initially, we assume 6 *Cloud Slices* (two of each in Fig. 7) are already deployed over the infrastructure with no optimization whatsoever. Then, we execute the following list of changes in the allocations and let the programs optimize the infrastructure accordingly:

1. A previously allocated *Cloud Slice* expires and releases resources. In this case, other active slices can be reconfigured to benefit from the changes in the infrastructure (*Slice termination event*).
2. A *Cloud Slice* is modified by the addition of virtual machines. Here, optimization is required to prevent that the placement of the new virtual machines conflicts with the active policy (*Slice modification event*).
3. The removal of virtual machines from a *Cloud Slice* 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 the nodes. 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 make deployed *Cloud Slices* take advantage of the new resources (*Administration event*).

We have monitored the changes and optimizations of the infrastructure during experimentation for all three optimization programs previously explained. The charts presented in Figs. 8 and 9 show the average residual memory capacity of the infrastructure respectively for OptimizeBalance and OptimizeGroup programs. The residual memory capacity (presented in the *Free Memory* axis) 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 node defined in our experiment is 4025 MB; therefore, the *Max* value never overcomes 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 (6 *Cloud Slices* randomly deployed). Every *Op* instant represents the memory capacities right after an optimization occurred, while *Ch** are instants that succeed a change, according to the list of changes previously explained.

In Fig. 8, it is possible to notice 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 *Mean* residual memory capacities, which means that there are nodes heavily loaded while others are not. Also, *Stdv* (standard deviation) is high at this instant, which indicates high variation between residual memory capacities of nodes. At the first *Op* instant, the OptimizeBalance program was already able to distribute the load among nodes, dropping the *Stdv* drastically and leading *Max*, *Min*, and *Mean* 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 *Mean* values and the increase of *Stdv*. Subsequent optimizations reconduct the infrastructure back to an optimized state.

Fig. 9 shows the same measures as Fig. 8, but considering the OptimizeGroup program. In this program, for the infrastructure to be considered optimized, *Min* and *Max* values need to be as far apart as possible, while *Stdv* needs to be high. This means that there are nodes with a lot of free memory (high *Max*) and others with their capacities almost fulfilled (low *Min*). That is a behavior that can be observed right after the first optimization.

The difference between the OptimizeBalance and OptimizeGroup programs is more clearly visible in Figs. 10 and 11. These charts show memory allocations per node throughout experimentation. The *Host ID* axis represents every node of the infrastructure and the *Instant* axis – just like Figs. 8 and 9 – represents the time span of experiments. The size of the bubbles in these charts displays the total memory allocated in every node for virtual machines. In Fig. 10 it is easy to visualize that at the beginning (*Start* instant) memory allocations seem unbalanced. Optimizations (*Op* instants) tend to bring all bubbles to the same size, whereas changes (*Ch** instants) reintroduce disturbance in allocations. It is interesting to notice that in *Ch4* the *Administrator* removes node number 10 from the infrastructure to perform some maintenance tasks. Because of this change, the *Administrator* migrated the virtual

Table 1
Triggering events.

Event name	Description
Slice creation	A new slice is created and can affect the already deployed ones
Slice termination	The duration time of the cloud slice has expired and it is no longer active
Slice modification	The slice is modified by adding or removing virtual machines to/from it
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)

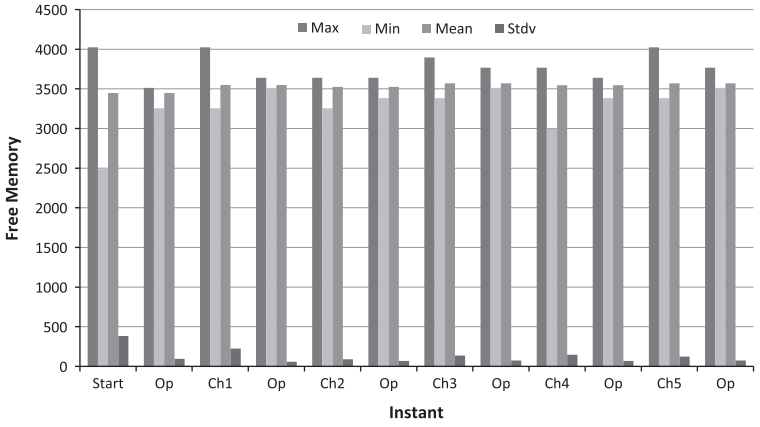

Fig. 8. Residual memory capacity using OptimizeBalance program.

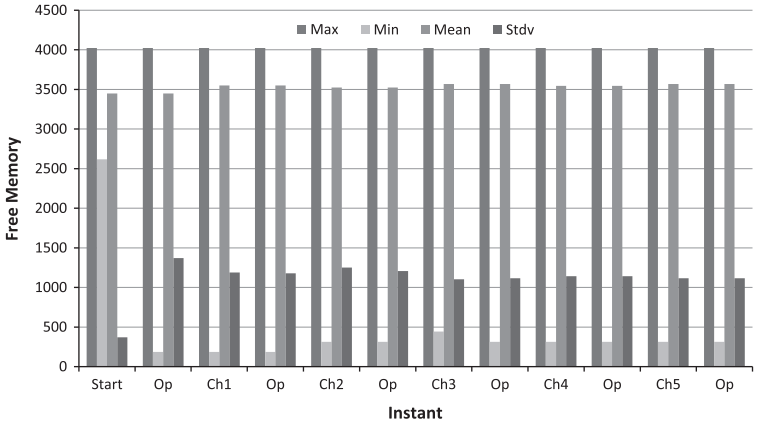

Fig. 9. Residual memory capacity using OptimizeGroup program.

machines from node 10 to node 15. The following optimization redistributed the load from node 15 to other nodes. After that, the next change (*Ch5*) reintroduces node number 10 and, on the optimization next to this change, node 10 receives virtual machines again.

In Fig. 11, it is possible to visualize that the behavior of memory allocations guided by the OptimizeGroup program is completely opposite to OptimizeBalance. After the first optimization, all the load is distributed among only 3 nodes, making all others idle. The OptimizeGroup program, if associated with energy management functions to turn on and off nodes on demand, can be certainly used for energy saving purposes.

The last chart presented in Fig. 12 concerns the results achieved with the OptimizeHops program. This chart is very similar to the ones presented in Figs. 8 and 9. However, instead of residual memory, the distance of pairs of virtual machines connected by virtual links is displayed on the *Hop Count* axis. Although we did not include routers in our emulated topology, we consider every switch as one hop in this experiment. At the beginning (*Start* instant), because of the random deployment, the longest virtual links connect virtual machines 6 hops apart from each other, while the *Mean* hop count is as high as 3.5 hops. After the first optimization, the OptimizeHops program was able to reduce the *Max* hop count to 5 and the *Mean* close to 1. It is important to

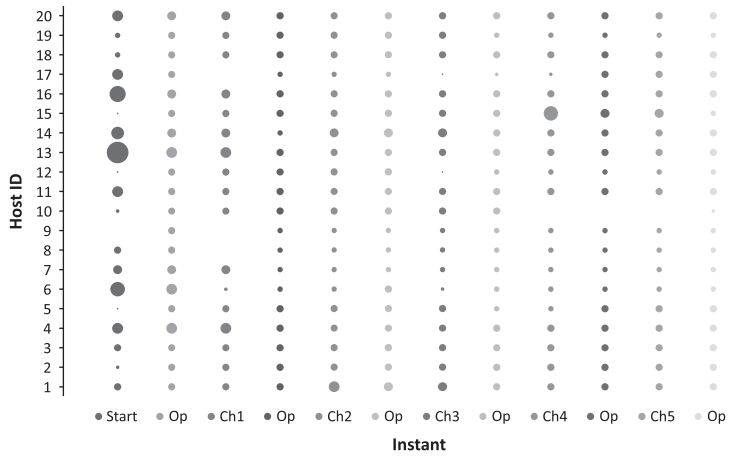

Fig. 10. Memory allocations per node using OptimizeBalance program.

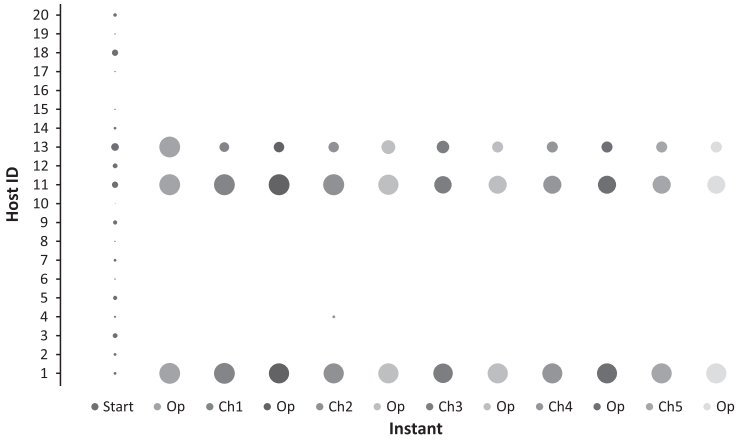

Fig. 11. Memory allocations per node using OptimizeGroup program.

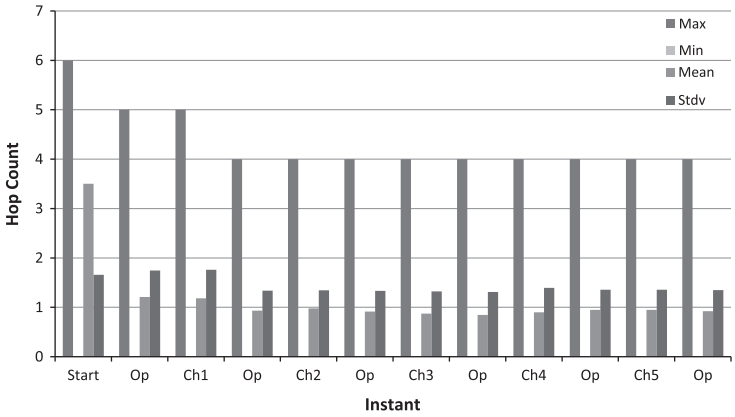

Fig. 12. Hop counts of virtual links using OptimizeHops program.

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 of the cases, the result was satisfactory, since after two rounds of

optimization the *Mean* hop count remains always below 1. However, this seems to be the limit for this program, since it was not able to reduce the *Max* hop count lower than 4.

6. Conclusion

In this paper, we have discussed the current picture of providing IaaS over clouds and the limitations found in this scenario. We have also introduced a new concept of cloud management platform 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. A prototype cloud management platform has been implemented following the concepts proposed and tested over an emulated infrastructure combining OpenFlow switches and LXC nodes.

Results obtained show the feasibility of our approach and 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 in terms of flexibility and adaptability to different situations of our proposed platform we employed three optimization programs based on well-known algorithms from the literature. 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 as possible. On the other hand, the OptimizeGroup program gathered all the load in only 3 nodes of the infrastructure most of the time. The last program, OptimizeHops, brought connected virtual machines close to each other considering the network topology of the underlying infrastructure. The mean hop count was reduced close to 1 hop, although the maximum hop count was never reduced further than 4 hops.

In future investigations we intend to write optimization programs to improve the performance of the running application, possibly based on metrics personalized by the *End-user* and collected from inside the application. We also intend to perform further evaluations of scalability and measure the impact of optimizations (specially migrations) on the performance of running applications.

References

[1] Rackspace Cloud Computing, Openstack Cloud Software, 2010. <http://openstack.org/> (accessed February, 2012).

[2] Eucalyptus, The Open Source Cloud Platform, 2009. <http://open.eucalyptus.com/> (accessed February, 2012).

[3] OpenNebula, The Open Source Solution for Data Center Virtualization, 2008. <http://www.opennebula.org/> (accessed February, 2012).

[4] R. Moreno-Vozmediano, R. Montero, I. Llorente, Key challenges in cloud computing: enabling the future Internet of services, IEEE Internet Comput. 17 (4) (2013) 18–25, http://dx.doi.org/10.1109/MIC.2012.69.

[5] Libvirt, Libvirt: The Virtualization API – Version 0.9.9, January 2012. <http://www.libvirt.org> (accessed February, 2012).

[6] N. McKeown, T. Anderson, H. Balakrishnan, et al., Openflow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2008) 69–74. http://dx.doi.org/10.1145/1355734.1355746.

[7] M. Carvalho, R. Esteves, G. Rodrigues, L.Z. Granville, L.M.R. Tarouco, A cloud monitoring framework for self-configured monitoring slices based on multiple tools, in: 9th International Conference on Network and Service Management 2013 (CNSM 2013), Zürich, Switzerland, 2013, pp. 180–184.

[8] LXC, Linux Containers, September 2012. <http://lxc.sourceforge.net/> (accessed September, 2012).

[9] Open vSwitch, An Open Virtual Switch, September 2012. <http://openvswitch.org/> (accessed September, 2012).

[10] Mininet, Mininet: Rapid Prototyping for Software Defined Networks, September 2012. <http://openflow.org/mininet> (accessed September, 2012).

[11] L.M. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner, A break in the clouds: towards a cloud definition, SIGCOMM Comput. Commun. Rev. 39 (1) (2008) 50–55. http://dx.doi.org/10.1145/1496091.1496100.

[12] T. Benson, A. Akella, A. Shaikh, S. Sahu, Cloudnaas: a cloud networking platform for enterprise applications, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, (SOCC), ACM, New York, NY, USA, 2011, pp. 8:1–8:13. http://dx.doi.org/10.1145/2038916.2038924.

[13] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, M. Tsugawa, Science clouds: early experiences in cloud computing for scientific applications, in: Cloud Computing and Its Applications (CCA), Chicago, USA, 2008, pp. 5.

[14] K. Keahey, I. Foster, T. Freeman, X. Zhang, Virtual workspaces: achieving quality of service and quality of life in the grid, Sci. Program. 13 (4) (2005) 265–275.

[15] B. Sotomayor, R. Montero, I. Llorente, I. Foster, Virtual infrastructure management in private and hybrid clouds, IEEE Internet Comput. 13 (5) (2009) 14–22, http://dx.doi.org/10.1109/MIC.2009.119.

[16] Reservoir, Resources and Services Virtualization without Barriers, 2008. <http://www.reservoir-fp7.eu/> (accessed May, 2014).

[17] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, D. Zagorodnov, The eucalyptus open-source cloud-computing system, in: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID), 2009, pp. 124–131. http://dx.doi.org/10.1109/CCGRID.2009.93.

[18] P. Sempolinski, D. Thain, A comparison and critique of eucalyptus, opennebula and nimbus, in: IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), 2010, IEEE, 2010, pp. 417–426.

[19] RACKSPACE Cloud Computing, OpenStack Quantum Project, 2011. <http://wiki.openstack.org/Quantum> (accessed April, 2012).

[20] Ofelia Control Framework (ocf), 2011. <http://fp7-ofelia.github.io/ocf/>.

[21] W. Kopsel, Ofelia – Pan-European Test Facility for Openflow Experimentation, 2011. <http://www.fp7-ofelia.eu/>.

[22] protoGENI, 2012. <http://www.protogeni.net/trac/protogeni>.

[23] GENI, Global Environment for Network Innovations (06 2011). <http://www.geni.net/>.

[24] Y. Sun, T. Harmer, A. Stewart, P. Wright, Mapping application requirements to cloud resources, in: Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science, vol. 7155, Springer, Berlin Heidelberg, 2012, pp. 104–112. http://dx.doi.org/10.1007/978-3-642-29737-3_12.

[25] F. Wuhib, R. Stadler, H. Lindgren, Dynamic resource allocation with management objectives – implementation for an openstack cloud, in: 8th International Conference on Network and Service Management (CNSM), 2012, pp. 309–315.

[26] R. Esteves, L.Z. Granville, H. Bannazadeh, R. Boutaba, Paradigm-Based Adaptive Provisioning in Virtualized Data Centers, in: Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM), Ghent, Belgium, 2013, pp. 169–176.

[27] L. Guo, Y. Guo, X. Tian, Ic cloud: A design space for composable cloud computing, in: 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010, pp. 394–401. http://dx.doi.org/10.1109/CLOUD.2010.18.

[28] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, R. Han, Elastic application container: a lightweight approach for cloud resource provisioning, in: 2012 IEEE 26th International Conference on Advanced Information Networking and Applications (AINA), 2012, pp. 15–22. http://dx.doi.org/10.1109/AINA.2012.74.

[29] A. Ali-Eldin, J. Tordsson, E. Elmroth, An adaptive hybrid elasticity controller for cloud infrastructures, in: Network Operations and Management Symposium (NOMS), 2012 IEEE, 2012, pp. 204–212. http://dx.doi.org/10.1109/NOMS.2012.6211900.

[30] M. Hasan, E. Magana, A. Clemm, L. Tucker, S. Gudreddi, Integrated and autonomic cloud resource scaling, in: IEEE/IFIP 3rd Workshop on Cloud Management (CloudMan), 2012, pp. 1327–1334. http://dx.doi.org/10.1109/NOMS.2012.6212070.

[31] Open Grid Forum, Open Cloud Computing Interface, 2012. <http://occi-wg.org/> (accessed September, 2012).
[32] Distributed Management Task Force (DMTF), Cloud Infrastructure Management Interface (CIMI) – Version 1.0.0, 2013. <http://www.dmtf.org/standards/cloud> (accessed May, 2013).
[33] Amazon, Amazon Elastic Compute Cloud (amazon ec2), 2013. <http://aws.amazon.com/ec2/> (accessed May, 2013).
[34] A. Strømmen-Bakhtiar, A.R. Razavi, Cloud computing business models, in: Cloud Computing for Enterprise Architectures, Springer London, 2011, pp. 43–60. http://dx.doi.org/10.1007/978-1-4471-2236-4_3.
[35] VXDL Forum, Virtual Infrastructure Description Language (VXDL) – Version 2.0, May 2011. <http://www.vxdlforum.org/> (accessed February, 2012).
[36] Distributed Management Task Force (DMTF), Open Virtualization Format (OVF), January 2010. <http://dmtf.org/standards/ovf> (accessed April, 2012).
[37] Nagios 3.5.0, 2013. <http://www.nagios.org/>.
[38] G. Koslovski, S. Soudan, P. Vicat-Blanc, Modeling cloud computing and cloud networking with vxdl, in: World Telecommunications Congress 2012, Cloud Computing in the Telecom Environment workshop, Miyazaki, Japan, 2012.
[39] W.P. de Jesus, J.A. Wickboldt, L.Z. Granville, Provinet an open platform for programmable virtual network management, in: Proceedings of 37th IEEE Computer Software and Applications Conference (COMPSAC), Kyoto, Japan, 2013, pp. 329–338.
[40] Django Software Foundation, Django 1.4, March 2012. <https://www.djangoproject.com/weblog/2012/mar/23/14/> (accessed April, 2012).
[41] POX, Pox Openflow Controller, 2012. <http://www.noxrepo.org/pox/about-pox/> (accessed September, 2012).
[42] OpenWrt: Wireless Freedom (Backfire 10.03.1), 2013. <https://openwrt.org/>.
[43] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, B. Ohlman, A survey of information-centric networking, IEEE Commun. Mag. 50 (7) (2012) 26–36, http://dx.doi.org/10.1109/MCOM.2012.6231276.
[44] M. Chowdhury, M. Rahman, R. Boutaba, ViNEYard – virtual network embedding algorithms with coordinated node and link mapping, IEEE/ACM Trans. Netw. 20 (1) (2012) 206–219.
[45] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, J. Rexford, Virtual routers on the move – live router migration as a network-management primitive, SIGCOMM Comput. Commun. Rev. 38 (2008) 231–242.
[46] X. Meng, V. Pappas, L. Zhang, Improving the scalability of data center networks with traffic-aware virtual machine placement, in: Proceedings of IEEE INFOCOM 2010, 2010, pp. 1–9. http://dx.doi.org/10.1109/INFCOM.2010.5461930.

Juliano Araujo Wickboldt is a Ph.D. student at the Federal University of Rio Grande do Sul (UFRGS), in Brazil. He achieved his B.Sc. degree in Computer Science at Pontifical Catholic University of Rio Grande do Sul in 2006. He also holds an M.Sc. degree from UFRGS conducted in a joint project with HP Labs Bristol and Palo Alto. Juliano was intern at NEC Labs Europe in Heidelberg, Germany for one year between 2011 and 2012. His current research interests include cloud resource management and software-defined networking.

Rafael Pereira Esteves is a Ph.D. student in computer science at the Institute of Informatics (INF) of the Federal University of Rio Grande do Sul (UFRGS), Brazil. Rafael received his B.Sc. and M.Sc. degrees in computer science from the Department of Informatics of the Federal University of Para, Brazil in 2007 and 2009, respectively. His research interests include network and service management, network virtualization, cloud computing, and software-defined networking.

Marcio Barbosa de Carvalho is a MSc. student at the Federal University of Rio Grande do Sul (UFRGS), in Brazil. He achieved his BSc degree in Computer Science at Federal University of Rio Grande do Sul (UFRGS), in 2010. His current research interests include management of cloud environments and virtualized infrastructures specially in regards to monitoring aspects of these environments.

Lisandro Zambenedetti Granville is associate professor at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS), Brazil. He received his M.Sc. and Ph.D. degrees, both in computer science, from UFRGS in 1998 and 2001, respectively. He is member of the Brazilian Internet Committee (CGI.br). He has served as a TPC member for many important events in the area of computer networks (e.g., IM, NOMS, and CNSM) and was TPC co-chair of DSOM 2007 and NOMS 2010.